COMP 551: Applied Machine Learning

Lecture Notes

Étienne Fortier-Dubois

Winter 2018

Contents

1 Introduction to machine learning

Imagine two tasks. In the first task, there are four pictures of people, one of which is male, and the goal is to identify the male face. In the second task, the goal is to compute $154443586 \times 29506220$. Which one is the most difficult task? For a human, the second is difficult while the first is trivial. For a computer, the reverse is true.

Alan Turing asked: Can machines think? He devised the Turing test, also called the imitation game, in which a computer must behave in a way indistiguishable from a human to pass the test.

1.1 What can machines do?

They can beat humans at chess (IBM Deep Blue, 1997). They can fly a helicopter (Stanford Autonomous Helicopter, 2004). They can answer questions (IBM Watson, 2011). They can play video games like Atari Breakout (Google DeepMind's Deep Q-learning, 2013). They can beat humans at Go (AlphaGo, 2016).

As of 2018, there has been a lot of progress in some practical applications. These include speech recognition and synthesis, language translation, medical analysis and drug discovery, financial applications, personal AI assistants (like Google Home and Amazon Alexa), robotics, self-driving cars, and even fiding new planets.

1.2 What is machine learning?

Machine learning is the study of algorithms that improve their performance P at some task T with experience E. The learning therefore involves the variables $< P, T, E >$

First example: The task T is to classify emails as spam or non-spam. P is the accuracy of the classification. E is a set of example maisls labelled as spam or non-spam.

Second example: T is the detection of fraudulent credit card transactions. P is accuracy. E is a set of historical transactions marked as legit or fraudulent.

1.3 Prediction problems

This is one of the most common types of machine learning problems. Given variable x, predict variable y.

Examples: Given the size of the house, predict the selling price. Given the current stock price of a company (at time t), predict the same after 10 minutes ($t + 10$). In these two examples, y is a real number ($y \in \mathbb{R}$). These problems are called **regression problems**.

Given an image of a hand-written digit, preduct the digit (e.g. for reading cheques). Given the temperature, humidity and wind, predict whether it will rain or not. In these two examples, y is categorical ($y \in \{0, 1, ..., 9\}$ in the first case, $y \in \{yes, no\}$ in the second case). These problems are called **classification problems**.

1.4 What is learning?

Definition of learning: Given the value of an input x, make a good precition of output y, denoted $\hat{y}$.

Definition of supervised learning: Given a traning set $\{x^i, y^i\}$ from i to N, find a function $f : x \to y$ such that a prediction $\hat{y}$ can be made from x. In other words, given a set of data points, find a way to predict further data points.

2 Introduction to machine learning, continued

2.1 Curve fitting example

This is a regression problem. Suppose we have data points created with a function, and we want to know what the function is. In our example (but not in real-life), we know that the function is $\sin(2\pi x)$. In addition, there is some Gaussian noise ϵ so that the data points do not a perferct curve. Thus

$$y = \sin(2\pi x) + \epsilon$$

Let's assume a linear model. The goal is to learn the parameters $w\{w_0, w_1\}$ of the model. The prediction equation is

$$\hat{y} = w_0 + w_1 x$$

If we do not add w_0, we can only cover lines passing through the origin, and so w_0 is known as the **bias** in ML literature. Thus the prediction depends on both x and w, and we write $\hat{y}(x; w)$.

The goal is that $\hat{y}(x; w)$ be as close as possible to y. This is done by minimizing the error function

$$E(w) = \frac{1}{2} \sum_{n=1}^{N} \{\hat{y}(x^n; w) - y^n\}$$

$E(w)$ is a quadrative function of the parameters. Therefore, the derivative of $E(w)$ with respect to w is linear.

But note that this model does not fit the data we have very well (since it was generated with a sin function). We can consider higher order polynomials. Using the fact that $x^0 = 1$, we can write

$$\begin{aligned} \hat{y}(x; w) &= w_0 x^0 + w_1 x^1 + ... + w_M x^M \\ &= \sum_{j=0}^{M} w_j x^j \end{aligned}$$

The error function is actually still a quadratic function of the parameters w, because the model is still linear. So the solution for w is constant.

In our example, a third-degree polynomial would appear to fit the data well. A ninth-degree polynomial actuall fits the data perfectly—there is zero training error. Which one is the best solution?

Knowing the original function (sin), the third-degree polynomial seems better, because it could predict better further data points. With $M = 9$, we

obtain an excellent fit to the training data, but it is a poor model. This is called **overfitting**.

But we know this only because we already know the true function. What to do when we don't?

2.1.1 Test set

The solution is to use a separate test set. When training the model, use data points labeled as the training set. Then, to assess how good the model is, use other data points—the test set. This is a proxy for performance.

We can evaluate performance (and compare the training with the testing) with root mean square error:

$$E_{RMS} = \sqrt{\frac{2E(w^*)}{N}}$$

If we plot the E_{RMS} against M (polynomial degree), we'll see low error for $M = 9$ for the training, but high error for the testing. We should pick the degree that minimizes testing error.

Why does this happen in the example with $M = 9$? As M increases, the magnitude of the coefficients of the weight parameters (w_i) gets larger. In fact, if we have 10 data points and 10 weights (w_0 to w_9), then the model finds weight such that it can fit each data point perfectly.

Solution 1 to overfitting is to simply use more data points. As you add them, the model will approximate the curve better and better (even as M stays constant).

Solution 2 is to add a penalty term to the error function to discourage the coefficients from reaching large values. For example:

$$E(w) = \frac{1}{2}\sum_{n=1}^{N}\{\hat{y}(x^n; w) - y^n\}^2 + \frac{\lambda}{2}||w||^2$$

where λ is a something coefficient. If λ is too high or too small, the error will be high; thus there is a region of values of λ that is optimal, and it is crucial to choose a correct λ.

We call λ a **hyperparameter** of the model. The difference between a hyperparameter and a parameter is that the former is fixed, and the second is learned.

2.1.2 Model selection

But how do we choose λ? Can we use the test set to choose it? No. The test set is supposed to be a proxy for completely new x.

The solution is to separate a third set, called **validation set**. Our initial data points will be divided into a training, a validation, and a testing set.

Validation works like this: (1) For different values of λ, train the model and compute the validation performance. (2) Pick the value of λ that has the best validation performance. (3) Compute the *test* performance for the model with the chosen λ. This is called **model selection**.

Note that M, i.e. the degree of the polynomial, is also a hyperparameter. This leads us to solution 3 to overfitting: try different values of M and select the value of M based on validation performance. This is another form of model selection. There are many other solutions to overfitting.

2.2 Machine learning pipeline

These are the usual steps to go through when solving a machine learning problem. In this course, we will spend a lot of time on steps 4 (model selection) and 5 (error function).

1. Define the input x and output y.
2. Collect examples for the the task.
3. Divide the examples into a training, validation, and test set.
4. Define your model: parameters and hyperparameters.
5. Define the error function (a.k.a. loss function) that you want to minimize.
6. For different values of hyperparemeters, learn model parameters (with the training set) by minimizing the loss function, and compute validation performance with the validation set.
7. Pick the best model based on validation performance.
8. Test the model with the test set.

2.3 Classification example

The curve fitting example was a regression problem. Here is an example of a classification problem.

Suppose we have a bunch of points that are either orange or blue, and are on plotted on two axes x_1 and x_2. (Typically, with two categories, we label them 0 and 1, or -1 and 1. Let's say blue = 0 and orange = 1). The goal is to be able to classify any new data point with known x_1 and x_2 as either blue or orange.

2.3.1 Model 1: linear model

We have $x = (x_1, x_2)$. We can generalize to $x_T = (x_1, x_2, ..., x_p)$. We also have parameters $w^T = (w_0, w_1, ..., w_p)$.

$$\hat{y} = w_0 + \sum_{j=1} P x_j w_j$$

$$= \sum_{j=0} P x_j w_j$$

where $x_0 = 1$. As a result, the scalar $\hat{y} = x^T w$ (inner product of two vectors). We need to learn the parameter vector w.

We do this the same way as in the linear regression example, by minimizing the squares (least squares). The residual sum of squares is:

$$RSS(w) = \frac{1}{2} \sum_{i=1}^{N} (y^{(i)} - x^{(i)^T} w)^2$$

We can write this in matrix notation. In the following, X is the data matrix; and y is the target vector.

$$X = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_p^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_p^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \cdots & x_p^{(N)} \end{pmatrix}, y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{pmatrix}$$

Rows in X (including the associated $y^{(n)}$) are **examples** (data points), and columns are **features** of the data. X is an $N \times p$ matrix; y is an N column vector; and w is a p row vector. Thus the residual sum of squares is:

$$RSS(w) = \frac{1}{2} (y - Xw)^T (y - Xw)$$

(We can check that the multiplications make sense. y is $N \times 1$, X is $N \times p$ and w is $p \times 1$. Thus Xw is $N \times 1$, and the transpose of $(y - Xw)$ is $1 \times N$. Multiplied with $(y - Xw)$, we have $(1 \times N)(N \times 1)$ which yields a scalar, the RSS.)

We can solve this in closed form. The solution is obtained by differenti-

ating with respect to w and setting the differential to 0.

$$\begin{aligned} -2\frac{1}{2}X^T(y - Xw) &= 0 \\ X^T(y - Xw) &= 0 \\ X^Ty - X^TXw &= 0 \\ X^TXw &= X^Ty \\ w &= (X^TX)^{-1}X^Ty \end{aligned}$$

Therefore, the closed form solution to the residual sums of squares is

$$w^* = (X^TX)^{-1}X^Ty$$

and the model with the learnt parameter w^* is

$$\hat{y} = {w^*}^T x$$

After we have computed $\hat{y}$ for a given data point, we classify it according to a decision boundary. For $\{0, 1\}$ classification,

$$\hat{y} = \begin{cases} 1 \text{ if } \hat{y} > 0.5 \\ 0 \text{ if } \hat{y} \leq 0.5 \end{cases}$$

and for $\{-1, 1\}$ classification,

$$\hat{y} = \begin{cases} -1 \text{ if } \hat{y} < 0 \\ 1 \text{ if } \hat{y} \geq 0 \end{cases}$$

where $\hat{y} = 0.5$ (in the first case) and $\hat{y} = 0$ (in the second case) is the decision boundary. In other words, a data point is classified as one or the other categories depending on whether the point falls on one or the other side of the boundary.

2.3.2 Model 2: nearest-neighbor

Use those observations in the training set T closest in input space to x to form the prediction $\hat{y}$.

$$\hat{y}(x) = \frac{1}{k} \sum_{x^i \in N_k(x)} y^i$$

where k is the number of neighbors. The neighborhood of x can be written N_k.

If $\hat{y}(x) > 0.5$ then classify as class 1; else, classify as class 0.

In other words, classify based on the classification of the neighbors ("majority voting" of the neighbors). This has the implicit assumption that the class distribution is **locally smooth**, i.e. the class of an object is similar to other objects in its neighborhood.

If we look at a 2D distribution of points, the decision boundary will look irregular, unlike in the linear model. Nearest-neighbor classification also responds to local clusters, and the decision boundary can create several disjoint areas if there are distinct clusters of points.

The output of the model depends on k. For instance, if $k = 1$, the decision boundary gets even more irregular, such that every point is correctly classified. But recall the overfitting lesson: the model where a cluster is made for every data point may not be valid for classifying new data points.

Similarly to the regression example, k is a hyperparameter of the algorithm, and can be chosen from a separate validation step. The hyperparameter should be picked such that error in the validation step is minimal.

Are there any other hyperparameters for the k-nearest neighbor algorithm? Yes: the metric to compute nearest-neighbors (here we use Euclidian distance by default).

2.3.3 Least squares vs. nearest-neighbors

In least squares (linear model), the decision boundary is very smooth and the algorithm is more stable, but there is a strong assumption that the decision boundary is linear. There is high bias and low variance.

In nearest-neighbors, the decision boundary is wiggly and depends on a handful of input points and their position. The algorithm is less stable. The assumption that the class distribution is locally smooth is not such a strong assumption. The bias is low and variance is high.

3 Statistical decision theory

3.1 Expected prediction error

In a regression problem, we have a real valued random input vector $x \in \mathbb{R}^P$; a real valued random output variable $y \in \mathbb{R}$; and the joint distribution of x, y: $Pr(x, y)$. The goal is to find a function $f(x)$ for predicting y given values of x.

We can evaluate the performance of any prediction function by defining a loss function $L(y, f(x))$ that penalizes errors in prediction by comparing it to the real data. A common loss function is squared error loss:

$$L(y, f(x)) = (y - f(x))^2$$

From this function, we can derive the expected prediction error (EPE):

$$\begin{aligned} EPE(f) &= E\left[(y - f(x))^2\right] \\ &= \int_x \int_y (y - f(x))^2 P(x, y) dx dy \\ &= \int_x \int_y (y - f(x))^2 P(x|y) dy \; P(x) dx \\ &= \int_x E_{y|x}\left[(y - f)^2 | X = x\right] P(x) dx \\ &= E_x E_{y|x}\left[(y - f)^2 | X = x\right] \end{aligned}$$

It suffices to minimize the EPE pointwise (differentiate):

$$f^* = \underset{f}{\operatorname{argmin}} \; E_{y|x}\left[(y - f)^2 | X = x\right]$$

$$\begin{aligned} \frac{\partial}{\partial f} E_{y|x}\left[(y - f)^2 | X = x\right] &= 0 \\ \frac{\partial}{\partial f} E_{y|x}(y^2 | X = x) + f^2 - 2E_{y|x}(y|X = x) f &= 0 \\ 2f - 2E_{y|x}(y|X = x) &= 0 \end{aligned}$$

The function that minimizes the EPE is therefore:

$$f^* = E_{y|x}(y|X = x).$$

This means that the best prediction of y at any point $X = x$ is the conditional mean(expected value of the prediction given the value of the data), when "best" is measured by average squared error. This is true for any model, not just linear regression.

3.2 Expected prediction error for k-nearest neighbors

The function that minimizes EPE for neareast-neighbor is:

$$\hat{f}(x) = Average(y_i|x_i \in N_k(x)).$$

In other words, the function averages the predictions made for the class of x for each of the k neighbors of x. This is similar to the regression EPE-minimization function, but with two approximations.

1. Expectation is approximated by averaging over sample data.
2. Conditioning at a point is relaxed to conditioning on some region "close" to the target point.

Note 1: For a large training data sample size N, the points in the neighorhood are likely to be close to x.

Note 2: As k gets larger, the average will get more stable.

Under mild regularity conditions on $P(x, y)$, we can show that as $N, k \to \infty$ such that $k/N \to 0$ (i.e., many less classes than samples), then $\hat{f}(x) \to E(y|x = n)$ (the function approximates the expected prediction error better).

Does that mean we have a universal function approximator? No. As the number of features increases, in high dimensions, we need a very large number of samples.

3.3 Local methods in high dimensions

Consider inputs uniformly distributed in a p-dimensional hypercube. Consider a hypercubical neighborhood about a target point to capture a fraction of the observations. In other words, we want to compute a fraction r of the unit volume.

The expected edge length is $e_p(r) = r^{1/p}$. For instance, in 10 dimensions, the edge length for a 1% fraction is $e_{10}(0.01) = 0.01^{1/10} \approx 0.63$. To cover 1% of the data, to form a local average, we must cover 63% of the range of each input variable.

Such neighborhoods are no longer "local". And if we reduce r, with fewer observations, variance will be high. This is known as the **Curse of Dimensionality**. In other words, to search for some number of nearest neighbors, we need to cover a larger fraction of the total volume when in higher dimensions. Note that intuition often breaks down in higher dimensions!

3.4 Expected prediction error for linear regression

For linear regression, the function that minimizes the EPE is:

$$\hat{f}(x) \approx x^T w$$

This is a model-based approach. We are specifying a model for the regression function.

$$w = [E(XX^T)]^{-1} E(XY)$$

Note: we have not conditioned on X. Rather, we have used our knowledge of the functional relationship to pool over values of X. As an approximation, we can replace the expectation in the w equation above by averages over training data.

3.5 Inductive bias

We have made the following assumptions:

1. Least squares assumes $f(x)$ is well approximated by a globally linear function.

2. k-nearest neighbors assumes $f(x)$ is well approximated by a locally constant function.

The assumptions made by any algorithm are known as the **inductive bias** of the algorithm.

We will cover a lot of model-based algorithms. For example, additive models:

$$f(x) = \sum_{j=1}^{P} f_j(r_j).$$

This retains the additivity of the linear model, but each coordinate function f_j is arbitrary.

3.6 Non-squared error loss

We have used squared error loss, L_2. We could also use L_1:

$$E|y - f(x)|.$$

The solution is different: $\hat{f}(n) = \text{median}(y|X = x)$. This is a different measure of location, and its estimates are more robust than those for the conditional mean.

L_1 criteria have discontinuities on their derivatives, however, which have hindered their widespread use.

3.7 Expected prediction error for classification

For classification problems, the same paradigm is valid, but we need a different loss function. The loss function should penalize prediction errors.

Let the output categorical variable G be the set of all possible classes $G_1, ..., G_k$. Let $\hat{G}$ be the estimate and G the true class.

The loss function is a $k \times k$ matrix where k is the cardinality of G. The function $L(k, l)$ is the price for classifying an observation belonging to class k in class l. For instance, classifying a person as having cancer when in fact they don't might carry a cost (and this cost would be different from classifying a person who has cancer as someone who doesn't).

A common loss function is the **0/1 loss function**. This function assigns a penalty of 0 for a correct classification and 1 for a misclassification.

The expected prediction error for a classification problem is

$$EPE = E\left[L(G, \hat{G}(X))\right].$$

Expectation is with respect to $P(G, X)$. Thus

$$EPE = E_X \sum_{k=1}^{k} L\left[G_k, \hat{G}(X)\right] P(G_k|X).$$

Again, it suffices to minimize EPE pointwise:

$$\hat{G}(x) = \underset{g \in G}{\operatorname{argmin}} \sum_{k=1}^{k} L(G_k, g) P(G_k|X = x)$$

For the 0/1 loss function, this yields a solution that can be written in either of two forms:

$$\hat{G}(x) = \underset{g \in G}{\operatorname{argmin}} \left[1 - P(g|X = x)\right]$$

$$\hat{G}(x) = \underset{g \in G}{\operatorname{argmax}} \left[P(g|X = x)\right]$$

This solution is known as the **Bayes classifier**. We classify the most probable class, using the conditional distribution $P(G|X)$. The error rate of the Bayes classifier is called the **Bayes rate**. This is the optimal Bayesian decision boundary.

The k-nearest neighbors method directly approximates this solution. In this algorithm, we do majority voting in the neighborhood. The approximations are:

1. The conditional probability at a point is relaxed to the conditional probability with its neighborhood.

2. Probabilities are estimated by training sample proportions.

4 Linear regression

The simplest model for linear regression is:

$$\hat{y}(x; w) = w_0 + w_1 x_1 + ... + w_p x_p$$

This is a linear function of the parameters w. This is also a linear function of the inputs $x_1, ..., x_p$, which is a limitation.

4.1 Linear basis function models

We can write the linear regression model like this:

$$\hat{y}(x; w) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(x)$$

where $\phi_j(x)$ is a **basis function**, a fixed non-linear function. Assuming $\phi_0(x) = 1$, we get

$$\begin{aligned} y(x; w) &= \sum_{j=0}^{M-1} w_j \phi_j(x) \\ &= w^T \phi(X) \end{aligned}$$

where $w = (w_0, ..., w_{M-1})^T$ and $\phi = (\phi_0, ..., \phi_{M-1})^T$.

The $\hat{y}(x; w)$ is a non-linear function of x. It is still linear with respect to the parameters, and hence is a linear model.

Below we describe some basis functions.

4.1.1 Polynomial basis function

If the basis function is

$$\phi_j(x) = x^j$$

then we get

$$\hat{y}(x; w) = w_0 + w_1 x + w_2 x^2 + ... + w_{M-1} x^{M-1}.$$

4.1.2 Gaussian basis function

$$\phi_j(x) = \exp\left\{ -\frac{(x - \mu_j)^2}{2s^2} \right\}$$

where μ_j governs the locations of the basis function in input space, and s governs the spatial scale. Note: there is no probabilistic interpretation. The normalization constant is unimportant here because the basis functions will be multiplied by the adaptive parameters w_j.

4.1.3 Sigmoidal basis functions

$$\phi_j(x) = \sigma\left(-\frac{(x-\mu_j)}{s}\right)$$

where we have a logistic sigmoid function

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

4.1.4 Identity basis function

$$\phi(x) = x$$

If we have $\hat{y}(x;w) = w_0 + w_1x_1 + ... + w_px_p$, then $\phi_j(x) = x_j = \sum_{j=0}^{P} w_j\phi_j w$.

4.2 Least squares

$$RSS(w) = (y-\phi w)^T(y-\phi w)$$

$$w^* = \underset{w}{\operatorname{argmin}}\, RSS(w)$$

$$w^* = (\phi^T\phi)^{-1}\phi^T y$$

$(\phi^T\phi)^{-1}\phi^T$ is the Moore-Penrose pseudo-inverse of matrix ϕ, i.e.

$$\phi^+ = (\phi^T\phi)^{-1}\phi^T.$$

Note: if ϕ is square and invertible, then $\phi^+ = \phi^{-1}$.

4.2.1 Geometry of least squares

Consider an N-dimensional space whose axes are given by $y^{(n)}$ so that $y = (y^{(1)}, ..., y^{(n)})^T$ is a vector in this space. Each basis function $\phi_j(x^{(n)})$, evaluated at N data points, can also be represented as a vector in the same space, denoted by φ_j.

φ_j is the jth column of ϕ, while $\phi(x^{(n)})$ is the nth row of ϕ.

If the number M of basis functions is smaller than the number N of data points, then the M vectors $\phi^j(x^{(n)})$ will span a linear subspace S of dimensionality M.

We know that $\hat{y}$ is a linear combination of the vectors ϕ_j. So it lives anywhere in the M-dimensional subspace.

SSE is the squared Euclidian distance between y and $\hat{y}$. Thus the least squares solution for w corresponds to that choice of $\hat{y}$ that lies in the subspace S and that is closest to y. This solution corresponds to the orthogonal projection of y onto the subspace S.

First issue: There may be columns of X that are not linearly independent; then X is not of full rank. For instance, if two features are perfectly correlated (e.g. date of birth and age), then $X^T X$ is singular and w^* is not uniquely defined. The solution is to filter the redundant features.

Second issue: There can also be rank deficiency if there are more features p than the number of training cases N. The solution is to reduce the number of features (dimensionality reduction) or add regularization (L1, L2).

4.3 Learning by gradient descent

This is a fundamental learning algorithm. Consider a one-dimensional input. The linear regression model is $\hat{y}(x; w) = w_0 + w_1 x$. The parameters are $w = (w_0, w_1)$. The loss function is

$$L(w_0, w_1) = \frac{1}{2N} \sum_{i=1}^{N} \left(\hat{y}(x^i; w) - y^i\right)^2$$

The goal is to minimize $L(w_0, w_1)$.

This is a simple example which we can solve analytically. But there are many situations where analytical solutions are not computationally possible. Then we can use the gradient descent approach. Visually, it is about taking step after step downward on the loss surface in order to reach the minimum.

4.3.1 Gradient descent outline

1. Start with some random w_0, w_1.
2. Keep changing w_0, w_1 to reduce $L(w_0, w_1)$ until we (hopefully) end at a minimum.

Algorithm: repeat, until convergence, the following:

$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} L(w_0, w_1)$$

and simultaneously update $j = 0$ and $j = 1$.

α is called the step size or learning rate. If it is too small, gradient descent will be slow. If it is too large, gradient descent can overshoot the minimum and may fail to converge (or even diverge). We see that α is a hyperparameter of the algorithm. Furthermore, it is very sensitive and must be chosen carefully.

Note: gradient descent can converge to a local minimum even with the learning rate α fixed, because the gradient becomes less steep as we approach the minimum. The algorithm will then automatically make smaller steps.

4.3.2 Linear regression example

Suppose have the loss function

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

with two parameters w_0, w_1. We get the gradients

$$\frac{\partial L}{\partial w_0} = \frac{2}{2N} \sum_{i=1}^{N} \left(w_0 + w_1 x^{(i)} - y^{(i)} \right)$$

$$\frac{\partial L}{\partial w_1} = \frac{2}{2N} \sum_{i=1}^{N} \left(w_0 + w_1 x^{(i)} - y^{(i)} \right) x^{(i)}.$$

The algorithm will therefore repeat, until convergence, the updates

$$w_0 = w_0 - \alpha \frac{1}{N} \sum_{i=1}^{N} \left(\hat{y}(x^{(i)}; w) - y^{(i)} \right)$$

and

$$w_1 = w_1 - \alpha \frac{1}{N} \sum_{i=1}^{N} \left(\hat{y}(x^{(i)}; w) - y^{(i)} \right) x^{(i)}.$$

4.4 The special nature of the linear regression loss function

The loss function for linear regression is a convex function. Thus is has only one minimum, which is the global minimum. This is because of the use of least square loss with a linear model.

For non-linear models, we might get non-convex loss functions, and then there is no guarantee to reach a global minimum; we can get a local one.

4.5 Online or sequential learning

In an online context, data come in a continuous stream. We use **stochastic gradient descent**. The loss function for stochastic gradient descent is a sum of losses for every data point:

$$\text{loss} = \sum_n \text{loss}_n.$$

We can approximate the gradient of this total loss by the gradient of individual data point loss.

$$w = w - \alpha \frac{\partial}{\partial w} \text{loss}_n$$

The stochastic step only tries to approximate the true gradient. The approximation error can help escaping a local minimum.

4.5.1 Online gradient descent algorithm

With N data points, repeat until convergence:
for i in 1 to N {

$$w_0 = w_0 - \alpha(\hat{y}(x^{(i)}; w) - y^{(i)})$$
$$w_1 = w_1 - \alpha(\hat{y}(x^{(i)}; w) - y^{(i)})x^{(i)}$$

}

One **epoch** is a full iteration of the loop, i.e. one sweep of all the examples in the training set.

4.6 Multiple outputs

So far, we have considered only a single target variable y. However, y could be a vector ($y \in \mathbb{R}^k$). How does that affect the regression model?

A first solution could be a set of basis functions for each component of y, i.e. as if we had independent regression problems.

A better solution is to use the same set of basis functions to model all the components of the target vector:

$$\hat{y}(x; w) = W^T \phi(x)$$

where W is an $M \times k$ matrix of parameters.

$$W^* = (\phi^T \phi)^{-1} \phi^T Y$$

where Y is an $N \times k$ matrix. For an indivdual target variable,

$$w_k = (\phi^T \phi)^{-1} \phi y_k = \phi^t y_k.$$

The solution to the regression problem decouples between the different target variables. We only need to compute a single pseudo-inverse matrix which is shared by all of the vectors W_k.

5 Bias-variance tradeoff, regularization

5.1 Empirical risk minimization

Let $X \in \mathbb{R}^P$, $y \in \mathbb{R}$. Our hypothesis h is $X \to y$. We calculate the expected prediction error

$$\begin{aligned} EPE(f) &= [L(y, h(x))] \\ &= \iint [L(y, h(x))]\, P(X, y)\, dx\, dy \end{aligned}$$

This is known as the **risk** associated with hypothesis h, denoted $R(h)$.

Our goal is to find a hypothesis h^* among the class of functions for which the risk $R(h)$ is minimal. In mathematical terms,

$$h^* = \operatorname*{argmin}_{h \in H} R(h).$$

In general, $R(h)$ cannot be computed because the distribution $P(x, y)$ is unknown to the learning algorithm. However, we can compute an empirical approximation for this risk by using the training set. The empirical risk formula is thus:

$$R_{emp}(h) = \frac{1}{N} \sum_{i=1}^{N} L\left(y^{(i)}, h(x^{(i)})\right)$$

and empirical risk minimization is to find the $\hat{h}$ that minimizes the empirical risk.

$$\hat{h} = \operatorname*{argmin}_{h \in H} R_{emp}(h)$$

Note 1: There is no guarantee that $h^* = \hat{h}$

Note 2: The inductive bias of the algorithm restricts the hypothesis class. For instance, linear models restrict the hypothesis class to linear functions only. The restricted hypothesis class $H' \subseteq H$ may or may not contain h^*.

5.2 Bias-variance decomposition

Consider a regression problem with the least squares loss function:

$$L(y, h(x)) = (h(x) - y)^2$$

Then the hypothesis that minimizes risk is

$$h^* = \operatorname*{argmin}_{h \in H} E\left[L(y, h(x))\right] = E[y|x]$$

This is the regression function. Let's start with the loss function and derivate:

$$\begin{aligned}\{h(x) - y\}^2 &= \{h(x) - E[y|x] + E[y|x] - y\}^2 \\ &= (h(x) - E[y|x])^2 + (E[y|x] - y)^2 + 2(h(x) - E[y|x])(E[y|x] - y)\end{aligned}$$

Thus we can write

$$\begin{aligned}E\left[L(y, h(x))\right] &= E\left[\{h(x) - y\}^2\right] \\ &= E\left[(h(x) - E[y|x])^2\right] + E\left[(E[y|x] - y)^2\right] + 2E\left[(h(x) - E[y|x])(E[y|x] - y)\right]\end{aligned}$$

There are three terms in the latter equation. Let us examine them in turn. The first term is:

$$\begin{aligned}E\left[(h(x) - E[y|x])^2\right] &= \int_x \int_{y|x} (h(x) - E[y|x])^2 \, P(y|x) \, dy \, P(x) \, dx \\ &= \int_x (h(x) - E[y|x])^2 \, P(x) \, dx\end{aligned}$$

(We can do this because the squared term is not a function of y.). The second term is:

$$\begin{aligned}E\left[(E[y|x] - y)^2\right] &= \int_x \int_{y|x} (E[y|x] - y)^2 \, P(y|x) \, dy \, P(x) \, dx \\ &= \int_x \int_{y|x} \left(E[y|x]^2 + y^2 - 2E[y|x]y\right) \, P(y|x) \, dy \, P(x) \, dx \\ &= \int_x E[y|x]^2 + E[y^2] - 2E[y|x]^2 \, P(x) \, dx \\ &= \int_x E[y^2] - (E[y|x])^2 \, P(x) \, dx \\ &= E_X\left[var(y|x)\right]\end{aligned}$$

and the third term is:

$$\begin{aligned}E\left[(h(x) - E[y|x])(E[y|x] - y)\right] &= E_X E_{y|X}\left(h(x)E[y|x] - h(x)y - (E[y|x])^2 + yE[y|x]\right) \\ &= E_X\left(h(x)E[y|x] - h(x)E[y|x] - (E[y|x])^2 + E[y|x]^2\right) \\ &= E_X(0) \\ &= 0\end{aligned}$$

Therefore we can write that the expectation of the loss function is

$$E[L(y, h(x))] = \int_x (h(x) - E[y|x])^2 P(x) \, dx + \int_x var(y|x) P(x) \, dx$$

Note 1: The second term is the variance of the distribution y, averaged over x. It represents the intrinsic variability of the target data and can be regarded as noise. It is independent of $h(x)$, and is therefore an irreducible minimum value of the loss function.

Note 2: The first term is a function of $h(x)$. It is a minimum only when $h(x) = E[y|x]$:

$$(h(x) - E[y|x])^2$$

In practice, we have a dataset D containing only a finite number N of data points. We run our algorithm and find a hypothesis that we call $h(x; D)$. We can check the following:

$$\begin{aligned} E_D\left[\{h(x;D) - E[y|x]\}^2\right] &= \{E_D[h(x;D)] - E[y|x]\}^2 + E_D\left[\{h(x;D) - E_D[h(x;D)]\}^2\right] + 0 \\ &= (\text{bias})^2 + \text{variance} \end{aligned}$$

The **bias** is the extent to which the average prediction over all datasets varies around the average. The **variance** is the extent to which solutions for individual datasets vary around their average (the extent to which $h(x; D)$ is sensitive to a particular choice of dataset).

The key idea is

$$\text{Expected loss} = (\text{bias})^2 + \text{variance} + \text{noise}$$

where

$$\begin{aligned} (\text{bias})^2 &= \int \{E_D[h(x;D)] - E[y|x]\}^2 \, p(x)\, dx \\ \text{variance} &= \int \{E_D[h(x;D)] - E_D(h(x;D))\}^2 \, p(x)\, dx \\ \text{noise} &= \int \{E[y|x] - y\}^2 \, p(x,y)\, dx\, dy \end{aligned}$$

There is a tradeoff between bias and variance. Very flexible models have low bias, but high variance. Rigid models have high bias and low variance. Generally, the more complex a model, the highest the variance but the lowest the bias.

5.2.1 Occam's razor

"One should not increase, beyond what is necessary, the number of entities required to explain anything". In other words, seek the simplest explanation. This is kind of inductive bias: we make the assumption that simpler models are better.

5.2.2 Regression example

Suppose we have $L = 100$ datasets, each containing $N = 25$ data points sampled from a $\sin(2\pi x)$ curve. Our model is 24 Gaussian basis functions.

Our average prediction is

$$\bar{h}(x) = \frac{1}{L}\sum_{l=1}^{L} h^{(l)}(x)$$

and

$$(\text{bias})^2 = \frac{1}{N}\sum_{n=1}^{N}\left\{\bar{(}x^{(n)}) - E[y^{(n)}|x^{(n)}]\right\}^2$$

$$\text{variance} = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{L}\sum_{l=1}^{L}\left\{h^{(l)}(x^{(n)}) - \bar{h}(x^{(n)})\right\}^2$$

As the model increases in complexity, we reach a point where the empirical risk is low, but the true risk is high. This is overfitting. Underfitting occurs when both the true and the empirical risk are high. The best model is the one that minimizes the true risk, i.e. doesn't overfit or underfit.

5.3 Regularization

Adding a regularization term to an error function helps controlling overfitting. The total error function can be written as:

$$E_D(w) + \lambda E_w(w)$$

where λ is a **regularization coefficient** that controls the relative importance of $E_w(w)$.

For the following, we will use the case of the SSE loss function, for which

$$E_D(w) = (y - Xw)^T(y - Xw).$$

5.3.1 L2 regularization

With L2 regularization, also known as ridge regression or weight decay, $E_w(w) = w^T w$. We want to minimize the following:

$$\min_w (y - Xw)^T(y - Xw) + \lambda w^T w$$

Differentiating and setting to 0:

$$-2X^T(y - Xw) + 2\lambda w = 0$$

$$(X^T X + \lambda I)w = X^T y$$
$$w = (X^T X + \lambda I)^{-1} x^T y$$

The solution adds a positive constant to the diagonal of $X^T\lambda$ before inversion. This makes the problem non-singular even if $X^T X$ is not of full rank.

The unconstrained optimization problem (the minimization above) can be converted into a constrained optimization problem

$$\min_w (y - Xw)^T (y - Xw)$$

subject to $w^T w \leq \eta$ where η can be interpreted as the radius of a circle over the origin when in 2D. One can show that

$$\eta \propto \frac{1}{\lambda}.$$

If we visualize a 2D parameter configuration (with w_0 and w_1), there is a circle of radius η at the origin. Somewhere outside of that are the possible parameter configurations $\hat{w}$. The optimal solution is the intersection of $\hat{w}$ and the constraint circle, $\hat{w}_{ridge}$. If we increase the value of λ, we decrese η and the size of the circle, and thus $\hat{w}_{ridge}$ will have a smaller magnitude.

5.3.2 L1 regularization

With L1 regularization, also known as lasso regression, $E_w(w) = |w|$. We want to minimize the following:

$$\min_w (y - Xw)^T (y - Xw) + \lambda |w|.$$

There is no closed form solution to this, because the function is not differentiable at $w = 0$. The corresponding constrained optimization problem is:

$$\min_w (y - Xw)^T (y - Xw)$$

subject to $|w| \leq \eta$ where $\eta \propto \frac{1}{\lambda}$. The 2D visualization replaces the constraint circle with a constraint square with the vertices on the axes. The $\hat{w}_{lasso}$ will be on one of the vertices, which means one of the parameters will be zero.

As we increase the value of λ, η decreases and hence more parameters w_i will go to zero. We can say that L1 regularization performs feature selection by setting the weights of irrelevant features to zero. L1 regularization prefers sparse models.

6 Linear classification

Consider a k-class classification problem, where the classes are $c \in \{C_1, ..., C_k\}$. There are two stages in classification:

1. **Inference stage**: Use training data to learn a model for $P(C_k|x)$ (the probability that data point x belongs to class C_k).

2. **Decision stage**: Use posterior probabilities to make optimal class assignments.

$$\hat{C}(x) = \underset{c \in C}{\operatorname{argmin}} \sum_{k=1}^{k} L(C_k, c) P(C_k|X = x)$$

where $L(C_k, c)$ is the price for classifying in class C_k an observation truly belonging to class c. If we use a $0/1$ loss function, this becomes

$$\begin{aligned} \hat{C}(x) &= \underset{c \in C}{\operatorname{argmin}}[1 - P(c|X = x)] \\ &= \underset{c \in C}{\operatorname{argmax}}[P(c|X = x)] \end{aligned}$$

which is the Bayes classifier (requiring the class probabilities).

6.1 Approaches to classification

There are three approaches to solving classification problems: generative models, discriminative models, and discriminant-based models.

6.1.1 Generative models

In this approach, we solve the inference problem of determining the class-conditional densities $P(X|C_k)$ (the probability that the data belongs to a given class) for each class C_k individually. We also infer the class probabilities $P(C_k)$. Then we use Bayes' theorem:

$$P(C_k|X) = \frac{P(C_k)P(X|C_k)}{P(X)}$$

where

$$P(X) = \sum_{k} P(X|C_k)P(C_k)$$

to get the posterior probabilities (of being in class C_k given the data). It would also be possible to learn the joint distribution $P(X, C_k)$ and normalize to get $P(C_k|X)$.

Once $P(C_k|X)$ is found, use decision theory to determine the class. This is known as a generative model, because we can generate synthetic data points in the input space by using the learnt distribution.

6.1.2 Discriminative models

In this approach, we directly model $P(C_k|X)$ and then use decision theory to determine the class.

6.1.3 Discriminant-based models

In this approach, we find a function $f(X)$, called a discriminant function, that maps X directly to class labels. There is no probability involved.

6.2 Linearly separable problems

Consider a two-class problem, with classes C_1 and C_2. If we plot the data points, they might cluster into two areas that can be separated by a linear decision surface. If such a decision surface can indeed separate the classes exactly, then the problem is said to be linearly separable.

A linear model for classification is one that uses, as a decision surface, a linear function of the input vector X and is hence defined by $(D-1)$-dimensional hyperplanes with the D-dimensional input space.

6.3 Target variable for classification

For regression, the target variable is $t \in \mathbb{R}$. For classification, the target variable is the class $c \in \{C_1, ..., C_k\}$.

In a two-class problem, we say $t = 1$ represents class C_1, and $t = 0$ represents class C_2. Therefore the target is $t \in \{0, 1\}$ and can be interpreted as the probability that the class is C_1, with the probability taking only extreme values of 0 and 1.

When we have $k > 2$ classes, we cannot just consider $t \in \{1, 2, ..., k\}$, because this would create artificial distance between classes: C_1 would be closer to C_2 than C_5, for instance. The solution is to use a 1-of-k coding scheme. Thus, if $k = 5$ classes, then class 1 corresponds to $t_1 = (1, 0, 0, 0, 0)^T$, class 2 to $t_2 = (0, 1, 0, 0, 0)^T$, and so on. Each t_k can be interpreted as the probability that the class is C_k.

6.4 Discriminant functions

Recall that a discriminant function takes an input vector X and assigns it to one of k classes, denoted C_k. If the function is a linear discriminant, then the decision surface is a hyperplane.

6.4.1 Two-class scenario

A simple linear discriminant function would be:

$$y(x) = w^T x + w_0$$

where w is the weight vector, and w_0 the bias. The decision boundary in this case is $y(x) = 0$. If $y(x) \geq 0$, then the data point is in class C_1; if $y(x) < 0$, then the data point is in class C_2.

The bias w_0 can be considered as threshold. For instance, if $w_0 = -7$, then $w^T x$ has to be at least $+7$ to be classified as C_1.

Consider two points, x_A and x_B, both of which lie on the decision surface. We know that

$$y(X_A) = w^T x_A + w_0 = 0$$
$$y(X_B) = w^T x_B + w_0 = 0$$

and thus $w^T(x_A - x_B) = 0$. In other words, the w vector is orthogonal to every vector lying on the decision surface—which means w determines the orientation of the decision surface.

Therefore, any point x' that is on the decision surface and closest to the origin can be represented as $x' = \alpha w$ for some scalar α. Also, since x' is on the decision surface:

$$w^T x' + w_0 = 0$$
$$\alpha w^T w + w_0 = 0$$
$$\alpha = -\frac{w_0}{||w||^2}.$$

The distance from x' to the origin is

$$\begin{aligned} ||x'|| &= ||\alpha w|| \\ &= \alpha ||w|| \\ &= -\frac{w_0}{||w||^2} ||w|| \\ &= -\frac{w_0}{||w||} \end{aligned}$$

Thus, while w determines the orientation of the decision surface, w_0 determines the location of the decision function. The value of $y(x)$ gives a signed measure of the perpendicular distance γ of the point x from the decision surface.

Consider an arbitrary point x and let $x_\perp$ be its orthogonal projection onto the decision surface, so that

$$x = x_\perp + \gamma \frac{w}{||w||}$$

$$w^T x = w^T x_\perp + \gamma \frac{w^T w}{||w||}$$

$$w^T x + w_0 = w^T x_\perp + w_0 + \gamma \frac{w^T w}{||w||}$$

Notice that the latter equation contains both $y(x)$ and $y(x_\perp)$.

$$y(x) = y(x_\perp) + \gamma \frac{w^T w}{||w||}$$

$$y(x) = 0 + \gamma ||w||$$

$$\gamma = \frac{y(x)}{||w||}.$$

In summary,

1. The decision surface is perpendicular to w;
2. The displacement of the decision surface is controlled by the bias parameter;
3. The signed orthogonal distance of a general point x from the decision surface is given by $\frac{y(x)}{||w||}$.

6.4.2 Multiple classes scenario

There are three ways to do k-class classification: one-versus-the-rest, one-versus-one, and a single discriminant comprising k linear functions

With a one-versus-the-rest classifier, we train $k-1$ classifiers, each of which solves a two-class problem of separating points in class C_k from points not in that class.

With a one-versus-one classifier, we need $k(k-1)/2$ binary discriminant functions, one for each possible pair of classes. Then we perform majority voting among the discriminant functions for classification.

With the third approach, consider k linear functions of the form:

$$y_k(x) = w_k^T x + w_{k0}.$$

If $y_k(x) > y_j(x)$ for all $j \neq k$, then x belongs to C_k. The decision boundary between two classes C_k and C_j is

$$y_k(x) = y_j(x)$$

$$w_k x + w_{k0} = w_j x + w_{j0}$$
$$(w_k - w_j)^T x + (w_{k0} - w_{j0}) = 0,$$

similar to the decision boundary for two classes. Therefore analogous geometrical properties apply.

Note: decision regions of such a discriminant are always singly connected and convex. Consider $x_A, x_B \in R_k$, where R_k is the region corresponding to class C_k. Any point $\hat{x}$ that lies on the line connecting x_A and x_B can be expressed as

$$\hat{x} = \lambda x_A + (1 - \lambda) x_B$$

with $0 \leq \lambda \leq 1$. By the linearity of the discriminant,

$$y_k(\hat{x}) = \lambda y_k(x_A) + (1 - \lambda) y_k(x_B)$$

We know that $y_k(x_A) > y_j(x_A)$ and $y_k(x_B) > y_j(x_B)$ for all $j \neq k$. Therefore $h_k(\hat{x}) > y_j(\hat{x})$ and so $\hat{x} \in R_k$. Thus R_k is singly connected and convex.

7 Indicator regression, PCA, LDA

7.1 Least squares for classification

Consider a classification problem with k classes and a 1-of-k binary coding scheme for the target vector t.

We know that least squares approximates the conditional expectation $E[t|X]$ of the target values given the input vector. For the binary coding scheme, this conditional expectation is given by the vector of posterior probabilities, so it makes sense to use least squares. However, there is no guarantee for the values to be in the $(0, 1)$ range.

Each class C_k is described by its own linear model:

$$y_k(x) = w_k^T x + w_0$$

for $k = 1, ..., k$. We can write this in vector notation:

$$Y(X) = \tilde{W}^T \tilde{X}$$

where $\tilde{W}$ is a matrix whose kth column comprises the $D + 1$ dimensional vector $\tilde{w}_k = (w_{k0}, w_k^T)^T$, and $\tilde{X}$ is the augmented input vector $(1, X^T)^T$.

Consider the training dataset $\{X^{(n)}, t^{(n)}\}_{n=1}^N$. We define a matrix T whose nth row is the vector $t^{(n)^T}$, together with a matrix $\tilde{X}$ whose nth row is $\tilde{X}^{(n)^T}$. The sum of squares error function is thus:

$$E_D(\tilde{w}) = \frac{1}{2}\mathrm{Tr}\left\{(\tilde{X}\tilde{W} - T)^T(\tilde{X}\tilde{W} - T)\right\}$$

and the solution to this is

$$\begin{aligned}\tilde{W} &= (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T T \\ &= \tilde{X}^+ T\end{aligned}$$

where $\tilde{X}^+$ is the pseudoinverse. The discriminant funtion is

$$y(x) = \tilde{W}^T w\tilde{X} = T^T(\tilde{X}^+)^T\tilde{X}.$$

Note 1: If every target vector in the training set satisfies some linear constraint $a^T t^{(n)} + b = 0$ for some constants a, b, then the least squares model prediction for any value of x will satisfy the same constraint so that $a^T y(x) + b = 0$. Since we use 1-of-k encoding, the model's predictions will sum to 1 for any value of x. But the values are not constrained to be in the range $(0, 1)$.

Note 2: The least squares solution lacks robustness to outliers. If there are some members of a class that lie far from the main cluster, the decision boundary may end up within the main cluster, for instance. The sum of squares error function penalizes predictions that are "too correct" in that they lie a long way on the correct side of the decision boundary.

Note 3: In many cases, the least squares method doesn't work well. For instance, suppose there are three classes, with the green clas data points being located roughly between the blue and red classes. The classifier may assign only a very small region to the green class compared to the other two.

7.2 Principal component analysis (PCA)

Let $X \in \mathbb{R}^d$ be a random vector. We wish to find $k < d$ directions that capture as much as possible of the variance of X. Principal component analysis allows us to do that.

This is an example of unsupervised learning: there is no y. The applications of PCA include dimensionality reduction, lossy data compression, feature extraction, and data visualization.

7.2.1 Formal definition

We want $P \in \mathbb{R}^d$ (the direction) such that $||P|| = 1$, so as to maximize $var(P^T X)$. Why must the magnitude of P be 1? Because otherwise we could maximize the variance by letting $||P|| \to \infty$.

Let $\mu = EX$ and

$$S = cov(X) = E[(X - \mu)(X - \mu)^T].$$

For any $P \in \mathbb{R}^d$, the projection $P^T X$ has mean $E[P^T X] = P^T \mu$ and variance

$$\begin{aligned} var(P^T X) &= E[(P^T X - P^T \mu)^2] \\ &= E[P^T (X - \mu)(X = \mu)^T P] \\ &= P^T S P. \end{aligned}$$

The goal is to maximize the variance. To do this, we will use the spectral decomposition of S, which is $Q \Lambda Q^T$, where Q is a matrix whose orthonormal row vectors are the eigenvectors of S, and Λ is a diagonal matrix with the corresponding eigenvalues. Also note that $QQ^T = I$. We will also define

$y = Q^T P$.

$$\begin{aligned}
\max_{||P||=1} P^T S P &= \max_{P \neq 0} \frac{P^T S P}{P^T P} \\
&= \max_{P \neq 0} \frac{P^T Q \Lambda Q^T P}{P^T Q Q^T P} \\
&= \max_{y \neq 0} \frac{y^T \Lambda y}{y^T y} \\
&= \max_{y \neq 0} \frac{\lambda_1 y_1^2 + ... + \lambda_d y_d^2}{y_1^2 + ... + y_d^2} \\
&\leq \lambda_1
\end{aligned}$$

where equality is attained in the last step when $y = e_1$, i.e. $P = Q^T e_1 = u_1$ when u_1 is the first eigenvector of S. The first principal component is the first eigen vector of $cov(X)$.

Similarly, the k-dimensional subspace that captures as much of the variance of X as possible is simply the subspace spanned by the top k eigenvectors of $cov(X)$: $u_1, ..., u_k$.

7.2.2 Procedure

1. Compute the mean μ and the covariance matrix S of the data X.
2. Compute the top k eigenvectors $u_1, ..., u_k$ of S.
3. Project X onto $P^T X$, where P^T is th e $k \times d$ matrix whose rows are $u_1, ..., u_k$.

7.2.3 Can we use PCA dimensions for classification?

No, not the raw method: projecting the data onto the first principal component collapses the classes instead of discriminating them. A better solution is to find the projection that maximizes the class separation. This is the idea behind Fisher's linear discriminant, or LDA.

7.3 Linear discriminant analysis (LDA)

Consider a two-class problem in which there are N_1 points of class C_1, and N_2 points of class C_2. The mean vectors of C_1 and C_2 are:

$$m_1 = \frac{1}{N_1} \sum_{n \in C_1} X^{(n)}$$

$$m_2 = \frac{1}{N_2} \sum_{n \in C_2} X^{(n)}$$

The idea is to choose w so as to maximize $w^T m_2 - w^T m_1$, i.e. find the parameters that give the largest difference between the classes. So that the expression cannot be made arbitrarily large by increasing the magnitude of w, we constrain $||w|| = 1$. Thus:

$$\max_w w^T m_2 - w^T m_1$$

$$\text{subject to } w^T w = 1$$

We can write the Lagrangian formulation of the problem:

$$L = w^T(m_2 - m_1) + \lambda(w^T w - 1)$$

$$\frac{\partial L}{\partial w} = m^2 - m_1 + 2\lambda w = 0$$

$$w = \frac{-1(m_2 - m_1)}{2\lambda}$$

$$w \propto (m_2 - m_1)$$

With this approach, two classes that are well separated in the original space may have considerable overlap in the projected space. This is due to the strongly nondiagonal covariances of class distributions.

7.3.1 Solution proposed by Fisher

Maximize a function that will give a large separation between the projected class means, while also giving a small variance within each class, thereby minimizing class overlap.

We can express the within-class variance of the transformed data, for class C_k, as:

$$s_k^2 = \sum_{n \in C_k} (w^T x^{(n)} - w^T m_k)^2$$

and thus the total within-class variance, for two classes, is $s_1^2 + s_2^2$. The Fisher criterion for classification is:

$$J(w) = \frac{(w^T m_2 - w^T m_1)^2}{s_1^2 + s_2^2}$$

Let's rewrite both the numerator and bottom parts of this expression.

$$
\begin{aligned}
(w^T m_2 - w^T m_1)^2 &= (w^T(m_2 - m_1))^2 \\
&= w^T(m_2 - m_1)(m_2 - m_1)^T w \\
&= w^T S_B w
\end{aligned}
$$

where S_B is the betwee-class covariance matrix.

$$
\begin{aligned}
s_1^2 + s_2^2 &= \sum_{n \in C_1} \left(w^T(x^{(n)} - m_1)\right)^2 + \sum_{k \in C_2} \left(w^T(x^{(k)} - m_2)\right)^2 \\
&= \sum_{n \in C_1} w^T(x^{(n)} - m_1)(x^{(n)} - m_1)^T w + \sum_{k \in C_2} w^T(x^{(k)} - m_2)(x^{(k)} - m_2)^T \\
&= w^T S_w w
\end{aligned}
$$

where

$$
S_w = \sum_{n \in C_1} (x^{(n)} - m_1)(x^{(n)} - m_1)^T + \sum_{k \in C_2} (x^{(k)} - m_2)(x^{(k)} - m_2)^T.
$$

We get a new formulation of the Fisher criterion:

$$
J(w) = \frac{w^T S_B w}{w^T S_w w}
$$

which we can differentiate and set to 0 to find the maximum.

$$
(w^T S_B w) S_w w - (w^T S_w w) S_B w = 0
$$

$$
(w^T S_B w) S_w w = (w^T S_w w) S_B w
$$

We know that $S_B = (m_2 - m_1)(m_2 - m_1)^T$. Therefore, $S_B w$ is always in the direction of $(m_2 - m_1)$. Since we do not care about the magnitude of w, only about its direction, we can drop the scalars $(w^T S_B w)$ and $(w^T S_w w)$. We get

$$
S_w w \propto (m_2 - m_1)
$$

$$
w \propto S_w^{-1}(m_2 - m_1).
$$

Note: If the within-class covariance is isotropic, so that S_w is proportional to the unit matrix, then w is proportional to the class means.

This model is known as Fisher's linear discriminant. However, it is not a discriminant. It is a specific choice of direction for the projection of data. The projected data can be used to construct a discriminant by choosing a threshold θ, such that if $w^T x \geq \theta$ then x is in C_1, and if $w^T x < \theta$ then x is in C_2. θ can be found by minimizing the training error.

8 GLMs, GDA, evaluation metrics

8.1 Generalized linear models

In linear regression, the model prediction $y(x; w)$ was given by a linear function of the parameters w. In the simplest case, the model is also linear in the input variables:

$$y(x) = w^T x + w_0$$

where $y(x)$ is a real number.

For classification, we wish to predict discrete class labels or posterior probabilities that lie in the range (0, 1). To do this, we transform the linear function of w using a non-linear function $f(.)$ (called an **activation function**) so that

$$y(x) = f(w^T x + w_0)$$

Note that this model is no longer linear in the parameters, because $f(.)$ is non-linear.

The decision surface is given by

$$y(x) = \text{ constant}$$

$$f(w^T x + w_0) = \text{ constant}$$

$$w^T x + w_0 = f^{-1}(\text{constant})$$

$$w^T x + w_0 = \text{constant}$$

Therefore, decision surfaces are linear functions of x, even if $f(.)$ is non-linear. Models of this type are called **generalized linear models** (GLM). GLMs have complex analytical and computational properties compared to linear models, but they are still simpler than more general nonlinear models.

8.2 Probabilistic generative models

These GLMs perform the classification task by modelling the class-conditional densities, $P(X|C_k)$, and the class priors $P(C_k)$, and then use Bayes' theorem to compute posterior probabilities $P(C_k|X)$, i.e. the probability of being in class C_k given data point x.

8.2.1 2-class problem

Let's consider a 2-class problem, with classes C_1 and C_2. We have:

$$\begin{aligned} P(C_1|x) &= \frac{P(X|C_1)P(C_1)}{P(X)} \\ &= \frac{P(X|C_1)P(C_1)}{P(X|C_1)P(C_1) + P(X|C_2)P(C_2)} \\ &= \frac{1}{1 + \frac{P(X|C_2)P(C_2)}{P(X|C_1)P(C_1)}} \\ &= \frac{1}{1 + \exp(-\ln \frac{P(X|C_1)P(C_1)}{P(X|C_2)P(C_2)})} \\ &= \frac{1}{1 + \exp(-a)} \\ &= \sigma(a) \end{aligned}$$

where

$$a = \ln \left(\frac{P(x|C_1)P(C_1)}{P(x|C_2)P(C_2)} \right)$$

and $\sigma(a) = \frac{1}{1+e^{-a}}$ is the logistic sigmoid function.

The sigmoid function has range $(0, 1)$ and the symmetry property $\sigma(-a) = 1 - \sigma(a)$. The inverse of the sigmoid is called the logit function:

$$a = \ln \left(\frac{\sigma}{1 - \sigma} \right).$$

It is a log of the ratio of probabilities. The expression

$$\ln \left(\frac{P(X|C_1)}{P(X|C_2)} \right)$$

for the two classes is also known as the log odds.

8.2.2 k-class problem, $k > 2$

With several classes, Bayes' theorem looks like this:

$$\begin{aligned} P(C_k|X) &= \frac{P(X|C_k)P(C_k)}{\sum_j P(X|C_j)P(C_j)} \\ &= \frac{\exp(a_k)}{\sum_j \exp(a_j)} \end{aligned}$$

where $a_k = \ln(P(X|C_k)P(C_k))$. This is a normalized exponential, which is a multiclass generalization of the logistic sigmoid. This function is also called the **sotfmax function**, as it represents a smoothed version of the 'max' function. This is because if $a_k >> a_j$ for all $j \neq k$, then $P(C_k|X) \approx 1$ and $P(C_j|X) \approx 0$.

8.3 Gaussian discriminant analysis (GDA)

This is also known as probabilistic LDA. The assumptions are:

1. Class conditional densities are Gaussian.
2. All classes share the same covariance matrix.

The posterior class probability is given by:

$$P(x|C_k) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma|^{1/2}} \exp\left\{-\frac{1}{2}(X-\mu_k)^T\Sigma^{-1}(X-\mu_k)\right\}$$

where μ_k is the mean, Σ is the covariance matrix, and $|\Sigma|$ is the determinant of the covariance matrix.

8.3.1 Two-class case

$P(C_1|X) = \sigma(a)$ where

$$\begin{aligned}
a &= \ln\left(\frac{P(X|C_1)P(C_1)}{P(X|C_2)P(C_2)}\right) \\
&= \ln\left(\frac{P(X|C_1)}{P(X|C_2)}\right) + \ln\left(\frac{P(C_1)}{P(C_2)}\right) \\
&= \ln\exp\left\{-\frac{1}{2}(X-\mu_1)^T\Sigma^{-1}(X-\mu_1) + \frac{1}{2}(X-\mu_2)^T\Sigma^{-1}(X-\mu_2)\right\} + \ln\left(\frac{P(C_1)}{P(C_2)}\right) \\
&= \frac{1}{2}(X^T\Sigma^{-1}X - X^T\Sigma\mu_1 - \mu_1^T\Sigma^{-1}X + \mu_1^T\Sigma^{-1}\mu_1 \\
&\qquad - X^T\Sigma^{-1}X + X^T\Sigma^{-1}\mu_2 + \mu_2^T\Sigma^{-1}X - \mu_2^T\Sigma^{-1}\mu_2) + \ln\left(\frac{P(C_1)}{P(C_2)}\right) \\
&= -X^T\Sigma\mu_1 - X^T\Sigma^{-1}\mu_2 - \frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \ln\left(\frac{P(C_1)}{P(C_2)}\right) \\
&= X^T(\Sigma^{-1}(\mu_1-\mu_2)) - \frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \ln\left(\frac{P(C_1)}{P(C_2)}\right) \\
&= w^TX + w_0
\end{aligned}$$

where

$$w = \Sigma^{-1}(\mu_1 - \mu_2)$$

$$w_0 = -\frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \ln\left(\frac{P(C_1)}{P(C_2)}\right)$$

We have found a way to write a in $P(C_1|x) = \sigma(a)$. Therefore we can write:

$$P(C_1|X) = \sigma(w^T X + w_0).$$

Note: the quadratic terms in x from the exponents of the Gaussian densities have cancelled due to the assumption of the common covariance matrices.

The model is a linear function of x. Therefore, the decision boundary is also a linear function of x. The prior probabilities $P(C_k)$ enter the model only through the bias parameter w_0. Changes in prior information have the effect of making parallel shifts of the decision boundary.

8.3.2 Learning the parameters

The parameter of the GDA model are the class means μ_1 and μ_2, the covariance matrix Σ, and the prior probabilities $P(C_1)$ and $P(C_2)$. How can we learn these parameters?

Consider a dataset $\mathcal{D} = \left\{X^{(n)}, t^{(n)}\right\}_{n=1}^{N}$ of N examples. The target variable is $t_n = 1$ for class C_2 and $t_n = 0$ for class C_2. Also let the prior probabilities be $P(C_1) = \pi$ and $P(C_2) = 1 - \pi$.

Consider data point $x^{(n)}$. The joint probabilities of this data point being in classes C_1 and C_2, respectively, are:

$$P(X^{(n)}, C_1) = P(C_1)P(X^{(n)}|C_1) = \pi\mathcal{N}(X^{(n)}|\mu_1, \Sigma)$$

$$P(X^{(n)}, C_2) = P(C_2)P(X^{(n)}|C_2) = (1-\pi)\mathcal{N}(X^{(n)}|\mu_2, \Sigma)$$

where $\mathcal{N}$ is the normal (Gaussian) distribution. The likelihood of the data point $x^{(n)}, t^{(n)}$ is given by putting these together:

$$\left[\pi\mathcal{N}(x^{(n)}|\mu_1, \Sigma)\right]^{t^{(n)}} \left[(1-\pi)\mathcal{N}(x^{(n)}|\mu_2, \Sigma)\right]^{1-t^{(n)}}$$

Assuming that the examples are independent and identically distributed (i.i.d.), we can therefore write the posterior probability of the entire dataset like this:

$$P(\mathcal{D}|\pi, \mu_1, \mu_2, \Sigma) = \prod_{n=1}^{N}\left[\left[\pi\mathcal{N}(X^{(n)}|\mu_1, \Sigma)\right]^{t^{(n)}} \left[(1-\pi)\mathcal{N}(X^{(n)}|\mu_2, \Sigma)\right]^{1-t^{(n)}}\right]$$

This is a **likelihood function**. The goal of GDA is to find the parameters that maximize the likelihood function.

8.3.3 Maximum likelohood solution

Usually, we maximize the log of the likelihood function instead of the likelihood function itself.

$$\ln P(\mathcal{D}|\pi, \mu_1, \mu_2, \Sigma) = \sum_{n=1}^{N} \ln \left[t^{(n)} \ln \mathcal{N}(X^{(n)}|\mu_1, \Sigma) + (1 - t^{(n)}) \ln(1-\pi)\mathcal{N}(X^{(n)}|\mu_2) \right]$$

To find the optimal value for all parameters, we will perform maximization with respect to π, μ_1, μ_2, and Σ.

Maximization with respect to π:

$$\sum_{n=1}^{N} \left[t^{(n)} \ln \pi + (1 - t^{(n)}) \ln(1-\pi) \right] + \text{constant}$$

We differentiate with respect to π and set to 0.

$$\sum_{n=1}^{N} \frac{t^{(n)}}{\pi} + \frac{1 - t^{(n)}}{1-\pi}(-1) = 0$$

$$\sum_{n=1}^{N} \frac{t^{(n)}(1-\pi) - (1-t^{(n)})\pi}{\pi(1-\pi)} = 0$$

$$\sum_{n=1}^{N} t^{(n)} - t^{(n)}\pi - \pi + t^{(n)}\pi = 0$$

$$\left(\sum_{n=1}^{N} t^{(n)} \right) - N\pi = 0$$

Therefore,

$$\pi = \frac{\sum_{n=1}^{N} t^{(n)}}{N} = \frac{N_1}{N} = \frac{N_1}{N_1 + N_2}$$

where N_1 is the number of points in class C_1 and N_2 is the number of points in C_2. In other words, the prior probability of class C_1, π, is the fraction of points in C_1 (and $(1 - \pi)$ is the fraction of points in C_2).

Maximization with respect to μ_1:

$$\sum_{n=1}^{N} t^{(n)} \ln \mathcal{N}(x^{(n)}|\mu_1, \Sigma) + \text{constant}$$

$$= -\frac{1}{2} \sum_{n=1}^{N} t^{(n)} (x^{(n)} - \mu_1)^T \Sigma^{-1} (x^{(n)} - \mu_1) + \text{constant}$$

We differentiate with respect to μ_1 and set to 0.

$$\sum_{n=1}^{N} t^{(n)}(X^{(n)} - \mu_1) = 0$$

$$N_1\mu_1 = \sum_{n=1}^{N} t^{(n)}X^{(n)}$$

$$\mu_1 = \frac{1}{N_1}\sum_{n=1}^{N} t^{(n)}X^{(n)}$$

Thus μ_1 is the mean of all the input vectors $X^{(n)}$ that are assigned to class C_1. Similarly,

$$\mu_2 = \frac{1}{N_2}\sum_{n=1}^{N} t^{(n)}X^{(n)}.$$

The maximization with respect to Σ is a bit involved, and we will derive the solution here. The solution is

$$\Sigma = \frac{N_1}{N}S_1 + \frac{N_1}{N}S_2$$

where

$$S_1 = \frac{1}{N_1}\sum_{n\in C_1}(X^{(n)} - \mu_1)(X^{(n)} - \mu_1)^T$$

$$S_2 = \frac{1}{N_2}\sum_{n\in C_2}(X^{(n)} - \mu_2)(X^{(n)} - \mu_2)^T.$$

Note: if we relax the assumption of a shared covariance matrix, and allow each class conditional density $P(X|C_k)$ to have its own covariance matrix Σ_k, then we will obtain quadratic functions of X. This will lead to Quadratic Discriminant Analysis (QDA) and the decision boundary will not necessarily be linear.

8.4 Evaluation metrics for classification

Considering a two-class problem, with a 'positive' and a 'negative' class, the most basic evaluation metric is **accuracy**.

$$\text{Accuracy} = \frac{\text{Number of correctly classified instances}}{\text{Total number of instances}}$$

This metric has issues. For instance, consider a cancer prediction example. 9900 patients do not have cancer ("no"), and 100 have it ("yes"). A classifier that would give the default answer "no" for all cases would achieve 99% accuracy, but would be useless since it cannot predict who actually has cancer.

Not every error has the same cost. Classifying patients without cancer as "yes" (false positive) is an error that is less dangerous than classifying patients with cancer as "no" (false negative). We can summarize this in a confusion matrix:

	True value:	Positive	Negative
Prediction:	Positive	True positive	False positive
	Negative	False negative	True negative

As an example, suppose we have a problem with 60 positives and 40 negatives. The ideal confusion matrix would be

	True value:	Positive	Negative
Prediction:	Positive	60	0
	Negative	0	40

while

	True value:	Positive	Negative
Prediction:	Positive	20	30
	Negative	40	10

would be avery bad one. (Note that the columns sum to 60 and 40 respectively.)

From this we can define two other metrics, **precision** and **recall**.

$$\text{Precision} = \frac{\text{True positives}}{\text{True positives} + \text{False positives}}$$

$$\text{Recall} = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}}$$

In other words, precision answers the question: Out of all positively classified predictions, how many are truly positive? Recall answers the question: Out of all truly positive examples in the data, how many were correctly classified as positive?

Precision and recall can have different usefulness depending on the problem. In face password systems, precision is important (we don't want false positives, i.e. unauthorized people to gain access), whereas in an email spam classifier, recall is more important (we don't want to falsely classify an email as spam).

It is easy to achieve high precision at the cost of low recall (and vice-versa). For instance, a classifier that always says "yes" will have 100% recall (it will identify all true positives and have no false negatives) but bad precision (it

will have a lot of false positives). A classifier that always says "no" will have 100% precision but bad recall.

We can combine precision in recall to get the fourth evaluation metric, the **F1 measure**, which is the harmonic mean of precision and recall.

$$\text{F1} = \frac{2(\text{precision})(\text{recall})}{\text{precision} + \text{recall}}$$

9 Naïve Bayes, logistic regression

In the last, lecture, we saw generative models for classification, in which we compute $P(X|C_k)$ and $P(C_k)$, and then use Bayes' law to compute $P(C_k|X)$.

In the GDA model, the class conditional densities are Gaussians and share a covariance matrix. In QDA, the covariance matrices are separate.

The two models also differ in their number of parameters. If there are k classes ans M features per example, then GDA has kM parameters for the means, $M(M+1)/2$ parameters for the shared covariance matrix, and $k-1$ parameters for the prior probabilities $P(C_k)$. QDA has the same number of parameters for the means and the prior probabilities, but there are k covariance matrices, and thus $kM(M+1)/2$ parameters for them.

QDA has more parameters and is more difficult to use. On the other hand, the shared covariance matrix assumption is very restrictive.

9.1 Gaussian Naïve Bayes

The Naïve Bayes assumption is that the features are conditionally independent given the class label. Formally,

$$\begin{aligned} P(X|C_k) &= P(X_1 ... X_M|C_k) \\ &= \prod_{i=1}^{M} P(X_i|C_k) \end{aligned}$$

i.e. the probability of data point X of being in class C_k is equal to the combined probabilities of all features X_i of X to be in belong to class C_k. This assumption makes the covariance matrix diagonal. This model is known as **Gaussian Naïve Bayes** (GNB).

GNB requires kM parameters for the means, kM parameters for the variance and $k-1$ parameters for $P(C_k)$. This is less than QDA, but also has a nonlinear decision boundary.

Note 1: If the diagonal covariance matrix is shared among the classes, then the decision boundary will be linear. GNB with shared diagonal covariance matrices is known as Diagonal LDA (DLDA).

Note 2: We can learn the covariance matrix in all these methods by using maximum likelihood.

9.2 Summary of methods for continuous input features

When the input features are continuous, we need a covariance matrix, which can be either arbitrary or diagonal, and either shared among the classes, or not.

Covariance matrix	Shared?	Model	Decision boundary
Arbitrary	Yes	**LDA/GDA**	Linear
	No	**QDA**	Nonlinear
Diagonal	Yes	**DLDA**	Linear
	No	**GNB**	Nonlinear

9.3 Naïve Bayes with discrete features

Consider binary features $x_i \in \{0, 1\}$. If there are M inputs, then a general distribution would correspond to a table of 2^M numbers for each class, containing $2^M - 1$ independent variables.

By the Naïve Bayes assumption,

$$\begin{aligned} P(X|C_k) &= P(X_1...X_M|C_k) \\ &= P(X_1|C_k)P(X_2|C_k, X_1)...P(X_M|C_k, X_1, ..., X_{M-1}) \qquad \text{(Chain rule)} \\ &= P(X_1|C_k)P(X_2|C_k)...P(X_M|C_k) \qquad \text{(Naïve Bayes assumption)} \\ &= \prod_{i=1}^{M} P(X_i|C_k) \end{aligned}$$

Thus the probability of being in class C_k depends on the combined probabilities of each discrete feature to belong to class C_k. Each such feature probability $P(X_i|C_k)$ can be modelled as a Bernoulli with parameter μ_{ki}, i.e. $P(X_i|C_k) = 1$. We can rewrite the model as

$$P(X|C_k) = \prod_{i=1}^{M} (\mu_{ki})^{x_i} (1 - \mu_{ki})^{1-x_i}.$$

Note that this requires only M parameters for each class.

9.3.1 Two-class problem

The likelihood of the data is $P(\mathcal{D}|\mu_1, \mu_2, \pi) =$

$$\prod_{n=1}^{N} \left(\pi \prod_{i=1}^{M} (\mu_{1i})^{x_i^{(n)}} (1 - \mu_{1i})^{1-x_i^{(n)}} \right)^{t^{(n)}} \left((1 - \pi) \prod_{i=1}^{M} (\mu_{2i})^{x_i^{(n)}} (1 - \mu_{2i})^{1-x_i^{(n)}} \right)^{1-t^{(n)}}$$

Let's call the terms in the large parentheses x and y, respectively, and maximize the log likelihood.

$$\ln P(\mathcal{D}|\mu_1, \mu_2, \pi) = \sum_{n=1}^{N} \left[t^{(n)} \ln x + (1 - t^{(n)}) \ln y \right]$$

Differentiate:

$$\frac{\partial \ln P(D|.)}{\partial \mu_{1i}} = \frac{\partial}{\partial \mu_{1i}} \left(\sum_{n=1}^{N} \left[t^{(n)} \ln(\mu_{1i})^{x_i^{(n)}} (1-\mu_{1i})^{1-x_i^{(n)}} \right] \right)$$
$$= \frac{\partial}{\partial \mu_{1i}} \left(\sum_{n=1}^{N} \left[t^{(n)} x_i^{(n)} \ln \mu_{1i} + t^{(n)} (1-x_i^{(n)}) \ln(1-\mu_{1i}) \right] \right)$$

and set to zero:

$$\sum_{n=1}^{N} \left(t^{(n)} x_i^{(n)} \frac{1}{\mu_{1i}} + t^{(n)} (1-x_i^{(n)}) \frac{1}{(1-\mu_{1i})} (-1) \right) = 0$$

$$\sum_{n=1}^{N} \left(t^{(n)} x_i^{(n)} (1-\mu_{1i}) - t^{(n)} (1-x_i^{(n)}) \mu_{1i} \right) = 0$$

$$\sum_{n=1}^{N} \left(t^{(n)} x_i^{(n)} - t^{(n)} x_i^{(n)} \mu_{1i} - t^{(n)} \mu_{1i} + t^{(n)} x_i^{(n)} \mu_{1i} \right) = 0$$

$$\sum_{n=1}^{N} t^{(n)} \mu_{1i} = \sum_{n=1}^{N} t^{(n)} x_i^{(n)}$$

$$\mu_{1i} = \frac{\sum_{n=1}^{N} t^{(n)} x_i^{(n)}}{\sum_{n=1}^{N} t^{(n)}}$$

Thus μ_{1i} is equal to the number of examples where $x_i = 1$ and $t = 1$, divided by the number of examples where $t = 1$. Similarly, μ_{2i} is equal to the number of examples where $x_i = 1$ and $t = 0$, divided by the number of examples where $t = 0$.

In other words, the class mean for a given feature i is equal to the number of examples that are classified in the class and have this feature, divided by the total number of examples classified as part of this class.

Note: If the features are categorical instead of binary, then the maximum likelihood solution for the parameters of the categorical distribution would be the frequency of occurrence of that category in the given class.

9.3.2 Example application: Text classification

Given a document, which is a collection of words, classify the document into one of k topics (e.g. politics, sports, arts).

Assume there are M words in the vocabulary. Each document can then be represented as an M-dimensional feature vector $X = (x_1, ..., x_m)$ where

each x_i is 1 if word i is present in the document, and 0 otherwise. Then the probability that a document belongs to class "politics" is

$$\begin{aligned} P(C_{politics}|X) &= P(X|C_{politics})P(C_{politics}) \\ &= \prod_{i=1}^{M} P(x_i|C_{politics})P(C_{politics}) \end{aligned}$$

There an issue: given a limited number of examples per class, we might not see all words appear at least once in documents of every class. If $x_i = 0$ whenever $t = 1$ (i.e. if word i is absent of all documents in class 1), then $\mu_{1i} = 0$. This would make $P(C_k|X) = 0$ often. To avoid this, we can use Laplace smoothing.

9.3.3 Laplace smoothing

We calculate the mean for a given feature i and class k like this:

$$\mu_{ki} = \frac{(\text{Number of examples where } x_i = 1 \text{ and } t = k) + 1}{(\text{Number of examples where } t = k) + 2}$$

Thus, if there is no example for a class (e.g. no "arts" documents), it reduces to a prior probability of $1/2$. Note that this is a type of bias of the model.

If all examples in class k have $x_i = 1$ (e.g. all "sports" documents have the word "play"), then the probability that a document that does not have x_i belongs to class C_k becomes small:

$$P(x_i = 0|C_k) = \frac{1}{(\text{Number of examples where } t = k) + 2}.$$

In other words, the bias decreases as we see the feature set to 1 more often.

9.4 Probabilistic discriminative models

Recall that in generative modelling, the posterior class probabilities are

$$P(C_1|X) = \sigma(w^T x + w_0)$$

for a two-class problem and

$$P(C|X) = \text{softmax}(w^T x + w_0)$$

for k-class modelling, where w and w_0 are functions of the mean, variance, and prior probabilities.

In the discriminative approach, we assume (w, w_0) are a vector of parameters and learn them directly by using maximum likelihood. This has two advantages:

1. Fewer parameters than the generative approach.

2. Improved predictive performance when the class-conditional density assumptions give a poor approximation of the true distributions.

9.5 Logistic regression model

This model, despite being called "regression", is used for classification. For a two-class case,

$$\begin{aligned} P(C_1|X) &= y(X) \\ &= \sigma(w^T x) \\ P(C_2|X) &= 1 - P(C_1|X) \end{aligned}$$

where σ is the logistic sigmoid function.

Compared to GDA-style models, the logistic regression model has only $M+1$ parameters. If there are more features, then there is a clear advantage in working with the logistic regression model more directly.

Note that the derivative of $\sigma(a)$ is given by

$$\frac{d\sigma}{da} = \sigma(1-\sigma).$$

9.5.1 Maximum likelihood for logistic regression

Consider a dataset $\{x^{(n)}, t^{(n)}\}_{n=1}^N$ where $t^{(n)} \in \{0,1\}$. Then the likelihood of the parameters (probability of a particular assignment t of classes to each data point, given the parameters w) is:

$$P(t|w) = \prod_{n=1}^{N} (y^{(n)})^{t^{(n)}} \{1-y^{(n)}\}^{1-t^{(n)}}$$

where $t = (t^{(1)}...t^{(N)})^T$ (the class assignment) and $y^{(n)} = P(C_1|X^{(n)})$ (the posterior class probabilities).

The error function here is the negative logarithm of the likelihood.

$$\begin{aligned} E(w) &= -\ln P(t|w) \\ &= -\sum_{n=1}^{N} \left\{ t^{(n)} \ln y^{(n)} + (1-t^{(n)}) \ln(1-y^{(n)}) \right\} \end{aligned}$$

where $y^{(n)} = \sigma(a^{(n)})$, where $a^{(n)} = w^T x^{(n)}$. We differentiate and take the gradient:

$$\begin{aligned}
\frac{\partial E}{\partial y^{(n)}} &= \frac{-t^{(n)}}{y^{(n)}} + \frac{1 - t^{(n)}}{1 - y^{(n)}} \\
&= \frac{(1 - t^{(n)})y^{(n)} - t^{(n)}(1 - y^{(n)})}{y^{(n)}(1 - y^{(n)})} \\
&= \frac{y^{(n)} - y^{(n)}t^{(n)} - t^{(n)} + y^{(n)}t^{(n)}}{y^{(n)}(1 - y^{(n)})} \\
&= \frac{y^{(n)} - t^{(n)}}{y^{(n)}(1 - y^{(n)})}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial y^{(n)}}{\partial a^{(n)}} &= \frac{\partial \sigma(a^{(n)})}{\partial a^{(n)}} \\
&= \sigma(a^{(n)})(1 - \sigma(a^{(n)})) \\
&= y^{(n)}(1 - y^{(n)})
\end{aligned}$$

The gradient of $a^{(n)}$ is $\nabla a_n = X^{(n)}$. The gradient of E is:

$$\begin{aligned}
\nabla E &= \sum_{n=1}^{N} \frac{\partial E}{\partial y^{(n)}} \frac{\partial y^{(n)}}{\partial a^{(n)}} \nabla a^{(n)} \\
&= \sum_{n=1}^{N} \frac{y^{(n)} - t^{(n)}}{y^{(n)}(1 - y^{(n)})} y^{(n)}(1 - y^{(n)}) X^{(n)} \\
&= \sum_{n=1}^{N} (y^{(n)} - t^{(n)}) X^{(n)}
\end{aligned}$$

The contribution of the gradient from data point n is given by the error $(y^{(n)} - t^{(n)})$ between the target value and the prediction of the model, times the input vector $X^{(n)}$.

This takes precisely the same form as the gradient of the sum of squares error function for the linear regression model.

9.6 Maximum likelihood and least squares

Consider linear regression. Assume that the target variable t is given by a deterministic function $y(x; w)$ with additive Gaussian noise, so that

$$t = y(x; w) + \epsilon$$

where ϵ is a zero-mean Gaussian random variable with precision (inverse variance) β. Thus the likelihood is:

$$P(t|X, w, \beta) = \mathcal{N}(t|y(x; w), \beta^{-1}).$$

Now consider a dataset $\{x^{(n)}, t^{(n)}\}_{n=1}^{N}$. The likelihood considering the entire dataset is:

$$P(t|X, w, \beta) = \prod_{n=1}^{N} \mathcal{N}(t^{(n)}|w^T X^{(n)}, \beta^{-1})$$

and the log-likelihood:

$$\begin{aligned} \ln P(t|X, w, \beta) &= \sum_{n=1}^{N} \ln \mathcal{N}(t^{(n)}|w^T X^{(n)}, \beta^{-1}) \\ &= \frac{N}{2} \ln \beta - \frac{N}{2} \ln(2\pi) - \beta E_D(w) \end{aligned}$$

where

$$E_D(w) = \frac{1}{2} \sum_{n=1}^{N} \{t^{(n)} - w^T x^{(n)}\}^2.$$

Thus, maximizing the log-likelihood with respect to w is equivalent to minimizing $E_D(w)$ w.r.t. w.

$$\frac{\partial \ln P(t|X, w, \beta)}{\partial w} = \sum_{n=1}^{N} \{t^{(n)} - w^T x^{(n)}\} X^{(n)^T}$$

Note 1: For a squarred loss function, the optimal prediction is given by the conditional mean of the target variable.

$$E[t|X] = \int t \, P(t|X) \, dt = y(x; w)$$

The Gaussian noise assumption implies that the conditional distribution of t given x is unimodal, which may be inappropriate for some applications.

Note 2: The failure of least squares for classification (lecture 7) can now be explained. Binary target vectors clearly have a distribution that is far from Gaussian and hence lead to poor performance.

10 Newton-Raphson method, Perceptron

10.1 Iterative reweighted least squares

Recall least squares for linear regression. We obtained the closed-form solution $w = (x^T x)^{-1} x^T y$. We were able to get such a solution because the error function was a quadratic function of the parameters.

In logistic regression, the error function is not quadratic (and not linear) due to the sigmoid function, and therefore there is no closed form solution. However, the error function for logistic regression is convex and there still exists a unique minimum ("bowl" shape). We can therefore perform gradient descent.

10.1.1 Newton-Raphson method

The Newton-Raphson (NR) method is a more efficient method than gradient descent. It is also an iterative scheme, and uses a local quadratic approximation of the error function. Specifically, it uses a second-order Taylor series expansion

$$E(w) \approx E(w_0) + (w - w_0)^T \nabla_w E(w) + \frac{1}{2}(w - w_0)^T H (w - w_0)$$

where H is the Hessian of E with respect to w evaluated at w_0.

We differentiate this function and set it to zero:

$$\begin{aligned} 0 + \nabla_w E(w) + (w - w_0)^T H &= 0 \\ (w - w_0)^T H &= -\nabla_w E(w) \\ (w - w_0)^T &= -H^{-1} \nabla_w E(w) \\ w &= w_0 - H^{-1} \nabla_w E(w) \end{aligned}$$

$$w^{(new)} = w^{(old)} - H^{-1} \nabla_w E(w)$$

This is the Newton-Raphson update equation for the parameters w. Compare with the gradient descent update equation: $w^{(new)} = w^{(old)} - \alpha \nabla_w E(w)$ with the step size α.

Note: If the error function is a quadratic function, then the Taylor series expansion is exact and the NR method will find the solution in one step.

10.1.2 Example: linear regression

$$
\begin{aligned}
E_D(w) &= \frac{1}{2}\sum_{n=1}^{N}\left\{w^T\phi(x^{(n)}) - t^{(n)}\right\}^2 \\
\nabla E(w) &= \sum_{n=1}^{N}(w^T\phi^{(n)} - t^{(n)})\phi^{(n)} \\
&= \phi^T\phi w - \phi^T t \\
H = \nabla\nabla E(w) &= \phi^T\phi
\end{aligned}
$$

We get the **Newton-Raphson update rule** (for linear regression):

$$
\begin{aligned}
w^{(new)} &= w^{(old)} - (\phi^T\phi)^{-1}(\phi^T\phi w^{(old)} - \phi^T t) \\
&= w^{(old)} - w^{(old)} + (\phi^T\phi)^{-1}\phi^T t \\
&= (\phi^T\phi)^{-1}\phi^T t
\end{aligned}
$$

which is the standard least squares solution.

10.1.3 Geometric view

See the notes for a geometric interpretation of gradient descent vs. Newton-Raphson. We can imagine NR to be finding a quadratic approximation of the true function at some point on the function; find the minimum of the approximation; and project the minimum on the true function. Then from that point the process repeats, until the approximation stops improving.

10.1.4 Newton-Raphson for logistic regression

$$
\begin{aligned}
E(w) &= -\ln p(t|w) \\
&= -\sum_{n=1}^{N}\left\{t^{(n)}\ln y^{(n)} + (1-t^{(n)})\ln(1-y^{(n)})\right\} \\
\nabla E(w) &= \sum_{n=1}^{N}(y^{(n)} - t^{(n)})X^{(n)} \\
&= X^T(y-t) \\
H &= \nabla\nabla E(w) \\
&= \sum_{n=1}^{N} y^{(n)}(1-y^{(n)})X^{(n)}X^{(n)^T} \\
&= X^T R X
\end{aligned}
$$

where R is an $N \times N$ diagonal matrix with $R_{nn} = y^{(n)}(1 - y^{(n)})$.

Note 1: the Hessian is no longer constant; it depends on w through the weighting matrix R.

Note 2: By property of the logistic sigmoid function, $0 < y^{(n)} < 1$. Therefore, $u^T H u > 0$ for an arbitrary vector u, and so H is positive definite. Thus the error function is a convex function of w and has a unique minimum.

The **Newton-Raphson update rule** (for logistic regression) is therefore:

$$\begin{aligned} w^{(new)} &= w^{old} - (X^T R X)^{-1} X^T (y - t) \\ &= (X^T R X)^{-1} \{(X^T R X) w^{old} - X^T (y - t)\} \\ &= (X^T R X)^{-1} X^T R Z \end{aligned}$$

where Z is an N-dimensional vector with elements $Z = X w^{(old)} - R^{-1}(y - t)$.

The NR update rule looks like a weighted least-squares equation. However, the weight matrix R is not constant: it depends on w. So we must apply the update equation iteratively, each time using the new w to compute R. Thus **iterative reweigted least squares** (IRLS).

Note: Maximum likelihood can exhibit severe overfitting for datasets that are linearly separable. Why? Because the maximum likelihood solution occurs when the hyperplane corresponding to $\sigma = 0.5$, equivalent to $w^T x = 0$, separates two classes and the magnitude of w goes to infinity. This problem will happen even if we have more data points in the number of parameters of the model.

How do we solve this? Two solutions: (1) Add regularization, or (2) Use better estimators (more on this in a few lectures).

10.2 Pros and cons of generative, discriminative, and discriminant-based models for classification

10.2.1 Generative models

- **Con**: Most demanding models
- **Con**: If x is high-dimensional, we need a large training set in order to be able to determine the class-conditional densities to reasonable accuracy.
- **Pro**: We can compute $P(x)$ as $\sum_k P(x|C_k)P(C_k)$. This can be useful for detecting new data points that have low probability under the model, for which prediction may be of low accuracy (outlier detection).

10.2.2 Generative vs. discriminative

If we only wish to make classification decisions, we can just use discriminative models. Class-conditional densities may have a lot of structure that has little effect on the posterior proabilities.

10.2.3 Discriminant-based vs. discriminative

The discriminant-based approach is even simpler, as it combines inference and decision stages into a single learning problem. However, we no longer have access to the class-conditional densities $P(C_k|x)$.

10.2.4 Advantages of $P(C_k|x)$ (discriminative models)

- **Minimizing risk**: If the loss matrix is subject to change from time to time, then knowing $P(C_k|x)$ allows us to trivially revise the minimum risk decision criterion by using decision theory. With a discriminant-based approach, we need to retrain the model.

- **Compensating for class priors**. Suppose we're doing cancer prediction. The dataset is heavily skewed with very few (1 in 1000) patients having cancer. Classifying every patient as no-cancer has a 99.9% accuracy. The solution is to (1) sample the rare class more and balance the class distribution in the dataset; (2) learn the posterior probabilities $P(C_k|x)$; (3) Calculate the new posterior probabilities:

  $$\frac{\text{old posterior}}{\text{class fraction in sampled data}} \times \text{class fraction in original data.}$$

  This is not possible in discriminant-based models.

- **Combining models**: Consider

  $$\begin{aligned} P(C_k|x_1, x_2) &\propto P(x_1, x_2|C_k)P(C_k) \\ &\propto P(x_1|C_k)P(x_2|C_k)P(C_k) \\ &\propto \frac{P(x_1|C_k)P(x_2|C_k)}{P(C_k)} \end{aligned}$$

  We combine $P(C_k|x_1)$ and $P(C_k|x_2)$ to predict $P(C_k|x_1, x_2)$.

- **Reject option**: We can set a threshold, e.g. $\theta = 0.5$, below which classification cannot be done. For instance, if there are three classes and the posterior probabilities of a given data point are 0.33, 0.33,

and 0.34, it may be better to not pick a class rather than picking the one with the highest probability. We cannot do this without access to $P(C_k|x)$.

10.3 Summary of the course so far

Machine learning can be divided into supervised and unsupervised learning. Unsupervised learning comprises **dimensionality reduction**, including **PCA**.

Supervised learning comprises regression and classification. In the list below, the first L or N indicates whether the model is linear or not, and the second L or N indicates whether the decision boundary is linear (GLM) or not.

- Regression
  - Linear regression: LL
  - Non-linear regression using basis functions: LN
- Classification
  - Discriminant-based
    - Indicator regression: LL
    - LDA (Fisher): LL
  - Discriminative
    - Logistic regression: NL
    - k-nearest neighbors: NN
  - Generative
    - GDA: NL
    - QDA: NN
    - Gaussian Naïve Bayes: NN
    - DLDA: NL
    - Discrete Naïve Bayes: NL

10.4 Separating hyperplane methods

This is a class of discriminant-based models. These models construct linear decision boundaries that explicitly try to separate the data into different classes as well as possible. One important algorithm for this is the perceptron.

10.5 Perceptron algorithm

This algorithm for separating hyperplanes was one of the first machine learning algorithms, published in 1958 by Rosenblatt. The discriminant function is

$$y(x) = f(w^T x)$$

where f is a step function of the form

$$f(a) = \begin{cases} +1 \text{ if } a \geq 0 \\ -1 \text{ if } a < 0 \end{cases} .$$

A natural choice of error function would be the total number of misclassified patterns. This function would be a piecewise constant function of w, with discontinuities wherever a change in w causes the decision boundary to move across one of the data points. However, methods based on changing w using the gradient of the error function cannot be applied, because the gradient is zero almost everywhere.

10.5.1 Perceptron error criterion

We want w such that

1. If $x^{(n)}$ is in class C_1, then $w^T x^{(n)} > 0$;
2. If $x^{(n)}$ is in class C_2, then $w^T x^{(n)} < 0$.

Since $t \in \{-1, +1\}$, we can combine (1) and (2) as

$$w^T x^{(n)} t^{(n)} > 0 \qquad \text{for any } x^{(n)}.$$

Thus, if a data point is correctly classified, we get zero error; if it is wrongly classified, then we want to minimize $-w^T x^{(n)} t^{(n)}$. The perceptron criterion is therefore:

$$E_p(w) = -\sum_{n \in \mathcal{M}} w^T x^{(n)} t^{(n)}$$

where $\mathcal{M}$ denotes the set of misclassified patterns.

Thus the contribution to the error associated with a particular misclassified pattern is a linear function of w, in regions of w space where the pattern is misclassified; and zero, in regions where it is correctly classified.

This function is not continuous, but it is piecewise linear. Therefore we can do stochastic gradient descent. Take any $(x^{(n)} t^{(n)})$:

$$\begin{aligned} w^{(new)} &= w^{(old)} - \eta \nabla E_p(w) \\ &= w^{(old)} + \eta x^{(n)} t^{(n)} \end{aligned}$$

where η is the step size. But note: $y(x)$ is unchanged if we multiply w by a constant. So we can set $\eta = 1$ without loss of generality. In other words, there is no need for a step size. The perceptron update rule is therefore:

$$w^{(new)} = w^{(old)} + x^{(n)}t^{(n)}$$

10.5.2 Algorithm pseudocode

- Initialize w randomly
- Repeat until all data points are correctly classified:
  - For each data point $x^{(n)}t^{(n)}$:
    - If $y(x^{(n)}) == t^{(n)}$, then do nothing
    - Else, $w = w + x^{(n)}t^{(n)}$

Note: In every step, the error w.r.t. that particular data point is reduced. But this might affect the classification of other points. Thus this update is not guaranteed to reduce the total error at every step.

10.5.3 Perceptron convergence theorem

If the dataset is linearly separable, then the perceptron learning algorithm is guaranteed to find an exact solution in a finite number of steps.

10.5.4 Issues

The 'finite' number of steps may be very large. The smaller the gap between the classes, the longer it takes to find it.

When the data is linearly separable, there are many solutions, and the perceptron algorithm will find one of them. Which one it is depends on the starting values. It may be that the solution is not optimal when we check against the test set.

When the data is not separable, the algorithm will not converge, and cycles will develop. The cycles can be long and hard to detect.

11 Separating hyperplanes, SVM

11.1 Max-margin classifier

This is an optimal separating hyperplane algorithm. It addresses the issues with the perceptron algorithm.

In linearly separable cases, the perceptron solution depends on (1) initial values of w and b; (2) the order in which the data points are presented. In other words, the decision boundary (hyperplane separation) may not be optimal; it falls in the gap between the two classes, but perhaps not in a way that will allow correct classification of a test set.

A margin is defined as the perpendicular distance between the decision boundary and the closest data point. The max-margin approach is to find a decision boundary that maximizes the margin.

Recall from lecture 6 that the perpendicular distance of a point x from a hyperplane defined as $y(x) = 0$ where $y(x) = w^T\phi(x) + b$ is given by

$$\frac{|y(x)|}{|w|}.$$

We're interested only in solutions where all data points are correctly classified, i.e. $t^{(n)}y(x^{(n)}) > 0$ for all n. Then the distance of a point $x^{(n)}$ to the decision surface is given by

$$\frac{t^{(n)}y(x^{(n)})}{||w||} = \frac{t^{(n)}(w^T\phi(x^{(n)}) + b)}{||w||}$$

The margin is the perpendicular distance to the closest point $x^{(n)}$ from the dataset. So we get

$$w, b = \underset{w,b}{\text{argmax}} \left(\min_n t^{(n)}(w^T\phi(x^{(n)}) + b)... \right)$$

The solution to this optimization problem is very complex.

11.2 Scaling

Scaling doesn't change the distance. See notes.

11.3 Canonical representation of decision boundaries

We can use the freedom allowed by scaling to set $t^{(n)}(w^T\phi(x^{(n)}) + b) = 1$ for the point closest to the surface. In that case, all data points $n = 1, ..., N$ will

satisfy the constraint:

$$t^{(n)}(w^T\phi(x^{(n)}) + b) \geq 1.$$

This is known as the canonical representation of the decision hyperplane (or boundary).

The constraint is said to be **active** when the equality holds for a data point. When it doesn't hold, the constraint is **inactive**.

In general, there will always be one active constraint, because there will always be a closest point. Once the margin is maximized, there will be at least **2** constraints: one on both sides of the margin.

We can therefore modify the complex optimization problem above into a much simpler, constrained optimization problem.

$$\underset{w,b}{\text{argmax}} \left(\frac{1}{||w||} \right)$$

subject to (for $n = 1, ..., N$):

$$t^{(n)}(w^T\phi(x^{(n)}) + b) \geq 1.$$

This is equivalent to

$$\underset{w,b}{\text{argmin}} \frac{1}{2}||w||^2$$

subject to (for $n = 1, ..., N$):

$$t^{(n)}(w^T\phi(x^{(n)}) + b) \geq 1.$$

Note 1: this is a quadratic problem.

Note 2: bias

We will call this the primal problem, in opposition to the dual problem (below).

11.3.1 Lagrangian formulation of the problem

$$L(w, b, a) = \frac{1}{2}||w||^2 - \sum_{n=1}^{N} a_n \left(t^{(n)}(w^T\phi(x^{(n)}) + b) - 1 \right)$$

where $a = (a_1, ..., a_N)^T \geq 0$ is the Lagrange multiplier. Note that we have a minus sign before the Lagrange term because we want to minimize with respect to w, b and mazimize with respect to a.

Look in the notes for the derivation. We end up with

$$\max \tilde{L}(a) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m k(x^{(n)}, x^{(m)})$$

with respect to a subject to constraints

$$a_n \geq 0, \ n = 1, ..., N$$

$$\sum_{n=1}^{N} a_n t^{(n)} = 0$$

where $k(x, x') = \phi(x)^T \phi(x')$, the kernel function.

Note 1: this is also a quadratic programming problem. It is the dual problem of the primal problem above.

Note 2: quadratic problems can be solved by using standard solvers available in the optimization literature. The solution of a quadratic problem with M variables has $O(M^3)$ computational complexity.

The primal problem is over parameters (there are M parameters). The dual problem is over Lagrange multipliers (there are N variables, where N is the number of data points). Therefore, we can choose which problem to solve depending on the number of parameters and data points. If $N >> M$, solve the primal problem; if $M >> N$, solve the dual problem.

11.3.2 Finding w, b after solving the dual problem

Solving the dual problem gives us the Lagrange multipliers a_n. We want to find w and b.

We know that

$$w = \sum_{n=1}^{N} a_n t_n \phi(x^{(n)})$$

and

$$\begin{aligned} y(x) &= w^T \phi(x) + b \\ &= \sum_{n=1}^{N} a_n t^{(n)} (x, x^{(n)}) + b \end{aligned}$$

The dual problem must satisfy the KKT conditions:

$$a_n \geq 0$$

$$t^{(n)}y(x^{(n)}) - 1$$

[1]

$$a_n(t^{(n)}y(x^{(n)}) - 1) = 0$$

For every data point, either $a_n = 0$ or $t^{(n)}y(x^{(n)}) = 1$.

Any point for which $a_n = 0$ will not appear in the summation above. Therefore, it plays no role in making predictions for new points. In other words, it is not in the margin. This allows us to discard most of the data points after the model is trained.

The remaining data (for which $t^{(n)}y(x^{(n)}) = 1$) are called the **support vectors**. These data points lie on the maximum margin hyperplane.

To get b, we note that for every support vector $x^{(n)}$,

$$t_n() = 1$$

where S is...

11.3.3 Error function

The max-margin classifier is minimizing the following error function (typo in the notes: sum to N, not infinity):

$$\sum_{n=1}^{N} = E_\infty(t^{(n)}y(x^{(n)}) - 1) + \lambda||w||^2$$

where $E_\infty(z)$ is 0 if $z \geq 0$; ∞ otherwise.

The error function ensures that the constraint are satisfied. Note that as long as $\lambda > 0$, its precise value plays no role.

11.4 Overlapping class distributions

The max-margin classifier assumes the data is linearly separable. It gives infinite error of the data point was misclassified.

What to do if the data is not linearly separable? We modify the approach so that the data points are allowed to be on the 'wrong side' of the margin boundary, but with a penalty that increases (linearly) with the distance from the boundary.

We create the slack variable $\epsilon_n \geq 0$ for $n = 1, ...N$. If data point n is on the margin boundary or on the correct side of the margin, then $\epsilon_n = 0$ and it is correctly classified. For other data points,

$$\epsilon_n = |t^{(n)} - y(x^{(n)})|.$$

[1] (typo in the notes: missing the -1)

If $0 < \epsilon_n \leq 1$, then the n lies inside the margin, but on the correct side of the boundary; we consider ...

Doing this relaxes the hard margin constraint to give a soft margin and allow some training points to be misclassified. Note that because the penalty increases linearly, this is sensitive to outliers!

The goal has become to maximize the margin while softly penalizing points that lie on the wrong side of the margin boundary:

$$\min C \sum_{n=1}^{N} \epsilon_n + \frac{1}{2}||w||^2$$

where $C > 0$, the 'cost parameter' (it is a hyperparameter), controls the tradeoff between the slack variable penalty and the margin.

Note that for any misclassified...

support vector machine (SVM)

12 SVM, Non-parametric Methods, Decision Trees

12.1 Wrapping up Support Vector Machines (SVM)

We have a margin separating the green class from the purple class. Any point n that falls on the correct side of the margin (e.g. a green point on the green side) has $y^{(n)}t^{(n)} \geq 1$ and $\epsilon_n = 0$. The remaining points have $\epsilon_n = 1 - y^{(n)}t^{(n)}$.

We can write the SVM objective function as follows:

$$\sum_{n=1}^{N} = E_{SV}(y^{(n)}t^{(n)}) + \lambda||w||^2$$

where $\lambda = (2C)^{-1}$. We also have

$$E_{SV}(y^{(n)}t^{(n)}) = [1 - y^{(n)}t^{(n)}]_+$$

(where $[\ ]_+$ denotes the positive part)

$$= \max(0, 1 - y^{(n)}t^{(n)}).$$

This is known as **hinge loss**.

We can draw the hinge loss together with other loss functions—see the course notes. The hinge loss function decreases linearly until it gets to $z = 1$ (z is a dummy variable standing for whatever we try to minimize), after which it stays at $E(z) = 0$.

The real error function we are interested in is $0/1$ loss, i.e. misclassification. This loss function is a discrete function where $E(z) = 1$ for $z < 0$, and $E(z) = 0$ for $z > 0$. But this discrete function is difficult to optimize, so we use another error function to approximate error. Squarred error would not be a good one, because the error increases for large values of z. Hinge loss and logistic regression error are possibilities because they are monotonic.

See notes for logistic regression error.

12.2 Parametric vs. non-parametric methods

Parametric methods have a finite number of parameters, and this number of independent of the dataset's complexity. Examples include logistic regression, GDA, primal SVN. These methods are simpler, faster, and require less data. However, they are constrianed, have limited complexity, and may give a poor fit.

Non-parametric methods have a potentially infinite number of parameters, and this number depends on the dataset's complexity: the model will

add new parameters if the data is too complex. The dual view of SVM (where the number of support vectors depends on the dataset complexity), and k-nearest neighbors (where the entire dataset constitutes the parameters), are examples of non-parametric methods. These methods are more flexible and powerful (they require less or weaker assumptions) and generally have better performance. However, they require more data, are slower, and are more prone to overfitting.

12.3 Power of basis functions

Linear models and generalized linear models (GLMs) have linear decision boundaries. But we can use non-linear basis functions to get non-linear decision boundaries.

See course notes.

12.4 Decision trees

We move to a new category of classification methods: decision trees. For this lecture, "attribute" means "feature" and "record" means "example".

Suppose we have training data with several fields: marital status, taxable income, whether the person refunded the bank or not. The goal is to determine if the person will cheat the bank. We can build a decision tree to illustrate this. Did the person refund the bank (yes/no)? If yes, the person is unlikely to cheat. If no, is the person married (yes/no)? And so on.

The idea is, given a training set, to learn the decision tree.

12.4.1 Hunt's algorithm

Let D_t be the set of training records that reach a node t. The general procedure of the algorithm is:

- If D_t contains a record that belongs to the same class y_t, then t is a leaf node labeled as y_t.
- If D_t is an empty set, then t is a leaf node labelled by the default class y_D.
- If D_t contains records that belong to more than one class, use an attribute test to split the data into smaller subsets (this is the most interesting case). Recursively apply this procedure to each subset.

How to split the data? We can use a greedy strategy: split the records based on an attribute test that optimizes a certain criterion. But this begs some questions:

1. How to specify the attribute test condition?
2. How to determine the best split?
3. When to stop splitting?

12.5 How to specify the attribute test condition?

This depends on the attribute types—nominal, ordinal, or continuous. It also depends on the numbers of ways to split—binary or multiway.

In a multiway split, we use as many partitions as there are distinct values of the attribute. For instance, suppose there are three possible values of "CarType": family, sports, luxury. A multiway split would split the dataset into three partitions. A binary split would group some of them to make only two partitions, e.g. {sports, luxury} vs. {family}. There is a need to find the optimal partitioning.

For continuous attributes, the solution is to discretize the data (either binary, e.g. "earns more than $80k?", or multiway, e.g. tax brackets).

12.6 How to determine the best split?

Suppose we have 20 records, 10 of which are in either of class 0 and 1. We could split according to whether the subjects own a car, but if this splits the classes roughly 50-50, we don't learn anything. We could split according to their student IDs, but then each partition would have only one record (overfitting). Some middle ground, where the partitions do match the classification to some extent, would be better.

The lesson is that nodes with homogeneous class distributions are preferred. In other words, we need to split in a way that minimizes "node impurity". There are several measures of node impurity. We will see three: the Gini index, entropy, and misclassification error.

12.6.1 Gini index

The Gini index of a given node t is

$$Gini(t) = 1 - \sum_j [P(j|t)]^2$$

where $P(j|t)$ is the relative frequency of clas j at node t.

When records are equally distributed among all classes, implying least interesting information, $Gini = 1 - \frac{1}{n_C}$ where n_C is the number of classes. When all records belong to the same class, implying the most interesting information, $Gini = 0$. For two classes, the range of Gini is from 0 to 0.5.

The course notes show examples of computing the Gini index for binary, categorical, and continuous attributes.

12.6.2 Entropy

$$Entropy(t) = -\sum_j P(j|t) \log p(j|t)$$

The disadvantage of entropy is that it tends to prefer splits that result in a large number of partitions, each being small but pure. This was the case for the student ID example above.

We can avoid this by calculating a gain ratio:

$$\text{gain ratio} = \frac{\text{gain}}{\text{split info}}$$

where the split info is $-\sum_j \frac{n_i}{n} \log \frac{n_i}{n}$. This adjusts information gain by the entropy of the partitioning (split info). Higher entropy parititoning (large number of small partitions) is penalized.

12.6.3 Classification error

$$Error(t) = 1 - \max P(i|t)$$

12.7 When to stop splitting?

As the number of nodes increases, decision trees gain higher accuracy. But eventually this is due to overfitting. This is due to error or noise in the data, and (even if the data is noise-free) coincidental regularities in the data when it is limited.

There are two ways to avoid overfitting. One is to stop growing the tree early enough. The other is to allow the overfitting, but then post-prune the tree. In practice, the post-prune approach is more successful. It is difficult to decide where to stop in the first approach. The next two subsections describe two ways of performing post-pruning.

12.7.1 Reduced error pruning

Consider each node of the tree (except the root node) as a candidate for pruning (typically go bottom-up). For each candidate, remove its subtree and make it a leaf node. Then look at the classification of the validation set, and prune the node if the classification is not worse in the validation set compared to the original training data.

If some nodes are due to accidental coincidences in the training data, they are likely to be pruned off since these coincidences are unlikely to occur also in the validation data.

12.7.2 Rule post-pruning

Convert the tree learned from training data into a set of rules. There is one rule for every path from the root to a leaf node. Prune each rule by removing any preconditions that result in improved accuracy. Then, sort the pruned rules by their estimated accuracy and consider them in sequence when classifying more data.

12.8 Advantages of decision trees

Decision trees are inexpensive to construct, and are extremely fast at classifying unknown records. They are easy to interpret when small. Also, they can handle all types of attributes easily.

13 Ensemble Learning: Bagging, Boosting

13.1 Decision trees as collections of classifiers

A decision tree can be seen as splitting the input space into different regions based on the attributes. Thus we can consider a decision tree with k leaves as k classifiers, each responsible for one of the different regions.

In the simplest model, each of these classifiers might be a constant value (1 if in region a, 0 if in region b, etc.). But we could also have different statistical models responsible for classification in each region. We can conceptualize this as a committee of experts, each with their own expertise and decision process, and with the committee as a whole being responsible for final classification.

The limitation to this is that the division of input space is based on hard splits in which only one model is responsible for making predictions for any given values of the hidden variables.

13.2 Ensemble models

We can use a probabilistic framework to combine models and soften the decision process. Let there be k models for a conditional distribution $P(t|x,k)$ where x is the input variable, t the target variable, and $k = 1, ..., K$ are the model indexes. Then

$$P(t|x) = \sum_{k=1}^{K} \pi_k(x)\ P(t|x,k)$$

where $\pi_k(x) = P(k|x)$ represents the input-dependent mixing coefficients.

This method also is known as **mixture of experts**. There are many other ways of combining models.

13.2.1 Average

The simplest approach to combining models is to average them. Each model makes a prediction, and we average them to yield the final prediction.

If we have a single dataset, we need to introduce variability between the different models within the committee.

13.2.2 Bootstrap datasets

Suppose the dataset has N data points. We can create a new dataset X_B by drawing N data points at random from X, with replacement, so that points in X can be duplicated in, or absent from, X_B. We repeat this process L times to generate that many datasets, each of size N.

13.3 Bootstrap aggregation (bagging)

Consider a regression problem. We construct M bootstrap datasets and train M copies $y_m(x)$ (where $m = 1, ..., M$) of a linear regression model. The committee's prediction then is

$$y_{com}(x) = \frac{1}{M}\sum_{m=1}^{M} y_m(x).$$

Suppose the true regression function we are trying to predict is given by $h(x)$. Then the prediction for one model is

$$y_m(x) = h(x) + \epsilon_m(x)$$

where $\epsilon_m(x)$ is the model's error.

From the average sum of squares error for each model,

$$E_X[\{y_m(x) - h(x)\}^2] = E_X[\epsilon_m(x)^2],$$

we can calculate the expected error made by the committee as a whole. The average error made by the models acting individually is

$$E_{AV} = \frac{1}{M}\sum_{m=1}^{M} E_X[\epsilon_m(x)^2]$$

and the expected error from the committee is

$$\begin{aligned}
E_{com} &= E_X\left[\left\{\frac{1}{M}\sum_{m=1}^{M} y_m(x) - h(x)\right\}^2\right] \\
&= E_X\left[\left\{\frac{1}{M}\sum_{m=1}^{M} \epsilon_m(x)\right\}^2\right] \\
&= \frac{1}{M^2} E_X\left[\left\{\sum_{m=1}^{M} \epsilon_m(x)\right\}^2\right] \\
&= \frac{1}{M^2}\sum_{m=1}^{M}\sum_{m=1}^{N} E_X\left[\epsilon_m(x)\epsilon_n(x)\right]
\end{aligned}$$

By assuming the errors have a mean of 0 and are uncorrelated, i.e.

$$E_X[\epsilon_m(x)] = 0,\ E_X[\epsilon_m(x), \epsilon_n(x)] = 0,\ m \neq n,$$

we get

$$E_{com} = \frac{1}{M^2} \sum_{m=1}^{M} E_X \left[\epsilon_m(x)^2 \right]$$
$$= \frac{1}{M} E_{AV}.$$

Thus, the average error of a model can be reduced by a factor of M simply by averaging M versions of the model.

Note 1: We use the same model in each case, but with different datasets. So we keep the bias of the model, but reduce variance. The error reduction happens because of variance reduction.

Note 2: $E_{com} \leq E_{AV}$.

Note 3: Often, errors due to individual models are not uncorrelated. In practice, the errors are typically highly correlated, and the error reduction overall is generally small. For instance, when we bag decision trees, the generated trees are highly correlated.

13.4 Random forests

Use k bootstrap replicates of the data to train k different trees. At each node, pick m variables at random ($m < M$ where M is the total number of features) and determine the best test (using normalized information gain). Recurse until the tree reaches maximum depth. We do not prune the trees.

Note 1: each tree has high variance, but the ensemble uses averaging to reduce variance.

Note 2: Random forests are very competitive in both classification and regression. They are used a lot in real-life contexts, and are typically the first thing that people try when attempting a new problem. However, they are still prone to overfitting.

13.5 Extremely randomized trees

Construct k decision trees. At each node, pick m attributes at random (without replacement) and pick a random test involving each attribute. Evaluate all tests (using a normalized information gain metric) and pick the best one for the node. Continue until a desired depth or a desired number of instances (n_{min}) at the leaf is reached.

Note 1: Very reliable method for both classification and regression

Note 2: The small the m, the more randomized the trees are. A small m is best, especially with large levels of noise.

Note 3: A small n_{min} means less bias and more variance, but variance is controlled by averaging over trees. Compared to single trees, we can pick a smaller n_{min} and minimize the bias.

13.6 Boosting

Boosting is a powerful technique for combining multiple 'base' classifiers into a committee whose performance can be significantly better than that of any of the base classifiers. Boosting can give good results even if the base classifiers have a performance only slightly better than random. Base classifiers are sometimes known as "weak learners".

The main difference between boosting and other ensemble methods, such as bagging, is that the base classifiers are trained **in sequence**.

Each base classifier is trained with a weighted form of dataset in which the weighting coefficients associated with each data point depends on the performance of the previous classfier. Points that are misclassified by one of the base classifiers are given greater weight when used to train the next classifier in the sequence.

Once all the classifiers have been trained, their predictions are combined through a weighted majority voting scheme.

For illustration purposes, we will consider a two-class problem with N data points, $\{x^{(n)}, t^{(n)}\}_{n=1}^{N}$ where $t^{(n)} \in \{-1, 1\}$. Each data point is given an associated weight parameter w_n which is initially set to $1/N$ for all data points.

13.6.1 Adaptive boosting (AdaBoost) algorithm

- Initialize the data weighting coefficients w_n by setting $w_n^{[1]} = 1/N$ for $n = 1, ..., N$.
- For $m = 1, ..., M$ (number of classifiers):
  - Fit a classifier $y_m(x)$ to the training data by minimizing the weighted error function:

$$J_m = \sum_{n=1}^{N} w_n^{[m]} \mathbb{I}(y_m(x^{(n)}) \neq t^{(n)})$$

  - Evaluate the quantities:

$$e_m = \frac{\sum_{n=1}^{N} w_n^{[m]} \mathbb{I}(y_m(x^{(n)}) \neq t^{(n)})}{\sum_{n=1}^{N} w_n^{[m]}}$$

and then use them to evaluate

$$\alpha_m = \ln \left\{ \frac{1 - e_m}{e_m} \right\}$$

– Update the data weighting coefficients

$$w_n^{[m+1]} = w_n^{[m]} \exp\{\alpha_m \mathbb{I}(y_m(x^{(n)}) \neq t^{(n)})\}$$

- Make predictions using the final model, which is given by

$$Y_M(X) = \text{sign}\left(\sum_{m=1} M \alpha_m y_m(x) \right)$$

in which e_m represents the weighted measures of the error rates of each of the base classifiers in the dataset, for iteration m; and α_m gives greater weight to more accurate classifiers when computing the overall output.

13.6.2 Example

Each base learner consists of a threshold on one of the input variables (also known as decision stumps). The notes include a figure where we see the decision boundary of the most recent learner, the combined decision boundary, and the weight assigned to each data point at each iteration.

13.7 Minimizing exponential error

Consider the error function defined by:

$$E = \sum_{n=1}^{N} \exp\left\{ -t^{(n)} f_m(x^{(n)}) \right\}$$

where $f_m(x^{(n)})$ is a classifier defined in terms of a linear combination of base classifiers $y_l(x)$ of the form

$$f_m(x) = \frac{1}{2} \sum_{l=1}^{m} \alpha_l y_l(x)$$

and where $t_n \in \{-1, 1\}$.

The goal is to minimize E with respect to both the weighting coefficient α_l and the parameters of the base classifiers $y_l(x)$.

Instead of doing a global error function minimization, however, we will suppose that the base classifiers $y_1(x), ..., y_{m-1}(x)$ are fixed, as are their coefficients $\alpha_1, ..., \alpha_{m-1}$. Thus we are minimizing only with respect to the last classifier, $y_m(x)$, and its coefficient α_m.

$$\begin{aligned} E &= \sum_{n=1}^{N} \exp\left\{-t^{(n)} f_{m-1}(x^{(n)}) - \frac{1}{2} t^{(n)} \alpha_m y_m(x^{(n)})\right\} \\ &= \sum_{n=1}^{N} w_n^{[m]} \exp\left\{-\frac{1}{2} t^{(n)} \alpha_m y_m(x^{(n)})\right\} \end{aligned}$$

where the coefficients $w_n^{[m]} = \exp\{-t^{(n)} f_{m-1}(x^{(n)})\}$ can be viewed as constants because we are only optimizing α_m and $y_m(x)$.

Let $\mathcal{T}_m$ be the set of data points that are correctly classified by $y_m(x)$, and $\mathcal{M}_m$ the set of points that are misclassified. ...

14 Stacking, Neural Networks

14.1 Exponential error function

In the last lecture, we saw the exponential error function:

$$\exp\{-ty(x)\}.$$

The expected error is thus (assuming a binary classification problem):

$$E_{x,t}[\exp\{-ty(x)\}] = \sum_t \int_x \exp\{-ty(x)\}P(t|x)P(x)dx$$

If we differentiate w.r.t. $y(x)$, we get

$$\frac{\partial}{\partial y(x)} E_{x,t}[\ ] = ...$$

Setting this equal to zero, we get

$$\exp\{y(x)\}P(t=-1|x) = \exp\{-y(x)\}P(t=1|x)$$

which when we rearrange yields

$$y(x) = \frac{1}{2}\ln\frac{P(t=1|x)}{P(t=-1|x)}.$$

We see that Ada Boost is trying to find the best approximation to the log odds ratio with the constraints of the optimization problem, i.e. the space of functions represented by the linear combination of base classifiers, and the sequential optimization strategy.

The exponential error gives a higher penalty for misclassification than other error functions like cross-entropy, hinge loss and $0/1$ loss.

14.2 Bagging vs. boosting

Bagging is faster, but has a small error reduction. It works well with "reasonable" classifiers. If some data are mislabeled, there may be some issues, but relatively few. Bagging is good at reducing variance.

By contrast, boosting is a slower algorithm, but it can lead to a higher error reduction. Boosting may have problems when a lot of data is mislabeled, because it will focus on these examples a lot and may overfit. Boosting reduces the bias of extremely weak learners that have high bias.

Which method is better depends on the constraints of any given problem. For instance, some learning problems may take a long time (e.g. a week to train a classifier) and might not be suitable to boosting if we don't have several weeks to spare.

14.3 Side-note: Cross-validation

Given a training dataset, we split the data into a training and a validation part to perform model selection. There is a chance that the split is not representative, for instance because most instances of a particular class end up by chance in the validation part.

One solution to this is **k-fold cross-validation**: divide the data into k disjoint folds; use $k-1$ folds for training and the remaining one for validation; then repeat the training with another set of $k-1$ folds to perform validation with all the folds. This will yield the average of all k folds' performance. However, this is more computationally costly.

In extreme cases with very few data points, we can use **leave-1-out cross-validation**, where each data point is in turn a 1-point validation set.

14.4 Stacked generalization

In this ensemble model approach, we "stack" a meta-model on top of multiple base models. Unlike in bagging and boosting, the base classifiers can be completely different. For instance, we use a Naïve Bayes classifier, an SVM, and a k-NN classifier as base models, and the outputs of these three models are given as input to the meta-model (which is itself a classifier such as perceptron, etc.).

One issue with this is that training and validation error can be different between models, and the meta-classifier may learn to give more weight to the model that has the least training error, even if it does not necessarily have less validation error.

To solve this, we must use k-fold cross-validation.

14.4.1 Algorithm

See the notes.

14.5 Neural networks

14.5.1 The infamous XOR problem

The XOR problem is to classify the logical function XOR based on the inputs (e.g. 0 XOR 1 = 1 and 1 XOR 1 = 0). In 1969, Minsky and Papert showed in their book *Perceptron* that the perceptron algorithm couldn't solve the XOR problem. This led to a loss of interest in AI and the first AI winter.

Now we know that the XOR problem is not linearly separable, and thus cannot be solved by algorithms such as the perceptron. However, we can add

non-linear basis functions to linear models like perceptron and hence obtain a non-linear decision boundary.

Basis functions project the data from the original space to a new space where it might be easy to learn to classify—they change the representation of the data. While it is easy to construct a basis function for the XOR problem, it is difficult to do for even simple real-life problems. Neural networks learn these basis functions (called "representation learning") along with learning to classify.

14.5.2 Representation learning

Along with the classifier parameters, we also learn the projection parameters by minimizing the error function.

Model also called Multi-Layered Perceptron (MLP)

Each unit is a neuron.

14.5.3 Example: prediction problem

We have an input $x \in \mathbb{R}^d$ and an output $t \in \mathbb{R}^k$. An MLP is defined as follows:

$$\begin{aligned} a(x) &= W^{(1)^T} x + b^{(1)} \\ h(x) &= g(a(x)) \\ y = f(x) &= o(W^{(2)^T} h(x) + b^{(2)}) \end{aligned}$$

where $a(x)$ is the hidden layer pre-activation, while $h(x)$ is the hidden layer after activation; $g(x)$ is the activation function, and o is the output activation function. The parameters of the MLP are $W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)}$.

Note 1: g has to be some non-linear activation function

Note 2: the activation function must be differentiable if we want to do gradient descent

14.5.4 Some standard activation functions g

Sigmoid activation function

tanh activation function

14.5.5 Some output activation functions o

- Identity function (if regression problem)
- Sigmoid function (if output needs to be between 0 and 1, i.e. a probability)

- Softmax function (if output needs to be a probability distribution over k classes)

14.5.6 Universal approximation theorem

"A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units." (Hornik, 1991)

This is nice in theory, but in practice, the number of hidden units required to solve a problem grows exponentially with the complexity of a problem (when we use gradient descent for learning).

14.5.7 More layers

Instead of widening the network (adding neurons to a single hidden layer), we can add more hidden layers.

An L-layer neural network has the following Layer preactivation Hidden layer activation Output layer activation

15 Neural Networks and Backpropagation

15.1 Learning in neural networks

Consider an MLP with n input and m output units. It can consist of any number of hidden units and can exhibit any desired fast-forward connection pattern.

$$h = g(W^{(1)^T} x + b(1))$$
$$y = o(W^{(2)^T} x + b(2))$$

where g and o are activation functions.

We have a training set $\{(x_i, p_i)\}_{i=1}^{P}$ and the error function

$$\epsilon = \frac{1}{2} \sum_{i=1}^{P} ||y_i - t_i||^2$$

where y_i is the output of the network for the ith input.

We can use gradient descent to learn the parameters (weight matrices and bias vectors) of the neural net. But note that the output y is a composite function of input x. Hence we can use the chain rule to compute the gradients.

Backpropagation is an efficient algorithm that implements the chain rule by using dynamic programming.

Notation note: P is the number of examples, n is the input dimension, and m is the output dimension. The parameters of the network are $w_1, ..., w_?$.

15.2 Backpropagation overview

Consider an extended network (see picture in the notes) that computes the error function automatically

15.3 Derivatives of network functions

The goal is to find a method for efficiently calculating the gradient of the one-dimensional network function according to the weights of the network.

15.3.1 B-diagram (backpropagation diagram)

Each node has two parts. One part contains the primitive function f associated with the node, and the ther part contains the derivative of f, f', for the same input.

We can separate the summation from the activation. The derivative of a summation of n arguments w.r.t. any one of them is 1.

The network is evaluated in two stages: feed-forward and backpropagation.

15.3.2 Feed-forward stage

Information comes from the left (derivative side). Assuming te input is x, each unit evaluates the primitive function $f(x)$ in its right side and $f'(x)$ in its left side, and stores them. Only the right part is passed to the units to the right. For the next unit, the input is now $f(x)$, and the unit will store e.g. $g(f(x))$ in its right side and $g'(f(x))$ in its left side.

15.3.3 Backpropagation stage

The whole network is run backwards, whereby the stored results are now used. There are three cases to consider.

- **Case 1**: function composition.
- **Case 2**: function addition. When there are two (or more) units pointing to the same summation unit to the right, we have function addition. The summation node will get derivatives of 1 in the feed-forward stage. In the backpropagation stage,
- **Case 3**: weighted edges.

Note: weighted edges are used in exactly the same way in both steps: they modulate the information transmitted in each direction by multiplying it by the edge's weight.

15.4 Backpropagation algorithm

Consider a network with a single real input x and network function F. The derivative $F'(x)$ is computed in two phases:

- **Feed-forward**: The input x is fed into the network. The primitive functions at the nodes and their derivatives are evaluated at each node. The derivatives are stored.
- **Backpropagation**: The constant 1 is fed into the output unit and the network is run backwards. Incoming information to a node is added, and the result is multiplied by the value stored in the left part of the unit. The result is transmitted to the left of the unit. The result collected at the input unit is the derivative of the network with respect to x.

15.4.1 Proof

Proposition: The backpropagation algorithm correctly computes the derivative of the network function F with respect to x.

We prove this by induction. The base case is: units in series, units in parallel, weighted edges. We proved this before with the three cases.

The induction hypothesis is: The algorithm works for any feed-forward network with n or fewer nodes.

Induction: if the algorithm works for an n-node network, then it will work for an $(n+1)$-node network.

15.5 Learning with backpropagation

Consider a weight w_{ij} (connecting the ith node to the jth node). This weight can be considered as an input channel into the subnetwork made of all paths starting at w_{ij} and ending in the single output unit of the network.

The information fed into the subnetwork in the feed-forward step was $o_i w_{ij}$, where o_i is the stored input of unit i. Let $x = o_i w_{ij}$ ...

Note: backpropagation is not a learning algorithm! The learning algorithm used in this method is still gradient descent.

15.6 The case of layered networks

Consider a network with n input units, k hidden units (in one hidden layer), and m output units.

... These formulas can be generalized to any number of layers.

15.6.1 Steps of the algorithm

- **Step 1**: Feed-forward computation.
- **Step 2**: Backpropagation to the output layer
- **Step 3**: Backpropagation to the hidden layer
- **Step 4**: Weight updates

15.7 More than one training examples

For weight

If we consider all training examples for a single update, this is full-batch gradient descent. When we use only one example, this is stochastic gradient descent.

15.8 Backpropagation in matrix form

Backpropagation can be performed efficiently by a computer (especially a GPU) when put into matrix form. The derivatives stored in the output units would be

$$D_2 = \begin{pmatrix} o_1^{(2)}(1-o_1^{(2)}) & 0 & \cdots & 0 \\ 0 & o_2^{(2)}(1-o_2^{(2)}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & o_m^{(2)}(1-o_m^{(2)}) \end{pmatrix}$$

and the derivatives stored in the hidden layer would be

$$D_1 = \begin{pmatrix} o_1^{(1)}(1-o_1^{(1)}) & 0 & \cdots & 0 \\ 0 & o_2^{(1)}(1-o_2^{(1)}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & o_m^{(1)}(1-o_m^{(1)}) \end{pmatrix}.$$

The derivatives of the quadratic deviation are

$$e = \begin{pmatrix} o_1^{(2)} - t_1 \\ o_2^{(2)} - t_2 \\ \vdots \\ o_m^{(2)} - t_m \end{pmatrix}$$

...

16 Training deep neural networks

16.1 Deep neural networks

A deep neural network (DNN) is any neural network with more than one hidden layer. They can be seen as a sequence of nonlinear projections of the data before the prediction layer.

In practice, there is not much gain in going deep. In fact, the performance is often worse when we go deep. Why? Because of the vanishing gradient problem.

When we use the sigmoid or tanh activation functions for hidden units, the range of values they can take is (0, 0) and (-1, 1) respectively. When we differentiate, the range of gradients with respect to their input is (0, 0.25) and (0, 1) respectively.

If we have two hidden layers h_1 and h_2, and thus a network with input layer (x), hidden layer h_1, hidden layer h_2, output layer y, and error layer E, and the weight matrices between the layers $W^{(1)}$, $W^{(2)}$, and $W^{(3)}$ (not between y and E).

$$\frac{\partial E}{\partial W^{(1)}} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W^{(1)}}$$

Let's assume a sigmoid activation function, and focus on

$$\begin{aligned}\frac{\partial h_2}{\partial h_1} &= \frac{\partial}{\partial h_1}(\sigma(W^{(2)}h_2 + b)) \\ &= \sigma' W^{(2)}\end{aligned}$$

where the range of σ' is (0, 0.25) and $W^{(2)}$ is typically initialized using a Gaussian with mean 0 and SD 1 (range is (-1, 1)).

Each term in the is the product of a weight matrix (with range (-1, 1)) and σ' (with range 0, 0.25) or $\tanh'$ (with range 0, 1). Gradients propagated to lower layers therefore shrink a lot. This is known as the vanishing gradient problem and is an active research problem.

16.2 Solution 1: Greedy layerwise pretraining

An autoencoder is a neural network whose outputs are the same as the inputs. We have an input layer x, a hidden layer h, and an output layer y such that $h = f(x)$ (for instance, $\sigma(wx + b)$) and $y = g(h)$ (for instance, $\sigma(vh + c)$).

The model is trained to minimize the reconstruction error. We use L2 loss if the inputs x are real-valued:

$$\frac{1}{2}(x - g(f(x)))^2$$

or cross-entropy loss if the x are binary.

An autoencoder is an encoder followed by a decoder. $h = f(x)$ is the encoding operation and $y = g(h)$ is the decoding operation. However, a hidden layer that is equal in size or larger than the input layer would lead to the model to just learn to copy the input into the output. The model is thus more useful when the hidden layer is smaller than the input layer.

Note: similarity to PCA. But we use only nonlinear autoencoders

Back to the greedy layerwise training solution. Given a prediction task $x \to y$ and a training set, we will use the following layerwise training procedure:

1. Given x, train an autoencoder: an encoder from x to h_1, with weight matrix $W^{(1)}$ (randomly initialized), and a decoder from h_1 to x with weight matrix $W^{(1)'}$.

2. Throw away the decoder. Consider h_1 as an input layer, and train another autoencoder (h_1 to h_2 to h_1, with weights $W^{(2)}$ and $W^{(2)'}$).

3. Repeat this procedure k times to get k hidden layers. (k is a hyperparameter.)

4. Add a classification layer y on top of the hidden layers, with weights $W^{(k+1)}$, and train the network for the actual prediction task by using backpropagation. In this stage we also update all the previous weights $W^{(i)}$.

Steps 1 to 3 are called greedy layerwise pretraining. Step 4 is called supervised finetuning.

This solution does not actually solve the vanishing gradient problem, but is much better than plain backpropagation.

16.3 Solution 2: Rectified linear units (ReLU)

ReLU activation is defined as follows:

$$f(x) = \max(0, x).$$

The gradient of this function is 0 if $x < 0$ and 1 if $x < 0$

16.3.1 Solution: Leaky ReLU

$f(x) = \max(\epsilon x, x)$ where e.g. $\epsilon = 0.01$.

16.4 Solution 3: Batch normalization

The idea of batch normalization is to normalize the preactivation such that the hidden unit activation will not saturate.

Inputs: preactivation x over a mini-batch $\mathcal{B} = \{x_1, ..., x_m\}$. Parameters to be learned: r, β.

Outputs: $\{y_i = BN_{r,\beta}(x_i)\}$. The mini-batch mean: The mini-batch variance:

We normalize:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^T + \epsilon}}$$

and scale and shift: