Lecture Notes on

Machine Learning

Kevin Zhou

kzhou7@gmail.com

These notes follow Stanford's CS 229 machine learning course, as offered in Summer 2020. Other good resources for this material include:

- Hastie, Tibshirani, and Friedman, *The Elements of Statistical Learning.*
- Bishop, *Pattern Recognition and Machine Learning.*
- Wasserman, *All of Statistics.*
- Russell and Norvig, *Artificial Intelligence: A Modern Approach.*
- Mackay, *Information Theory, Inference, and Learning Algorithms.*
- Michael Nielsen's online book, *Neural Networks and Deep Learning*.
- Jared Kaplans's Contemporary Machine Learning for Physicists lecture notes.
- A High-Bias, Low-Variance Introduction to Machine Learning for Physicists.

The most recent version is here; please report any errors found to kzhou7@gmail.com.

Contents

1 Supervised Learning

1.1 Introduction

We begin with an overview of the subfields of machine learning (ML).

- According to Arthur Samuel, ML is the field of study that gives computers the ability to learn without being explicitly programmed. This gives ML systems the potential to outperform the programmers that made them. More formally, ML algorithms learn from experiences by using them to increase a performance measure on some task.

- Broadly speaking, ML can be broken into three categories: supervised learning, unsupervised learning, and reinforcement learning.

- Supervised learning problems are characterized by having a "training set" that has "correct" labels. Simple examples include regression, i.e. fitting a curve to points, and classification. Supervised learning has its roots in statistics.

- Generically, a supervised learning algorithm will minimize some cost function over a hypothesis class $\mathcal{H}$ evaluated on the training set. This will give a model that can be used to predict labels on a "testing set". Choosing the minimization procedure itself constitutes an entire field of study, "optimization". We'll mostly avoid the details of it here.

- Picking a good $\mathcal{H}$ is called the model selection problem. If it's too simple, it may underfitting, being not be powerful enough to capture the trust; if it's too complicated, it may overfit to the training set, and do badly on the testing set. This is called the bias-variance tradeoff.

- In unsupervised learning, the goal is instead to find structure in an unlabeled data set. A typical example is clustering, e.g. finding news articles about the same topic, or groups in a social network. Other examples include dimensionality reduction, (probability) density estimation, and outlier/anomaly detection. There are also semi-supervised learning algorithms, which do classification when only some of the examples are labeled, and weakly supervised algorithms, which can cope with noisy labels.

- In reinforcement learning (RL), we think in terms of an "agent" which moves between "states" in an "environment", seeking to maximize a reward signal by learning an appropriate "policy", without being explicitly told how. Most popular depictions of ML focus on RL, which has its roots in control theory from engineering. It also has roots in biology, with similar language used to describe animal learning. Examples include learning to play a board game, or maneuvering a robot past obstacles.

- Often, RL systems model and predict their environment, and may use supervised or unsupervised learning algorithms as subroutines. However, the overall problem is more general, and more realistic. Also, model-free RL systems exist, such as genetic algorithms or simulated annealing.

- One of the fundamental problems of RL is the explore-exploit tradeoff, i.e. whether the agent should keep on doing the same thing to get a known reward, or try new things for the potential of an even higher reward. We can describe this tradeoff formally by having the agent maximize a "value function", which accounts for both the reward itself, and the possible rewards it could get by further exploration.

- As an example, if an agent plays tic tac toe against a fixed opponent, it's easy to find a way to guarantee a tie. But if the opponent is imperfect, there could be a different sequence of moves that could force a win. If the agent explores sufficiently, it can find this sequence.

Next, we fix notational conventions.

- Given a function $f\colon \mathbb{R}^{m\times n} \to \mathbb{R}$, the gradient of f with respect to A is the $m \times n$ matrix of partial derivatives
$$(\nabla_A f(A))_{ij} = \frac{\partial f}{\partial A_{ij}}.$$
This includes the gradient with respect to a vector as a special case.

- As examples of vector gradients, we have
$$\nabla_x (b^T x) = b, \quad \nabla_x (x^T A x) = (A + A^T)x.$$
Note that in the second expression we can take A to be symmetric without loss of generality, since it doesn't change the left-hand side. We'll drop the subscript when it's clear what the gradient is with respect to, and we distinguish vectors and matrices by context.

- As basic examples of matrix gradients, we have
$$\nabla_A (\operatorname{tr} AB) = B^T, \quad \nabla_{A^T} f(A) = (\nabla_A f(A))^T.$$

- A slightly trickier example is
$$\nabla_A (\det A) = (\det A)(A^{-1})^T$$
which is best shown using the component expression for the determinant. As a special case, for positive definite A we can always define $\log\det A$, and
$$\nabla_A (\log\det A) = A^{-1}$$
where we used the fact that A is symmetric.

- Given a function $f\colon \mathbb{R}^n \to \mathbb{R}$, the Hessian matrix with respect to x is the symmetric $n \times n$ matrix of second derivatives, which we write as
$$(\nabla_x^2 f(x))_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}.$$
In other words, when we write ∇^2, we mean $\nabla \otimes \nabla$, not $\nabla \cdot \nabla$ as in conventional in physics. (We could also take the second derivative with respect to a matrix, giving a rank 4 tensor.) The function f is convex if the Hessian is positive semidefinite (PSD).

- As a simple example,
$$\nabla_x^2 (x^T A x) = A + A^T = 2A$$
where we assumed A was symmetric.

Example. For a symmetric matrix A, consider the optimization problem

$$\max x^T Ax \text{ such that } x^T x = 1.$$

Intuitively, x should be aligned with the eigenvector of A with largest eigenvalue. To show this formally, we form the Lagrangian

$$\mathcal{L}(x, \lambda) = x^T Ax - \lambda(x^T x - 1)$$

which then gives

$$\nabla_x L = 2Ax - 2\lambda x = 0$$

which is precisely the condition that x is an eigenvector of A.

Note. For a multivariate Gaussian random variable, $x \sim \mathcal{N}(\mu, \Sigma)$, we have

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

where μ is a d-dimensional vector and Σ is a $d \times d$ PSD matrix. Suppose we split these into blocks,

$$x = \begin{pmatrix} x_A \\ x_B \end{pmatrix}, \quad \mu = \begin{pmatrix} \mu_A \\ \mu_B \end{pmatrix}, \quad \Sigma = \begin{pmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{pmatrix}.$$

The block form of Σ^{-1} is

$$\begin{pmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{pmatrix} = \begin{pmatrix} (\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA})^{-1} & -(\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA})^{-1}\Sigma_{AB}\Sigma_{BB}^{-1} \\ -\Sigma_{BB}^{-1}\Sigma_{BA}(\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA})^{-1} & (\Sigma_{BB} - \Sigma_{BA}\Sigma_{AA}^{-1}\Sigma_{AB})^{-1} \end{pmatrix}.$$

The marginal and conditional distributions are also Gaussian,

$$x_A \sim \mathcal{N}(\mu_A, \Sigma_{AA}), \quad x_A | x_B \sim \mathcal{N}(\mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}(x_B - \mu_B), \Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA})$$

along with the analogous expressions with A and B swapped. The first result just follows from the definitions of Σ and Σ_{AA}, while the second can be shown by completing the square.

For the conditional distribution, the term added to the mean represents how knowledge of x_B shifts the distribution of x_A, which is why it is proportional to Σ_{AB}, while the term subtracted from the variance represents how x_B reduces the uncertainty in x_A. This is easier to see in the case where the blocks have one element each,

$$x = \begin{pmatrix} x_a \\ x_b \end{pmatrix}, \quad \mu = \begin{pmatrix} \mu_a \\ \mu_b \end{pmatrix}, \quad \Sigma = \begin{pmatrix} \sigma_a^2 & \rho\sigma_a\sigma_b \\ \rho\sigma_a\sigma_b & \sigma_b^2 \end{pmatrix}$$

where ρ is the correlation. Then the result above reduces to

$$x_a | x_b \sim \mathcal{N}(\mu_a + \rho\sigma_a(x_b - \mu_b)/\sigma_b, \sigma_a^2(1 - \rho^2)).$$

1.2 Linear Regression

The input and output sets are $\mathcal{X}$ and $\mathcal{Y}$. In regression, these will be continuous values; for classification, $\mathcal{Y}$ will be discrete. The input variables/features are $x_j^{(i)}$ where j indexes the feature, and the outputs/targets are $y^{(i)}$. A pair $(x^{(i)}, y^{(i)})$ is a training example, and m training examples form the training set, each with n features. The goal is to learn a hypothesis function $h \colon \mathcal{X} \to \mathcal{Y}$ so that $h(x)$ predicts the value of y on the testing set. Given this background, we now describe linear regression.

- In linear regression, we let h be linear,
$$h_\theta(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x$$
where we can have an intercept by including a feature $x_0 = 1$. The parameters θ_i are weights.

- To learn θ, we will have our algorithm minimize a cost function, also called the loss function or empirical risk. The choice
$$J(\theta) = \frac{1}{2} \sum_i |h_\theta(x^{(i)}) - y^{(i)}|^2$$
results in least squares regression, though other choices are possible.

- For least squares, we can find the optimal θ in closed form. A more general method is gradient descent, which works for general cost functions, and scales better for large data sets. We start with an initial guess for θ, and at each step perform the update
$$\theta := \theta - \alpha \nabla_\theta J(\theta)$$
where α is called the learning rate.

- Explicitly, for least squares we have
$$\nabla_\theta J(\theta) = \sum_i (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)}.$$
This is called the LMS update rule, or the Widrow–Hoff learning rule.

- This procedure is called batch gradient descent, since each step depends on all of the data points. However, when there are many data points, this can be slow. Another option is stochastic gradient descent, where we loop through the data points i, and at each step set
$$\theta := \theta + \alpha(y^{(i)} - h_\theta(x^{(i)})) x^{(i)}.$$
Typically, stochastic gradient descent can get close to the minimum faster, so it is preferred for large data sets. Another advantage is that it is less prone to getting stuck in local minima, since iterating over data points gives the trajectory some natural "jitter". (This isn't relevant for least squares, where the cost function is convex.)

Note. Getting gradient descent to work in practice requires some know-how. If α is too small, convergence is slow; if it's too large, $J(\theta)$ doesn't decrease monotonically. In general, α should be decreased as the learning continues, to hone in on the minimum; this is especially useful for stochastic gradient descent. Also, note that gradient descent and least squares are not "reparametrization invariant". If one of the features has much larger values than the others, it dominates the gradient and the cost, so convergence of the other θ_i is slow and the final result may not be useful. In general it's best to shift and scale all the features into a standard range, which e.g. can be done by replacing features with their z-scores.

Note. We can also minimize the least squares loss function analytically. Combine the feature vectors into a matrix X, and the outputs into a vector y. Then the loss function is

$$J(\theta) = \frac{1}{2}(X\theta - y)^T(X\theta - y)$$

and setting the derivative $\nabla_\theta J$ to zero gives the "normal equations",

$$X^T X\theta - X^T y = 0.$$

This condition has a geometric interpretation: when the length of the error vector $X\theta - y$ is minimized, it should be orthogonal to the column space of X, since that gives the set of possible outputs. Thus, $X\theta - y$ must be in the nullspace of X^T. Solving for θ, we get

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

Computationally, the hard part of using this expression is the matrix inversion, which takes $O(m^3)$ time. It may turn out that $X^T X$ is noninvertible, e.g. if some of the features are redundant. However, for this purpose it's good enough to use the pseudoinverse.

Note. Linear regression may appear weak, but its flexibility actually lies in the choice of features; you can fit anything with a line, if you choose the right (nonlinear) features. For example, if we use both x and x^2 as features, then we're really fitting a quadratic. The features can be chosen using knowledge of the data set. There are also learning algorithms that effectively automatically choose features.

Note. Why least squares? Suppose that the output and input variables are related by

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}$$

where the error term $\epsilon^{(i)}$ represents unmodeled effects or random noise. The central limit theorem motivates us to model the $\epsilon^{(i)}$ as iid multivariate Gaussians with zero mean and variance σ^2. Then the likelihood function is

$$L(\theta) = p(y^{(i)}|x^{(i)};\theta) = \prod_i \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right).$$

Maximizing the likelihood of the data is equivalent to minimizing the least squares cost function.

Note. Linear regression is a parametric algorithm, which means that after training, it can make predictions by storing a fixed number of parameters. A nonparametric algorithm instead needs space linear in the size of the training set. One example is locally weighted linear regression. Here, to output the prediction for a query point x, we minimize the cost function

$$J(\theta) = \sum_i w^{(i)}(x^{(i)}, x)\,(y^{(i)} - \theta^T x^{(i)})^2$$

and return $\theta^T x$, where the weights emphasize the data points near x, e.g.

$$w^{(i)} = \exp\left(-\frac{(x^{(i)} - x)^2}{2\tau^2}\right)$$

where τ is the bandwidth parameter. To construct the cost function at all, we need to keep all $x^{(i)}$. This is a useful method when the data is not too high-dimensional, and one wants to get a decent result without having to think about feature engineering.

Note. What's the difference between machine learning and statistics? Some people quip "$200,000 per year", or that machine learning is just "regressions done in California". There's definitely some truth to that, since people try to apply spin for funding applications and press releases. But more seriously, while many algorithms appear in both fields, machine learning cares more about the predictions $h_\theta(x)$, while statistics cares more about extracting and understanding the parameters θ. Thus, machine learning commonly includes much more complex procedures that are extremely powerful, at the cost of rendering the parameters uninterpretable.

1.3 Logistic Regression

Logistic regression works for binary classification, where the outputs are labels $y^{(i)} \in \{0, 1\}$.

- As a simple hack, we could use linear regression to find θ, compute $\theta^T x$, and report which of 0 or 1 it's closer to. However, this doesn't actually make any sense. For example, suppose that we are classifying tumors as malignant or benign. A very large tumor will almost certainly turn out to be malignant. Since this will be no surprise, such a data point should have very little effect on the cost function, but in least squares such outliers generally have a massive effect.

- The conceptual reason for this is that the likelihood (from which we can infer a cost function) is qualitatively different. Let's suppose that the data is generated by

  $$P(y = 1|x; \theta) = h_\theta(x).$$

  Then the log-likelihood of the data, which is effectively the negative of our cost function, is

  $$\ell(\theta) = \log L(\theta) = \sum_i y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$

  which is very different from the least squares cost function; we call it the cross-entropy loss.

- Since the data is generated from a probability in $[0, 1]$, it's natural to output a probability in $[0, 1]$. In a generalized linear model, we assume $h_\theta(x)$ can be written as

  $$h_\theta(x) = g(\theta^T x).$$

  However, this still leaves the choice of g open. In logistic regression, we choose

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

  which is called the sigmoid function. It is useful because it smoothly transitions between 0 and 1, the input z can be thought of as the log-odds ratio, and the resulting cost function has an analytically tractable form. We will give a more canonical justification for it below.

- Now, we need to evaluate the derivative of the log-likelihood. Suppressing the superscripts,

  $$\nabla_\theta \ell(\theta) = \left(\frac{y}{g} - \frac{1-y}{1-g}\right) \nabla_\theta g(\theta^T x) = (y(1-g) + (1-y)g)x = (y - h_\theta(x))x$$

  where we used the useful identity

  $$g'(z) = g(z)(1 - g(z)).$$

- Thus, in stochastic gradient ascent we would maximize the log-likelihood by updating

  $$\theta := \theta + \alpha(y^{(i)} - h_\theta(x^{(i)}))\, x^{(i)}$$

  which is remarkably the exact same functional form as in linear regression. The difference between the two algorithms is entirely in the choice of $h_\theta(x)$. In particular, there is no analogue of the normal equations here. The cost function is still convex, ensuring no local minima.

Note. There are many alternatives to gradient descent. Newton's method is a fast method to find the zeroes of a function, but the zeroes of the derivative are the extrema. Thus, to maximize $\ell(\theta)$ for a one-dimensional θ, we can perform the iteration

$$\theta := \theta - \frac{\ell'(\theta)}{\ell''(\theta)}.$$

For multiple features, this generalizes to

$$\theta := \theta - H^{-1}\nabla_\theta \ell(\theta)$$

where $H = \nabla_\theta^2 \ell(\theta)$ is the Hessian matrix. Applying this method to logistic regression is called Fisher scoring. Newton's method converges in much fewer steps than gradient descent. Formally, it has "quadratic convergence", heuristically meaning that for reasonable functions, if the error is $\epsilon \ll 1$, then the error after the next step is ϵ^2. However, each step takes much longer, since calculating H takes $O(n^3 m)$ time.

Note. Once we've computed $h_\theta(x)$, we can use the output values to construct a decision boundary, by assigning a class of 1 when $h_\theta(x)$ is above some cutoff. For logistic regression, the decision boundary is always a hyperplane perpendicular to θ. Like for linear regression, a more complicated decision boundary can be constructed by using nonlinear features.

Note. We could also try choosing g to force an output of 0 or 1, by letting $g(z) = \theta(z)$. If we continue to use the same update rule,

$$\theta := \theta + \alpha(y^{(i)} - h_\theta(x^{(i)}))\, x^{(i)}$$

then we have the perceptron learning algorithm. Since both $h_\theta(x^{(i)})$ and $y^{(i)}$ are 0 or 1, the update rule actually has a very simple form. In stochastic gradient descent, we can think of the algorithm as outputting a prediction $h_\theta(x^{(i)})$ as it sees each data point (thereby making it an "online" learning algorithm). Then the update rule is

$$\text{if prediction for } i \text{ wrong, then set } \theta := \theta \pm \alpha x^{(i)}.$$

In the 1960s, the perceptron was thought to describe how neurons behave, with a 1 describing a neuron firing. It also forms a crucial component of a neural network. However, it doesn't have the same probabilistic interpretation that linear or logistic regression have.

1.4 Generalized Linear Models

Linear and logistic regression are examples of generalized linear models (GLMs).

- A class of probability distributions is in the exponential family if it has the form
$$p(y;\eta) = b(y)\exp(\eta^T T(y) - a(\eta))$$
where η is the natural/canonical parameter, $T(y)$ is the sufficient statistic (often equal to y itself), $b(y)$ is the base measure, and $a(\eta)$ is the log partition function, which just normalizes the distribution. A choice of $T(y)$, $a(\eta)$, and $b(y)$ defines a family of distributions parametrized by η. In general η and $T(y)$ can be vectors, but we'll focus on one-parameter families.

- To derive the cost function for linear regression, we assumed $p(y|x;\theta)$ was Gaussian. For logistic regression, we assumed $p(y|x;\theta)$ was Bernoulli. Both of these distributions are in the exponential family. For Bernoulli(ϕ), where we took $\phi = g(\theta^T x)$, note that
$$p(y;\phi) = \phi^y(1-\phi)^{1-y} = \exp\left(\left(\log\left(\frac{\phi}{1-\phi}\right)\right)y + \log(1-\phi)\right)$$
which corresponds to
$$\eta = \log\frac{\phi}{1-\phi}, \quad T(y) = y, \quad b(y) = 1, \quad a(\eta) = -\log(1-\phi).$$
Interestingly, this means the probability is
$$\phi = \frac{1}{1+e^{-\eta}}$$
which is the sigmoid function.

- The Gaussian distribution was a two-parameter family, but we can just set $\sigma^2 = 1$ since it played no essential role. We have
$$p(y;\mu) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}(y-\mu)^2\right)$$
from which we read off
$$\eta = \mu, \quad T(y) = y, \quad b(y) = \frac{e^{-y^2/2}}{\sqrt{2\pi}}, \quad a(\eta) = \mu^2/2.$$
The multinomial, Poisson, gamma, exponential, beta, and Dirichlet distributions are all members of the exponential family. On the other hand, the set of uniform distributions on intervals $[c,d]$ is not a member, because the support of the distribution is determined by $b(y)$ alone, and hence cannot depend on the parameters c and d.

- More complicated families of distributions can be described with more complicated $T(y)$. For example, for a general multivariate Gaussian, we can take
$$T(y) = (y, yy^T), \quad \eta = (\Sigma^{-1}\mu, -\Sigma^{-1}/2)$$
where the inner product of the two matrices should be interpreted as an element-wise product.

- In a generalized linear model (GLM), we assume the output distribution $p(y;\eta)$ is in the exponential family, and furthermore that η for each data point depends linearly on the inputs,
$$\eta = \theta^T x$$
or for a multi-parameter family, $\eta_i = \theta_i^T x$, where θ is unknown.

- We fit θ using maximum likelihood. Suppressing data indices,

$$\ell(\theta) = \log p(y; \theta^T x) = \log b(y) + \theta^T x\, T(y) - a(\theta^T x)$$

which gives us a simple expression for the gradient,

$$\nabla_\theta \ell(\theta) = (T(y) - a'(\eta))x.$$

Furthermore, $\ell(\theta)$ is concave. This is the real power of the GLM: for a wide variety of possible data distributions, we automatically get a model that is easy to train.

- Taking the normalization condition and differentiating with respect to η gives

$$g(\eta) \equiv a'(\eta) = E[T(y;\eta)]$$

where $g(\eta)$ is called the canonical response function, and g^{-1} is the called the canonical link function. Thus, we can view $E[T(y;\eta)]$ as the prediction of the GLM for y, since it's the difference of the prediction and the reality that appears in the gradient.

- The GLM has further nice properties. Taking another derivative gives the identity

$$\mathrm{var}(y;\eta) = a''(\eta)$$

with further derivatives yielding higher cumulants.

- Ordinary least squares is simple, since here the canonical response function is the identity,

$$h_\theta(x) = E[y|x;\theta] = \mu = \eta = \theta^T x.$$

For a Bernoulli distribution, the canonical response function is the logistic function,

$$h_\theta(x) = E[y|x;\theta] = \phi = \frac{1}{1+e^{-\eta}} = \frac{1}{1+e^{-\theta^T x}}.$$

This explains why we used the logistic function earlier.

- For positive integer-valued y, we can use the Poisson distribution. For positive real y, which is relevant in "survival analysis", we could use a gamma or an exponential. For $y \in [0,1]$, which could represent a probability, we could use a beta distribution. Of course, the choice of distribution to use also depends on what we know about the problem.

Example. Consider a classification problem where $y \in \{1, 2, \ldots, k\}$. The data is multinomial distributed, and this is an example of a $(k-1)$-parameter set of distributions in the exponential family. To see this explicitly, let $\phi_i = p(y = i; \phi)$. Then the multinomial distribution is described by

$$\eta_i = \log(\phi_i/\phi_k), \quad T(y)_i = 1(y = i), \quad b(y) = 1, \quad a(\eta) = -\log(\phi_k)$$

as can be seen by plugging in the definitions. The outputs are the probabilities,

$$E[T(y;\eta)_i] = \phi_i = \frac{e^{\eta_i}}{\sum_{j=1}^k e^{\eta_j}}$$

where in a GLM, $\eta_i = \theta_i^T x$ and the θ_i are fit by maximizing the log likelihood. The quantity on the right-hand side here is the canonical response function $g(\eta)$ for the multinomial distribution. It is called the softmax function, and it generalizes the logistic function. This model can be trained by gradient descent in an almost identical fashion to logistic regression; the log-likelihood is the cross entropy between the distribution $E[T(y;\eta)_i]$ and $1(y = i)$.

1.5 Kernels

As mentioned, we can fit nonlinear functions with a linear model and nonlinear features. To make this distinction explicit, we'll write the "original" input values, called "attributes", as x, and the features as $\phi(x)$, where $\phi\colon \mathbb{R}^d \to \mathbb{R}^p$ is a feature map. We let there be n training examples.

- Stochastic gradient descent for linear regression takes the form

  $$\theta := \theta + \alpha(y^{(i)} - \theta^T \phi(x^{(i)}))\, \phi(x^{(i)}).$$

  A typical nonlinear feature map could contain the x_i themselves, along with all quadratic combinations $x_i x_j$ and cubic combinations $x_i x_j x_k$. But this makes every step of gradient descent take $O(p) = O(d^3)$ time, which could be slow if there are many attributes.

- However, note that each step of gradient descent adds a term to θ proportional to $\phi(x^{(i)})$. So if θ is initially zero, it can be written as

  $$\theta = \sum_i \beta_i \phi(x^{(i)})$$

  where we can think of β_i as the weight of training example i. (Also, the representer theorem states that for a wide variety of cost functions, the optimal θ can also be written in this form.)

- Now, all relevant quantities can be written in terms of inner products of the $\phi(x^{(i)})$. In one step of stochastic gradient descent on example i, the coefficient β_i updates by

  $$\beta_i := \beta_i + \alpha \left(y^{(i)} - \sum_j \beta_j \phi(x^{(j)})^T \phi(x^{(i)}) \right)$$

  where this expression works equally well if the $\phi(x^{(i)})$ are an undercomplete or overcomplete basis for feature space. The final prediction is

  $$\theta^T \phi(x^{(i)}) = \sum_j \beta_j \phi(x^{(j)})^T \phi(x^{(i)}).$$

- Thus, it suffices to precompute the values

  $$K(x, z) \equiv \phi(x)^T \phi(z)$$

  where the function K is called the kernel. It would seem this takes $O(n^2 p)$ time, but many kernels can be computed much faster. For example, for the cubic features above,

  $$K(x, z) = 1 + \sum_i x_i z_i + \sum_{ij} x_i x_j z_i z_j + \sum_{ijk} x_i x_j x_k z_i z_j z_k$$

  which has the simple form

  $$K(x, z) = 1 + x^T z + (x^T z)^2 + (x^T z)^3$$

  which only requires $O(d)$ time to compute. However, note that each step of gradient descent now takes $O(n)$ time, while previously it didn't scale with n at all.

- More generally, the kernel $K(x,z) = (x^T z + c)^k$ corresponds to all features up to order k, with c controlling the relative weighting of different orders.
- Intuitively, the dot product measures how close two vectors are, so a kernel function should fill the same role. This motivates the choice of the Gaussian/radial basis kernel,

  $$K(x,z) = \exp\left(-\frac{\|x-z\|^2}{2\sigma^2}\right).$$

  This is essentially an infinite-dimensional polynomial kernel, with a penalty for high degree terms parametrized by σ. For large enough σ, this kernel can perfectly separate *any* training set, but only by overfitting, drawing a narrow decision boundary around each data point. Decreasing σ regularizes the model. This is a case where the feature mapping is so complicated that it's better to not think about it at all.
- Clearly, not all functions $K(x,z)$ correspond to possible sets of features. However, many valid kernel functions can be constructed using the following rules.
  - $K(x,z) = 1$ is a kernel function, corresponding to the constant feature.
  - If $K(x,z)$ is a kernel function, so is $f(x)K(x,z)f(z)$, corresponding to changing the feature mapping from $\phi(x)$ to $f(x)\phi(x)$.
  - Kernel functions are closed under addition, corresponding to concatenating feature vectors.
  - Kernel functions are closed under multiplication, corresponding to taking the set of quadratic combinations of the elements of a feature vector.

  Using these rules, we can construct the kernels mentioned above.
- Alternatively, let $K_{ij} = K(x^{(i)}, x^{(j)})$ be the kernel matrix. Clearly, the kernel matrix must be symmetric to have a valid kernel. Moreover, letting $\phi_k(x)$ denote the k^{th} component of $\phi(x)$,

  $$z^T K z = \sum_{ij} z_i K_{ij} z_j = \sum_{ijk} z_i \phi_k(x^{(i)}) \phi_k(x^{(j)}) z_j = \sum_k \left(\sum_i z_i \phi_k(x^{(i)})\right)^2$$

  which implies that K is positive semidefinite.
- Mercer's theorem states that the reverse is true: if the kernel matrix is symmetric and positive semidefinite for any data set, then it is a valid kernel. The proof requires functional analysis. At the level of functions, it turns out $K(x,z)$ is a valid kernel if and only if

  $$\int dx\, dz\, f(x)K(x,z)f(z) \geq 0$$

  for any function f.
- The polynomial or Gaussian kernels work well for image classification, where the attributes are the pixel values. They are also useful for string classification, e.g. in text or genomics. In this case the feature space is the set of substrings of some length; this is extremely high-dimensional, but the kernel can be computed relatively quickly using dynamic programming.
- The kernel trick gets worse as the data set gets larger, since each step of gradient descent takes longer, and the training set must be saved at test time to output predictions. On the other hand, we'll see later that SVMs with kernels have sparse coefficients, mitigating this problem.

2 More Supervised Learning

2.1 Generative Learning

Our "discriminative" algorithms above were based on fitting a model of $p(y|x)$. A "generative" learning algorithm instead models $p(x|y)$ and $p(y)$, where the $p(y)$ are called the class priors. Given these, we can use Bayes' rule,

$$p(y|x) \propto p(x|y)p(y)$$

to infer $p(y|x)$. We begin with Gaussian discriminant analysis (GDA).

- In GDA, we assume $p(x|y)$ is multivariate Gaussian,

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

  For concreteness, let's suppose we're doing binary classification, and for simplicity suppose the Gaussians share the same Σ. Then

  $$y \sim \text{Bernoulli}(\phi), \quad x|y=i \sim \mathcal{N}(\mu_i,\Sigma).$$

- We can fit the parameters ϕ, μ_i, and Σ by maximizing the log-likelihood of the data. In our discriminative algorithms, the likelihood was the product of the conditional probability of seeing each $y^{(i)}$ given its $x^{(i)}$. For a generative algorithm, we model both x and y, so we instead want the product of the joint likelihood of the pairs $(x^{(i)}, y^{(i)})$.

- Explicitly, we have

  $$\ell(\phi,\mu_0,\mu_1,\Sigma) = \sum_{i=1}^m \log p(x^{(i)}|y^{(i)};\mu_0,\mu_1,\Sigma) + \sum_{i=1}^m \log p(y^{(i)};\phi)$$

  and setting the derivatives to zero gives

  $$\phi = \frac{1}{m}\sum 1\{y^{(i)}=1\}, \quad \mu_j = \frac{\sum 1\{y^{(i)}=j\}x^{(i)}}{\sum 1\{y^{(i)}=j\}}, \quad \Sigma = \frac{1}{m}\sum (x^{(i)}-\mu_{y^{(i)}})(x^{(i)}-\mu_{y^{(i)}})^T$$

  where 1 is an indicator function. In other words, the parameters are just the empirical values, e.g. the mean is the observed class mean.

- It turns out that $p(y=1|x) = g(\theta^T x)$ where g is the logistic function, and θ depends on the above parameters. However, it won't be the same θ as found by logistic regression; the two algorithms genuinely differ. More generally, if the $x|y=i$ are drawn from the same one-parameter exponential family distribution, differing only in the natural parameter, then $p(y=1|x)$ will have this form.

- If we model the two classes with distinct Σ's, then $p(y=1|x)$ is a logistic function of a quadratic in x, so the decision boundary (defined by a constant value of $p(y=1|x)$) is generally nonlinear.

- GDA makes stronger assumptions than logistic regression, since it additionally assumes $p(x|y)$ is Gaussian with common Σ. If this is true, then GDA is "asymptotically efficient", i.e. it is rigorously the best algorithm is the limit of large training sets, and will likely do better than logistic regression on small ones. Furthermore, GDA is more computationally efficient. Logistic regression works for more general situations, so it is generally preferred if one has a large data set and high computing power, but many important applications don't come with "big data".

Naive Bayes is a generative learning algorithm that works on feature vectors with discrete values. We'll use the running example of classifying text into spam and non-spam, where $x_j = 1$ if an email contains the j^{th} word in the dictionary.

- Given a dictionary of d words, it takes $O(2^d)$ parameters to specify a general form for $p(x|y)$, which is clearly unusable. The naive Bayes assumption is that all of the x_i are conditionally independent given y, so
$$p(x|y) = \prod_j p(x_j|y)$$
which reduces the number of parameters to $O(d)$.

- This is a distinct assumption from independence of the x_j, which would be unreasonable; such a model would not know that "Nigerian" and "prince" tend to occur together. Conditional independence is still not remotely true, e.g. within the class of spam emails, "Nigerian" makes "prince" more likely, but "cheap" makes "Viagra" more likely. However, the hope is that this does not adversely affect the classification accuracy, and it indeed works well in practice.

- The parameters $\phi_{j|y=k} = p(x_j = 1|y = k)$ and $\phi_k = p(y = k)$ are found by maximum likelihood, and again match the empirical probabilities,
$$\phi_{j|y=k} = \frac{\sum_i 1\{x_j^{(i)} = 1 \wedge y^{(i)} = k\}}{\sum_i 1\{y^{(i)} = k\}}, \quad \phi_k = \frac{\sum_i 1\{y^{(i)} = k\}}{m}$$
where the wedge denotes conjunction. Similar results hold when x_i has more than two possible values. Naive Bayes can also be applied to continuous variables by discretizing them.

- If there's a word that appears in the testing data but wasn't in the training data, then Naive Bayes will produce nonsense 0/0 predictions. Also, rare words might inadvertently cause 100% identification as spam or non-spam. In Laplace smoothing, we insert one dummy observation of every outcome to avoid this, meaning
$$\phi_{j|y=k} = \frac{\sum_i 1\{x_j^{(i)} = 1 \wedge y^{(i)} = k\} + 1}{\sum_i 1\{y^{(i)} = k\} + 2}.$$
This can be shown to be the optimal prescription under certain mathematical conditions.

- This specific prescription is also what we get if we assume a flat prior on each $\phi_{j|y=k}$. If $p(\phi)$ is flat, then after observing p positive counts and q negative counts, the posterior distribution is
$$p(\phi|p, q) = \phi^p(1 - \phi)^q$$
which has an expectation value of $(p + 1)/(p + q + 2)$. Variations on our prescription above, such as adding different numbers of dummy observations, correspond to a family of priors.

- In practice, the dictionary of words should be constructed from what actually appears in the training data, rather than copied from a standard dictionary, for computational efficiency. Also, it may be useful to remove stop words, since they occur often but carry little useful information.

- The algorithm we described above uses the "Bernoulli event model". To generate an email under its assumptions, we first decide if the email is spam or not (so y is Bernoulli), and then for the j^{th} word in the dictionary we independently decide if it's in the email or not (so x_j is binary and $x_j|y$ is Bernoulli). But for text classification, there's a better method. Keeping the first step the same, for the ℓ^{th} word in the email we independently decide which word it is (so x_ℓ is a word in the dictionary and $x_\ell|y$ is multinomial).
- We are still using the naive Bayes assumption of conditional independence, but now we can account for the frequency with which words are used. The parameters are $\phi_k = p(y = k)$ as before, and $\phi_{j|y=k} = p(x_\ell = j|y = k)$ for any ℓ. The likelihood is

  $$L(\phi) = \prod_i \phi_{y^{(i)}} \prod_{\ell=1}^{d_i} \phi_{x_\ell^{(i)}|y^{(i)}}$$

  where d_i is the number of words in email i. Maximizing it gives the expected results,

  $$\phi_{j|y=k} = \frac{\sum_i \sum_\ell 1\{x_\ell^{(i)} = j \wedge y^{(i)} = k\}}{\sum_i d_i 1\{y^{(i)} = k\}}, \quad \phi_k = \frac{\sum_i 1\{y^{(i)} = k\}}{m}.$$

  To do Laplace smoothing, we would add 1 to the numerator of $\phi_{j|y=k}$ and d to the denominator.

2.2 Neural Networks

Neural networks are based on a crude model of the brain.

- In the brain, neurons receive information from other neurons via dendrites, then decide whether or not to fire, and output that information along their axons. From the perspective of machine learning, the decision of firing or not is a highly nonlinear step, which allows neurons to learn complex hypotheses. Thus, in a neural network, each neuron weights inputs from other neurons, and applies a nonlinear function to form its output. We can think of logistic regression or the perceptron as a neural network with a single neuron.
- For regression and classification, we can use the same cost/loss functions as before. For concreteness, the cost for one example for regression with the least squares loss would be

  $$J^{(i)}(\theta) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2.$$

  In practice, however, it is very difficult to train a deep neural network with this loss. It is generally best to convert everything into a classification problem if possible, e.g. predicting a star rating could be viewed as a classification problem with five classes.
- For a single neuron, the output is

  $$h_\theta(x) = \sigma(w^T x + b)$$

  where b is a bias term, w is a vector of weights, and σ is a nonlinear activation function. Typical examples include

  $$\sigma(t) = \begin{cases} \max(t, 0) & \text{rectified linear unit (ReLU)} \\ 1/(1 + e^{-z}) & \text{sigmoid} \\ \tanh z & \text{hyperbolic tangent} \end{cases}.$$

Since the derivatives of these functions must be computed often, it's useful to have simple expressions for them,

$$\sigma'(t) = \begin{cases} \theta(t) & \text{ReLU} \\ \sigma(t)(1-\sigma(t)) & \text{sigmoid} \\ 1-\sigma(t)^2 & \text{hyperbolic tangent} \end{cases}.$$

- In a multi-layer fully-connected neural network, every neuron in a layer is connected to every neuron in the next. The output of layer k is

  $$a^{[k]} = \sigma(W^{[k]}a^{[k-1]} + b^{[k]})$$

  where $W^{[k]}$ is a matrix of weights, $b^{[1]}$ is a vector of biases, the inputs are $a^{[0]} = x$, and the outputs are $h_\theta(x) = W^{[r]}a^{[r-1]} + b^{[r]}$. Note that the $W^{[k]}$ need not be square, i.e. we can have any number of neurons per layer. For a real-valued output, the final layer has one neuron.
- This matrix notation is elegant, and it is also practically important. When neural networks are trained, as much as possible should be put into "vectorized" form. The recent interest in neural networks is due to their great power when they are sufficiently large, which requires computation power to be used as efficiently as possible.
- It's doubtful if neural networks actually resemble the brain. In deep learning, the layer structure is emphasized due to the efficiency of matrix multiplication, and one can have architectures with 1000 layers, which seem unlikely to exist in real brains. On the other hand, the "one learning algorithm" hypothesis says that distinct regions of the brain all run on essentially the same learning algorithm, which neural networks might partially capture. For example, if visual input is given to the auditory cortex in a lab animal, it will learn to see, though not well.
- Opponents of the "one learning algorithm" idea point out the great complexity of the brain, its division into many distinct subparts, and the existence of detailed instinctual knowledge (e.g. for language acquisition). In AI, Minsky's "society of mind" hypothesis posits that human intelligence arises from the interaction of a diverse range of relatively simple "agents".
- A linear model can deal with nonlinearity if it is given an appropriate feature map, which requires careful "feature engineering". Neural networks are useful because they essentially do this automatically; the output of each layer can be thought of as constructing useful features for the next layer to operate on. We can even think of the first $r-1$ layers as defining a kernel, and then swap out the final layer for one of the other algorithms we've considered.

Example. Consider two binary-valued inputs. A linear decision boundary, e.g. from a single neuron, can represent boolean operators such as (N)AND and (N)OR. Such a decision boundary can't represent XOR, but a two-layer neural network can, because XOR can be written in terms of a two-layer binary circuit.

Note. A neural network with a single hidden layer can approximate any function. To see this, note that any reasonable activation function becomes a step function if the weights are large enough. Let the output of each hidden neuron be a step function in the input, and let the output neuron just sum up the hidden neuron outputs. The conclusion follows since any function can be approximated as a sum of step functions.

While this “universality” property might sound powerful, it’s not relevant to practical learning, because almost everything is universal. For example, polynomials can approximate any function arbitrarily well, as can sums of cosines and sines, as can discretizations (which these step functions essentially are). This doesn’t mean that we can replace all of machine learning with Taylor series. What we really need are algorithms that efficiently learn and represent the *particular* patterns that exist in real data; this is the power of deep neural networks.

Next, we discuss how neural networks are trained.

- To perform gradient descent, it’s useful to define the intermediate quantity
$$z^{[k]} = W^{[k]} a^{[k-1]} + b^{[k]} = W^{[k]} \sigma(z^{[k-1]}) + b^{[k]}.$$
If we know $\delta^{[k]} = \partial J / \partial z^{[k]}$, then we can easily compute $\partial J / \partial z^{[k-1]}$ by the chain rule,
$$\frac{\partial J}{\partial z_i^{[k-1]}} = \frac{\partial J}{\partial z_j^{[k]}} \frac{\partial z_j^{[k]}}{\partial z_i^{[k-1]}} = \frac{\partial J}{\partial z_j^{[k]}} W_{ji}^{[k]} \sigma'(z_i^{[k-1]}).$$
In matrix/vectorized notation, this would read
$$\delta^{[k-1]} = (\delta^{[k]} W^{[k]}) \circ \sigma'(z^{[k-1]})$$
where the $\circ$ denotes element-wise multiplication.
- The final layer $\delta^{[r]}$ is easy to compute. For example, for the least squares cost function,
$$\delta^{[r]} = z^{[r]} - y^{(i)}$$
and the relation above allows us to recursively compute the $\delta^{[k]}$. Since this goes from the last layer to the first, it is known as “backpropagation”.
- Given the $\delta^{[k]}$, we can read off the gradients with respect to the parameters,
$$\frac{\partial J}{\partial W_{ij}^{[k]}} = \delta_i^{[k]} a_j^{[k-1]}, \quad \frac{\partial J}{\partial b_i^{[k]}} = \delta_i^{[k]}$$
or in matrix/vectorized notation,
$$\frac{\partial J}{\partial W^{[k]}} = \delta^{[k]} (a^{[k-1]})^T, \quad \frac{\partial J}{\partial b^{[k]}} = \delta^{[k]}.$$
Different sources will differ on the index conventions, though the substance is the same.
- Mathematically, backpropagation is just the chain rule from introductory calculus. But it’s an important idea because it’s far more efficient than a naive calculation of the gradient. Consider a neural network with p parameters. Naively, calculating the gradient due to a single parameter requires changing that parameter and seeing how the output changes, which takes $O(p)$ time, leading to $O(p^2)$ total time. Backpropagation memoizes the results at layer k to speed up the calculation at layer $k-1$, so the whole process takes $O(p)$ time.
- Also note that to perform backpropagation, we need to know all the $z^{[k]}$. But we already do this as an intermediate step while computing the output $z^{[r]}$ in “forward propagation”.

- In practice, it's useful to do "minibatch" stochastic gradient descent, i.e. using several training examples simultaneously. This is useful because using one training example at a time results in many matrix-vector multiplications that must be done sequentially. With a minibatch, every operation is a matrix multiplication, allowing us to parallelize.

- Notationally, we generalize all the vectors above to matrices, with each column representing one training example. Then we have, e.g.

  $$Z^{[k]} = W^{[k]}\sigma(Z^{[k-1]}) + B^{[k]}$$

  where each column of $B^{[k]}$ is a copy of $b^{[k]}$. The backpropagation steps become

  $$\Delta^{[k-1]}_{i\ell} = \Delta^{[k]}_{j\ell} W^{[k]}_{ji} \sigma'(Z^{[k-1]}_{i\ell})$$

  which in matrix notation is

  $$\Delta^{[k-1]} = ((W^{[k]})^T \Delta^{[k]}) \circ \sigma'(Z^{[k-1]}).$$

  The expressions for the parameter derivatives are similar, but with a sum over the new index,

  $$\frac{\partial J}{\partial W^{[k]}} = \Delta^{[k]}(A^{[k-1]})^T, \quad \frac{\partial J}{\partial b^{[k]}_i} = \sum_j \Delta^{[k]}_{ij}.$$

- In terms of linear algebra, some of the expressions above look unnatural, but computationally they are very efficient since they are just matrix multiplications, element-wise multiplications, and summations over rows. Also note that there is no need to explicitly construct the matrix $B^{[k]}$. The operations above with $B^{[k]}$ can be performed efficiently by "broadcasting" $b^{[k]}$.

- In general, we can run into objects with more than two indices, which we call tensors. Conventions differ on the index ordering, so other sources may have a different matrix multiplication order, or extra transposes, relative to us.

- In a physics context, we would usually indicate what is being kept constant for each partial derivative. For the neural networks we've considered above, this isn't necessary because of the layer structure: each layer only influences the final result through its influence on the next layer. Thus, partial derivatives of quantities at layer k are defined by changing layer k alone and computing its effect on the later layers. This can become slightly more subtle for networks without a straightforward layer structure.

- For classification with n classes, we can have n neurons in the final layer, with outputs $z^{[r]}_i$. These can be converted to probabilities using the softmax function,

  $$p_i = \frac{e^{z^{[r]}_i}}{\sum_j e^{z^{[r]}_j}}$$

  and the cost function is the log-likelihood $-\log p_{y^{(i)}}$.

- When we covered linear and logistic regression, the cost functions had clear probabilistic interpretations. The interpretation of a cost function is much less clear for a neural network; we instead just use cost functions if they lead to good performance. For example, another common option for classification is to take sigmoids in the final layer, rather than a softmax, and use the cross-entropy loss.

Note. As with all learning algorithms, neural networks need to be regularized for good performance. As described later, we can use L1 or L2 regularization; the latter tends to perform better but the former gives more interpretable parameters. Another natural idea would be to regularize by shrinking the neural network, but in general this is not preferred; we get the best performance by using as many parameters as computation power permits.

A better regularization technique, specific to neural networks, is "dropout", where for each minibatch we temporarily remove a random half of the hidden neurons; for predictions, we restore all neurons and halve the weights. Dropout was found to be very effective in the early 2010s. Heuristically, it works because it makes each neuron learn less fragile features, since it cannot rely on its neighbors. Another way to think about it is that it trains multiple neural networks inside the main neural network simultaneously, and the output is an ensemble average. Another useful idea is to augment the training set, e.g. in image classification we could add slightly rotated or cropped versions of the original training images. This keeps the neural network from fitting to irrelevant details.

Note. How do we choose the activation function? In general, it's just whatever works best, but heuristically it's useful for the activation function's range to match the output we want, so the sigmoid is good for classification, the tanh is good for bounded quantities, and ReLU is good for regression.

Both tanh and the sigmoid have the "vanishing gradient" problem: their derivatives can be very small when a neuron is saturated, and this problem worsens exponentially as we backpropagate, making the early layers in a deep network hard to train. ReLU does not have this problem as long as the argument is positive, but when the argument is negative the gradient vanishes entirely. This leads to the "dying ReLU" problem: in the course of training, neurons can effectively die, never outputting anything nonzero again. An alternative is the "leaky ReLU" function $\max(x/10, x)$. Also, even though all of our examples have gradients with magnitude less than 1, the fact that each neuron has many associated weights means that gradients can grow during backpropagation, leading to numeric instability; this is the "exploding gradient" problem. The inherent instability of gradient descent is one of the main obstacles in deep learning.

ReLU is the most popular activation function, because it is very cheap to evaluate and quick to train with, but research is done on alternatives. For example, Google researchers proposed the Swish activation function $x/(1+e^{-x})$, which seems to perform better than ReLU. It has the unusual property of being nonmonotonic! One can also use different activation functions for different layers. In general, not much is known for sure about activation functions.

Note. There are many small tricks needed to train neural networks reliably. It's important to initialize the weights to random values, in order to "break the symmetry" of the neurons in each layer. To prevent neurons from beginning already saturated, and hence training slowly, these values should be $O(1/\sqrt{n})$ where n is the number of inputs/outputs to each neuron; refinements of this idea include Xavier and He initialization. One should also normalize the inputs, again to avoid saturation. Finally, one should perform sanity checks on everything, such as the gradients, step sizes, regularization, and loss function, before committing to a long computation. Even if all these sanity checks pass, many things can go wrong in training, as amusingly shown here.

Also, we have talked about implementing (stochastic) gradient descent, but in practice one would often use an off-the-shelf gradient descent algorithm such as Adam or Adagrad, reviewed here. These algorithms often converge faster, using tricks such as adaptive learning rates for different parameters, maintaining "momentum" between update steps (i.e. treating the gradient as a force which produces an acceleration, in the presence of friction), annealing (gradually slowing) the learning rate, and using second derivative information. On the other hand, vanilla gradient descent with a predefined annealing schedule is quite robust. To prevent overfitting, training can also feature "early stopping", i.e. stopping once performance on the validation set stops improving.

Note. Some historical context about deep learning, ML, and AI. In popular culture, ML and AI are synonyms, but in most universities, they're distinct courses that cover completely different material. The professors for AI courses tend to be about three decades older, and both sides actively avoid mentioning each other. What's going on?

Technically, ML is just a subfield of AI, which consists of the techniques described in these notes. But when people contrast the two, they typically are comparing ML to "symbolic AI" (also called "good old fashioned AI"), a family of ideas that was dominant in the 1950s to 1980s. Symbolic AI explicitly encodes structure in terms of human-readable representations such as logical propositions. For example, a symbolic approach to recognizing a picture of an cat would begin by saying that it is a member of the category "animal", and has a property "legs" which has the value "four", and so on. Searle's Chinese room argument, for example, focused on such systems. This paradigm was so dominant that the cognitive science major at Stanford is called "symbolic systems".

But while it sounds intuitively appealing, symbolic AI was brittle, catastrophically failing when applied to real-world problems. Moravec's paradox was the observation that AI systems found emulating human reasoning about formally defined objects easy (for instance, AI systems rapidly progressed in chess playing, defeating the world champion in 1997), but couldn't perform sensorimotor tasks like image recognition as well as a human infant. This led to "AI winters" in the early 1970s and late 1980s, where funding for AI would collapse after a cycle of hype failed to lead to results. For this reason, some ML practitioners refer to ML as "the part of AI that actually works".

Philosophically, the reason for the failure is as in the Chinese room argument: symbolic systems are just pushing around symbols without knowing what they really mean, i.e. without "grounding" them in the real world. Practically, this leads to bad performance because real-world problems require a tremendous amount of tacit knowledge to solve, which cannot reasonably be encoded by hand. Many AI practitioners concluded that true AI required "embodied cognition", giving the AI sensory experience in the real world, but ML methods were able to circumvent this by picking up tacit knowledge from the training data by themselves.

Deep learning also has a long history, as it is the paradigmatic example of non-symbolic AI. Simple versions of neural networks, called perceptrons, were investigated in the 1950s and 1960s. However, they were infamously demolished in the 1969 book *Perceptrons*, by Minsky and Papert, which led to the growth of symbolic AI in its place. Among other things, *Perceptrons* showed that the simplest neural networks couldn't compute XOR (which is why it is a standard example in ML courses today) and dismissed deep neural networks as a "sterile" extension.

Of course, this wasn't the only reason people didn't use deep learning. Deep learning requires an extreme amount of computation power, and proponents of perceptrons made many bold claims that were infeasible given the computers available at the time. In the 1980s, deep learning was revived under the name of "connectionism", and backpropagation began to be used in 1985. This led to a surge of interest, and in 1989, Yann LeCun used neural networks to identify digits on the MNIST data set. However, progress stalled due to lack of computation power until around the year 2000, when GPUs began to be used. In the late 2000s, the paradigm of "big data" emerged, as large tech companies accumulated the vast data sets require to train large neural networks. In the 2010s, the combination of increasing computation power, improved neural network architectures and training techniques, and widely available huge data sets led to an explosion of spectacular results that continues today.

Despite all these innovations, training deep neural networks remains much harder than any other algorithm mentioned in these notes. Sometimes one will have to perform days of hyperparameter tuning before the output is even better than chance. If any of the other techniques in these notes are powerful enough to solve a problem, they generally can do it with less hassle, less data, and less computation time. But essentially all state-of-the-art results are achieved with deep learning.

2.3 Support Vector Machines

Support vector machines (SVMs) are a discriminative classification algorithm, qualitatively different from logistic regression. They began to be used in 1995, and are one of the best simple "off the shelf" learning algorithms, especially when there is little data available. However, interest in them is declining due to the greater availability of data and computation power, which allow neural networks to greatly outperform them.

- SVMs output decision boundaries, rather than probabilistic predictions. Suppose for concreteness that a data set can be perfectly separated by a hyperplane. Generically, multiple different hyperplanes will work. The key idea behind the SVM is that it picks the hyperplane with the largest "margin", i.e. with the greatest distance to any of the data points, which is called the optimal margin classifier. Note that it is determined by the closest data points to the boundary; these are the "support vectors".

- Specifically, for binary classification we output

  $$h(x) = \operatorname{sign}(w^T x + b)$$

  where the notation is conventional, and we have separated out the intercept b. The decision boundary is thus at $w^T x + b = 0$.

- For each data point i, we define the functional margin

  $$\hat{\gamma}^{(i)} = y^{(i)}(w^T x^{(i)} + b)$$

  where a high number indicates a confident and correct prediction. The functional margin of the entire data set is the minimum of the $\hat{\gamma} = \min_i \hat{\gamma}^{(i)}$.

- The functional margin is not invariant under scaling w and b, even though this doesn't change the decision boundary. So it is more useful to think about maximizing the geometric margin,

  $$\gamma = \min_i \gamma^{(i)}, \quad \gamma^{(i)} = \frac{\hat{\gamma}^{(i)}}{|w|}.$$

  We could then maximize the geometric margin using, e.g. gradient descent on w and b.

- However, there is a better way. We note that maximizing the geometric margin is equivalent to minimizing $|w|$ for fixed functional margin $\hat{\gamma} = 1$,

  $$\text{minimize}_{w,b}\ \frac{1}{2}|w|^2 \text{ such that } y^{(i)}(w^T x^{(i)} + b) \geq 1.$$

  This optimization problem only involves a quadratic "objective function" subject to linear constraints, so it can be solved using powerful "quadratic programming" methods. (The case where the objective function is also linear is "linear programming", and both are subsets of convex optimization.)

We can train an SVM more efficiently by transforming the "primal" optimization problem above to a "dual" problem.

- Consider a generic constrained optimization problem

  $$\text{minimize}_w \; f(w) \text{ such that } h_i(w) = 0.$$

  In the method of Lagrange multipliers, this can be solved by defining the Lagrangian

  $$\mathcal{L}(w, \beta) = f(w) + \sum_i \beta_i h_i(w)$$

  and setting its partial derivatives to zero, since this yields precisely the same constraints.

- However, for SVMs we have inequality constraints. For the problem

  $$\text{minimize}_w \; f(w) \text{ such that } h_i(w) = 0 \text{ and } g_i(w) \leq 0$$

  we can define a generalized Lagrangian,

  $$\mathcal{L}(w, \alpha, \beta) = f(w) + \sum_i \alpha_i g_i(w) + \sum_i \beta_i h_i(w).$$

- Consider the quantity

  $$\theta_{\mathcal{P}}(w) = \max_{\alpha, \beta \colon \alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta) = \begin{cases} f(w) & \text{if } w \text{ satisfies all constraints} \\ \infty & \text{otherwise} \end{cases}.$$

  Then solving the primal problem is equivalent to minimizing $\theta_{\mathcal{P}}(w)$. Let p^* be the corresponding optimal value.

- The dual problem switches the order of the maximum and minimum. We let

  $$\theta_{\mathcal{D}}(\alpha, \beta) = \min_w \mathcal{L}(w, \alpha, \beta)$$

  and the solution to the dual problem is

  $$d^* = \max_{\alpha, \beta \colon \alpha_i \geq 0} \theta_{\mathcal{D}}(\alpha, \beta).$$

  In general, the maximum of a minimum is at least the minimum of a maximum, so $d^* \leq p^*$, and the difference is called the duality gap.

- It can be shown that if f and the g_i are convex, and the h_i are affine, and the constraints g_i are strictly feasible (i.e. we can simultaneously have $g_i(w) < 0$), then there is no duality gap,

  $$p^* = d^* = \mathcal{L}(w^*, \alpha^*, \beta^*).$$

  In this case, by combining all of the constraints from solving the primal and dual problems, the solution must obey the KKT conditions,

  $$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial \beta_i} = 0, \quad \alpha_i^* g_i(w^*) = 0, \quad g_i(w^*) \leq 0, \quad \alpha^* \geq 0.$$

  The first two are familiar from Lagrange multipliers. The third is the dual complementarity condition, which states that if $\alpha_i^* > 0$, then the corresponding constraint is binding.

- Our SVM optimization problem has a generalized Lagrangian of

$$\mathcal{L}(w,b,\alpha) = \frac{1}{2}|w|^2 - \sum_i \alpha_i(y^{(i)}(w^T x^{(i)} + b) - 1)$$

  which has only inequality constraints. The conditions for the duality gap to vanish hold, so we can solve the dual problem instead.

- To construct $\theta_{\mathcal{D}}(\alpha, \beta)$ we set $\partial\mathcal{L}/\partial w_i = \partial\mathcal{L}/\partial b = 0$, giving

$$w = \sum_i \alpha_i y^{(i)} x^{(i)}, \quad \sum_i \alpha_i y^{(i)} = 0.$$

  Plugging these results back in gives

$$\theta_{\mathcal{D}}(\alpha) = \sum_i \alpha_i - \frac{1}{2}\sum_{ij} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)} \cdot x^{(j)})$$

  and the dual problem is to maximize this subject to $\alpha_i \geq 0$.

- At first glance, the dual problem might seem harder, because there are many more parameters, with one α_i per data point. However, by the dual complementarity condition, α_i is only nonzero for the support vectors, so the vast majority are zero!

- Furthermore, the dual problem is written in terms of inner products of the data points $x^{(i)}$. Once the optimal α_i are found, we can output predictions on a new data point x by computing

$$w^T x + b = \sum_i \alpha_i y^{(i)} (x^{(i)} \cdot x) + b$$

  which is also in terms of inner products. Thus, we can directly "kernelize" the SVM. Note that the value of b can be inferred by noting that $y^{(i)}(w^T x^{(i)} + b) = 1$ for the support vectors.

- So far, we've assumed the data set is linearly separable, but this might not be the case, even after using a feature map, in which case there is no solution at all. Moreover, a single outlier or misclassified data point can strongly affect the entire decision boundary. We can address this by adding "slack" to the constraints, yielding the optimization problem

$$\text{minimize}_{w,b,\xi}\, \frac{1}{2}|w|^2 + C\sum_i \xi_i \text{ where } y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i \text{ and } \xi_i \geq 0.$$

  We recover the previous formulation in the limit $C \to \infty$.

- Following the same logic as before, the dual problem is to maximize the same $\theta_{\mathcal{D}}(\alpha)$ as above, but subject to $0 \leq \alpha_i \leq C$ rather than just $0 \leq \alpha_i$. Again, $\alpha_i = 0$ for the "obvious" data points, while those that impact the decision boundary are nonzero.

- The sequential minimal optimization (SMO) algorithm is a particularly fast way to solve the dual problem, invented in 1998. The essential idea is to optimize only two of the α_i at once. This is fast, because each optimization step can be done in closed form. This approach, of optimizing a subset of the coefficients at once, is called coordinate ascent.

2.4 Bayesian Methods

In the classical learning algorithms we've seen above, we identify a single "best fit" model. However, in a Bayesian method we instead return a posterior distribution over models, giving a useful way to quantify uncertainty. We begin with the example of Bayesian linear regression.

- In Bayesian linear regression, we start with a prior distribution over the parameters. For concreteness, we let $\theta \sim \mathcal{N}(0, \tau^2 I)$, which is reasonable if the features have been appropriately normalized. We assume the data set is generated with Gaussian noise as before,

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}, \quad \epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2).$$

- Using Bayes' theorem, we have the parameter posterior,

$$p(\theta|S) \propto p(\theta) \prod_i p(y^{(i)}|x^{(i)}, \theta).$$

  Given a new input x_*, we have the posterior predictive distribution of the corresponding y_*,

$$p(y_*|x_*, S) = \int p(y_*|x_*, \theta) p(\theta|S) \, d\theta.$$

  Finally, a confidence region can be plotted by encompassing, for each x_*, a region in y_* that contains a given proportion of the probability. (Confidence regions can also be made in the frequentist approach, but they have a different meaning, as discussed in the notes on Statistics.)

- In general, these integrals are expensive to evaluate, and Bayesian methods usually use approximations such as maximum a posterior (MAP) estimation, where some posterior distributions are replaced with delta function peaks at their maxima.

- However, for the particular assumptions we have made, the solution can be written down in closed form! Using our earlier results about multivariate Gaussians, it can be shown that

$$\theta|S \sim \mathcal{N}(\mu_S, A^{-1}), \quad y_*|x_*, S \sim \mathcal{N}(x_*^T \mu_S, x_*^T A^{-1} x_* + \sigma^2)$$

  where X has the training examples in its rows, $y_i = y^{(i)}$, and

$$A = \frac{1}{\sigma^2} X^T X + \frac{1}{\tau^2} I, \quad \mu_S = \frac{1}{\sigma^2} A^{-1} X^T y.$$

  In other words, all distributions involved are Gaussian, thanks to its nice closure properties. Note that in the limit $\tau \to \infty$, the mean of y_* matches the maximum likelihood estimate. More generally, τ can be viewed as a regularization parameter, biasing the coefficients towards zero.

Next, we consider the more involved example of Gaussian process regression.

- A stochastic process is a probability distribution over functions with domain $\mathcal{X}$. When $|\mathcal{X}|$ is finite, this just reduces to the notion of a multivariate probability distribution.

- A Gaussian process is a stochastic process so that for any finite subcollection $x_1, \ldots, x_n \in \mathcal{X}$, the $f(x_i)$ are multivariate Gaussian. Explicitly, we have

$$\begin{pmatrix} f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} m(x_1) \\ \vdots \\ m(x_n) \end{pmatrix}, \begin{pmatrix} k(x_1, x_1) & \cdots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \ldots & k(x_n, x_n) \end{pmatrix} \right)$$

where m and k are the mean and covariance functions. We write

$$f(\cdot) \sim \mathcal{GP}(m(\cdot), k(\cdot, \cdot))$$

and the definition above implies

$$m(x) = \mathbb{E}[f(x)], \quad k(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))].$$

For $k(\cdot, \cdot)$ to yield a valid Gaussian process, the covariance matrix above must be PSD for any choice of x_i. Mercer's theorem thus implies $k(\cdot, \cdot)$ is valid if and only if it is a kernel function.

- A typical example is to set the mean function to zero, and use the kernel

$$k(x, x') = \exp\left(-\frac{1}{2\tau^2}|x - x'|^2\right)$$

which we call the squared exponential kernel to avoid confusion with Gaussians. A function drawn from this Gaussian process are smooth and vary little on scales less than τ. Another option is $k(x, x') = \min(x, x')$, which gives the Wiener process, which models Brownian motion.

- In Gaussian process regression, we wish to fit a function to data, and model the output as

$$y^{(i)} = f(x^{(i)}) + \epsilon^{(i)}$$

as before. We use a prior distribution over functions of

$$f(\cdot) \sim \mathcal{GP}(0, k(\cdot, \cdot))$$

for some valid covariance function $k(\cdot, \cdot)$.

- As before, we collect our training examples into a matrix X and vector y. We also define the vector f by $f_i = f(x^{(i)})$. Given testing examples X_* with corresponding vector f_*, we would like to compute the distribution of y_*.

- By the definition of a Gaussian process,

$$\begin{pmatrix} f \\ f_* \end{pmatrix} \Bigg| X, X_* \sim \mathcal{N}\left(0, \begin{pmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{pmatrix}\right)$$

where, e.g. we have $K(X, X)_{ij} = k(x^{(i)}, x^{(j)})$. The output values are

$$\begin{pmatrix} y \\ y_* \end{pmatrix} \Bigg| X, X_* = \begin{pmatrix} f \\ f_* \end{pmatrix} + \begin{pmatrix} \epsilon \\ \epsilon_* \end{pmatrix} \sim \mathcal{N}\left(0, \begin{pmatrix} K(X,X) + \sigma^2 I & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) + \sigma^2 I \end{pmatrix}\right).$$

- Since (y, y_*) is multivariate Gaussian, the distribution of y_* conditional on y is also multivariate Gaussian. Using identifies derived above, we have

$$y_* | y, X, X_* \sim \mathcal{N}(\mu^*, \Sigma^*)$$

where

$$\mu^* = K(X_*, X)Ay, \quad \Sigma^* = K(X_*, X_*) + \sigma^2 I - K(X_*, X)AK(X, X_*), \quad A = (K(X, X) + \sigma^2 I)^{-1}.$$

Note that even though the function spaces involved are intimidatingly infinite-dimensional, the definition of Gaussian processes in terms of finite subsets of $\mathcal{X}$ allows us to arrive at this answer using only finite-dimensional computations. However, since it is a nonparametric method, we need to keep the whole training set to make predictions.

- It turns out that Bayesian linear regression is equivalent to Gaussian process regression, upon using an appropriate kernel; however, showing this directly is messy. Just like Bayesian linear regression, Gaussian process regression automatically gives a confidence region. The kernel function itself plays the role of the regulator, and can be selected based on the structure of the data to improve performance.
- In geostatistics, Gaussian process regression is known as kriging, and was originally used to determine the distribution of underground gold for mining.

3 General Learning

3.1 Bias-Variance Tradeoff

In this section we describe some general features of machine learning. We begin with an explicit example of a bias-variance tradeoff in parameter estimation in statistics, which overlaps with the notes on Statistics.

- Consider an iid data set $(x^{(i)}, y^{(i)})$ with n data points, where $x^{(i)}$ is a d-element vector, combined into an $n \times d$ matrix X. In the context of statistical inference, the $x^{(i)}$ are fixed, and the corresponding $y^{(i)}$ is generated from it according to some probability distribution, with unknown parameter $\theta^* \in \mathbb{R}^d$.
- The goal is to estimate θ^*. An estimator $\hat{\theta}$ is a function of the data. Since the $y^{(i)}$ are random, $\hat{\theta}$ is itself a random variable, and its distribution is called the sampling distribution.
- We define the bias and variance as

  $$\mathrm{Bias}(\hat{\theta}) = \mathbb{E}[\hat{\theta} - \theta^*], \quad \mathrm{Var}(\hat{\theta}) = \mathrm{cov}[\hat{\theta}].$$

  The mean squared error is defined as

  $$\mathrm{MSE}(\hat{\theta}) = \mathbb{E}[|\hat{\theta} - \theta^*|^2].$$

  This can be decomposed into a bias and variance term using the usual "parallel axis theorem" reasoning, $\mathbb{E}[x^2] = \mathrm{var}(x) + \mathbb{E}[x]^2$, giving

  $$\mathrm{MSE}(\hat{\theta}) = \mathbb{E}[|\hat{\theta} - \mathbb{E}[\hat{\theta}]|^2] + |\,\mathbb{E}[\hat{\theta}] - \theta^*|^2 = \mathrm{tr}\,\mathrm{Var}(\hat{\theta}) + |\,\mathrm{Bias}(\hat{\theta})|^2.$$

  Both bias and variance contribute, and typically reducing one increases the other. Informally, the bias is the part that survives as the number of data points goes to infinity.
- As an example, we consider regularized linear regression. In this case, we assume

  $$y = X\theta^* + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \tau^2 I).$$

  The maximum likelihood estimator is

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

  which one can show is not biased.
- In L_2 regularized linear regression, also known as Ridge regression or Tikhonov regression, we minimize the cost function

  $$J(\theta) = \frac{\lambda}{2}|\theta|^2 + \frac{1}{2}\sum_i (y^{(i)} - \theta^T x^{(i)})^2$$

  for $\lambda > 0$, and by similar reasoning to that used to derive the normal equations, we have

  $$\hat{\theta} = (X^T X + \lambda I)^{-1} X^T y.$$

  Note that the inverse exists since $X^T X + \lambda I$ is PSD, since $X^T X$ is and λ is positive.

- This can be written in a slightly clearer form by letting
$$X^TX = UDU^T, \quad D = \operatorname{diag}(\sigma_i^2)$$
in which case
$$(X^TX + \lambda I)^{-1} = U \operatorname{diag}(1/(\sigma_i^2 + \lambda))U^T.$$
- Plugging in our expression for y, we have
$$\hat{\theta} = U \operatorname{diag}(\sigma_i^2/(\sigma_i^2 + \lambda))U^T\theta^* + (X^TX + \lambda I)^{-1}X^T\epsilon.$$
Since $\mathbb{E}[\epsilon] = 0$, the first term alone yields the bias. As is intuitive from the cost function, nonzero λ biases $\hat{\theta}$ towards zero.
- From the perspective of the variance, the first term is just a constant (since X is fixed), so only the second term contributes. It in turn is just a linear transformed Gaussian, so
$$\begin{aligned}\operatorname{cov}[\hat{\theta}] &= (X^TX + \lambda I)^{-1}X^T \operatorname{cov}[\epsilon]((X^TX + \lambda I)^{-1}X^T)^T \\ &= \tau^2(X^TX + \lambda I)^{-1}(X^TX)(X^TX + \lambda I)^{-1} \\ &= U \operatorname{diag}(\tau^2\sigma_i^2/(\sigma_i^2 + \lambda)^2)U^T\end{aligned}$$
which decreases as λ is increased.

In machine learning, the bias-variance tradeoff is slightly different.

- We are more interested in predictions than parameters, so we would like to quantify the bias-variance tradeoff in terms of prediction error. Let's suppose that the data is generated by
$$y = f(x) + \epsilon$$
where the random error ϵ is iid on the data points, and satisfies
$$\mathbb{E}[\epsilon] = 0, \quad \operatorname{var}(\epsilon) = \tau^2.$$
We don't know $f(x)$, but we would like to learn it from training data, so that our learned function $\hat{f}$ yields a good prediction for y on a previously unseen test data point (x_*, y_*).
- Again, we can define a mean squared error,
$$\operatorname{MSE}(\hat{f}) = \mathbb{E}\left[(y_* - \hat{f}(x_*))^2\right].$$
Plugging in $y_* = f(x_*) + \epsilon$ and expanding the square, we have
$$\operatorname{MSE}(\hat{f}) = \mathbb{E}[\epsilon^2] + \mathbb{E}\left[(f(x_*) - \hat{f}(x_*))^2\right].$$
Splitting the last term using "parallel axis theorem" reasoning again, we get
$$\operatorname{MSE}(\hat{f}) = \tau^2 + \mathbb{E}[\hat{f}(x_*) - f(x_*)]^2 + \operatorname{var}[\hat{f}(x_*)]$$
where the terms represent the irreducible error due to noise, bias, and variance.

- To relate this to our bias and variance for estimators, suppose that $f(x) = \theta^{*T} x$ and we use Ridge regression to find $\hat{f}$, giving $\hat{f} = \hat{\theta}^T x$. Then

$$\text{Bias}(\hat{f}) = \text{Bias}(\hat{\theta})^T x, \quad \text{Var}(\hat{f}) = x^T \, \text{Var}(\hat{\theta}) x.$$

Note. L2 regularization can be simply implemented in stochastic gradient descent, giving

$$J_\lambda(\theta) = \frac{\lambda}{2}\|\theta\|^2 + \frac{1}{2n}\sum(h_\theta(x^{(i)}) - y^{(i)})^2$$

That is, in every iteration we shrink θ by a constant factor. We can also regularize by adding a penalty proportional to the L1 norm $\|\theta\|_1$ of the weight vector. This is called lasso regularization, since many elements of the weight vector get pulled all the way down to zero, making the model more transparent. On the other hand, L2 regularization tends to perform better, and is easier to train since it has a smoother cost function. In "elastic net" regularization, we use both penalties at once, which gives some of the benefits of each.

It might seem strange that our discussion of regularization focuses on linear regression, which is the simplest algorithm we've covered. But even linear regression can overfit, typically if there are more features than data points. Of course, similar regularization techniques can be applied to essentially all the supervised learning algorithms we've discussed above, including neural networks.

Note. Adding terms to the cost function as in L1 and L2 regularization might seem ad hoc, but it has a probabilistic interpretation. In Bayesian linear regression (with a Gaussian prior on θ), the mean of the posterior distribution for θ is precisely the result of L2 regularization. Therefore, the output of L2 regression can be interpreted as MAP estimation under this prior. Using MAP estimation with a "Laplace" prior of the form $p(\theta) \sim e^{-\|\theta\|_1/\lambda}$ yields L1 regularization.

3.2 Practical Advice

For general problem, we don't expect a clean decomposition of the error into bias and variance terms. In fact, finding a general definition of bias and variance is an open question! But despite a lack of theoretical foundation, thinking in terms of bias and variance is essential in practice.

- In practice, we split our data set into a testing and training set, and split the training set into a training and a validation/development set. The model is trained on the training set, and errors for both the training set and validation set are computed.

- High bias (underfitting) corresponds to having a high error on both. It can be fixed by using an more expressive model or adding features. High variance (overfitting) corresponds to having a high error on just the validation set, which means the algorithm is fitting to noise in the training set. It can be fixed by getting more data, increasing regularization, or simplifying the model.

- A typical split is 70/30 for the training and validation sets. As a typical methodology, one might train a variety of models with different hyperparameters on the training set, and choose the one with the lowest error on the validation set.

- When data is scarce, we can do a more elaborate procedure to avoid "losing" training data. In k-fold cross validation, we instead split the training set into k equal subsets. We train each model k times, each time picking a different subset to act as the validation set, then average the validation errors. This method is called k-fold cross validation, and a typical value is $k = 10$.

- For cases with extremely scarce data, we can set k to the total number of data points, which is called leave-one-out cross validation. A disadvantage of higher k is the increased time required.
- The *final* performance of the model should be reported as error on the testing set, which should not be used at all during the above procedure. The reason we want a separate testing set is that typically one goes through many models. If we validated on the testing set, then we would effectively be fitting hyperparameters to it, making the error on the testing set lower than the true generalization error.
- On the other hand, this argument means that we are fitting hyperparameters to the validation set, so how can we know if we are overfitting to it? In practice, one can use multiple layers of validation sets, one for each cycle of hyperparameter tuning.
- A typical train/validation/test split would be 60/20/20. However, when we have very large data sets, the testing set can be much smaller (e.g. 0.1%), since its only purpose is to estimate the generalization error.

One particular example of model selection is feature selection.

- Often, one will have many possible features, but only a small proportion will be relevant to the learning algorithm. However, given d features there are 2^d possible subsets, so trying each subset is not an option.
- One heuristic idea is to use a greedy "forward search" which adds in features one at a time, at each step adding the one that results in the best validation error. This requires $O(d^2)$ calls to the learning algorithm. We can also do "backward search", removing the least useful feature at each step. Another option is to see the order in which the features "turn on" as λ is decreased in lasso regularization.
- These options are "wrapper model" feature selection algorithms, because they call the learning algorithm as a subroutine. A "filter" feature selection method computes a score for each feature reflecting how informative it is. Examples of scoring functions include the correlation and the mutual information,
  $$\mathrm{MI}(x_i, y) = \sum_{x_i, y} p(x_i, y) \log \frac{p(x_i, y)}{p(x_i)p(y)}.$$
  We then take the top k features, where k is chosen to minimize validation error.
- In information theory, there are statistics such as the Akaike information criterion or Bayesian information criterion, which measure how informative the model is relative to its simplicity.

Next, we give some more specific practical advice.

- In almost all cases, after putting together a machine learning algorithm, the result just "won't work". The outputs will be qualitatively wrong, and fixing it is like debugging.
- To see if there's enough data, try plotting a "learning curve", i.e. the cost functions as a function of the number of data points in the training set. If these curves have flattened out, the data is sufficient to train the model, though the model itself might still exhibit high bias.

- We've already discussed diagnosing bias and variance above. There can also be issues with the learning procedure, e.g. gradient descent might not be converging. To diagnose this, it's useful to run sanity checks, such as plotting the data, or comparing cost functions to a dummy model such as the one that always predicts the average.
- It is also important to clean and sanity check the data, by looking at summary statistics such as the range. Sometimes, features can be missing entirely, or have nonsensical values. In these cases, the corresponding data points can be thrown out entirely, or one can use "data imputation", filling the missing values in with a reasonable reference value.
- There are many other possible subtle issues with data. For example, the full data set might contain duplicates, which then appear in both the training and testing set, underestimating the generalization error. Or, the data might have irrelevant features, e.g. the infamous story of an algorithm that "learned" to detect camouflaged tanks by looking at the weather, since all photos with tanks were taken on cloudy days.
- Another possible issue is that one is minimizing the wrong cost function, i.e. a lower cost function does not correspond to qualitatively good performance. For example, in spam detection one might want to penalize false positives more than false negatives, but the logistic regression cost function does not account for this.
- Hyperparameters should generally be tuned by trying values on a log scale. For more than one hyperparameter, try an automated "grid search". This can get very inefficient for many hyperparameters, in which case one can try specialized hyperparameter selection algorithms, such as random search (values picked randomly from a pre-specified distribution) or Bayesian hyperparameter optimization (choosing the next values to try based on performance of past ones). There are even gradient-based or evolutionary optimization techniques.

Example. Consider a reinforcement learning setup for flying a helicopter, which is trained to minimize a cost function in a simulator, but qualitatively doesn't work well in reality. If the setup works qualitatively well in simulation but not in reality, then the problem is in the simulator. (For example, the simulated detector inputs might have more or less noise than in reality.) Otherwise, if the real-life human control achieves a lower cost function than the algorithm, then the cost function is not being minimized properly. But if the human has a higher cost function, then the cost function itself is not reflecting good autonomous flight.

Example. Consider a facial recognition algorithm. Often, such algorithms are made of many components strung together, such as one that cleans up the image, one that detects the face, one that detects the eyes, and so on. In this case, we can determine how much error is attributable to each component by plugging in the ground truth and seeing how the accuracy changes. An "ablative analysis" runs in the opposite direction. We can see how much, e.g. a new feature helped a classifier by removing it entirely. This is important in applications where speed is a concern, or in research where one wants to understand the reason for good performance.

3.3 Evaluation Metrics

Above, we have emphasized mathematically motivated cost functions. However, the metric one uses to evaluate the quality of a ML model often differs from the cost function, and depends on the real-world context. For concreteness, we focus the case of binary classification models that output real-valued scores, such as logistic regression.

- Suppose we set a threshold for the output score, so that everything about the threshold is classified as positive. Let the number of positive and negative examples be P and N, with a total number of examples S. Then the prevalence of positive examples is P/S.
- Upon applying the classifier, the positive examples are split into true positives and false negatives, while the negative examples are split into true negatives and false positives. These four numbers are typically presented together in a "confusion matrix". In statistics, we call false positives type I errors, and false negatives type II errors.
- Given these quantities, we can define a number of evaluation metrics.
  - The accuracy is the fraction of examples classified correctly, $(\mathrm{TP} + \mathrm{TN})/\mathrm{S}$.
  - The precision is the fraction of positive labeled examples that are positive, $\mathrm{TP}/(\mathrm{TP} + \mathrm{FP})$.
  - The recall/sensitivity is the fraction of positive examples that are labeled positive, TP/P.
  - The negative precision and negative recall/specificity are defined analogously.
  - In general, there is a tradeoff between precision and recall, and sensitivity and specificity. The F_1 score is the harmonic mean of precision and recall, and hence accounts for both.

  The point of using evaluation metrics beyond just the accuracy is that often the classes are asymmetric. For example, often the positive class is rare, so it is reasonable to define metrics that focus on it; a typical cost function would be swamped by the majority class.
- We can also use a weighted accuracy, which assigns a weight to each entry in the confusion matrix. This is useful if the weights reflect our utility, e.g. they can account for false positives hurting more or less than false negatives.
- We can adjust the threshold that divides positively and negatively labeled examples, leading to a tradeoff against specificity and sensitivity. This can be visualized using a ROC (receiver operating characteristic) curve, which plots the two against each other. The general quality of the classifier can be quantified by the area under the ROC curve. Note that randomly guessing positive with probability p would produce a straight ROC curve, and hence an area of 1/2. Similarly, we can plot the precision against the recall.
- Everything mentioned above only depends on the ranking of the scores. However, it is also useful to have a model make "confident" predictions, i.e. if the scores are probabilities, positive examples should have high probabilities. This is just the motivation behind the standard log-loss cost function used in logistic regression, but it can also be used as an evaluation metric for other models.
- Calibration is the property that data points assigned probability p turn out positive with probability p. This is quantified by the Brier score, which is the mean square deviation of the predictions $h(x^{(i)})$ from the $y^{(i)}$. We used this as a cost function for linear regression, and in the context of classification it is useful because it is a proper scoring rule, i.e. it is minimized when the classifier is calibrated.

Some of these ideas about scoring rules come from decision theory, which is in turn tied to game theory and Bayesian statistics.

3.4 PAC Learning

In this section, we discuss some rigorous theorems about learning, within the "probably approximately correct" (PAC) learning framework. Though we have discussed validation sets above, we'll put them aside and just assume a whole training set, for mathematical simplicity.

- We assume that our n training examples are drawn iid from the same distribution $\mathcal{D}$. For concreteness, we focus on binary classification.
- A hypothesis is a function from the sample space to $\{0, 1\}$. For a hypothesis h, we define the training error (also called the empirical risk) as

  $$\hat{\epsilon}(h) = \frac{1}{n} \sum_{i=1}^{n} 1\{h(x^{(i)} \neq y^{(i)}\}.$$

  The generalization error is the expected error on a new data point drawn the same way,

  $$\epsilon(h) = P_{(x,y)\sim\mathcal{D}}(h(x) \neq y).$$

  This assumption that this test point is drawn from the same distribution as the training data is one of the strongest assumptions of PAC learning.
- We consider a set of hypotheses $\mathcal{H}$. The simplest way to pick one is to use the one that minimizes the empirical risk,

  $$\hat{h} = \operatorname*{argmin}_{h\in\mathcal{H}} \hat{\epsilon}(h).$$

  Our goal is to show that $\epsilon(\hat{h})$ is, with high likelihood, not too much higher than $\hat{\epsilon}(\hat{h})$, i.e. that $\hat{h}$ is probably approximately correct. This gives a theoretical guarantee on performance.
- The union bound states that for any k events A_i,

  $$\mathbb{P}(A_1 \cup \cdots \cup A_n) \leq \mathbb{P}(A_1) + \cdots + \mathbb{P}(A_k).$$

  The Hoeffding inequality/Chernoff bound states that for iid random variables $Z_i \sim \text{Bernoulli}(\phi)$, the mean $\hat{\phi} = \sum_i Z_i/n$ is bounded near its expectation value by

  $$\mathbb{P}(|\phi - \hat{\phi}| > \gamma) \leq 2\exp(-2\gamma^2 n).$$

We now establish a theorem for a finite hypothesis class, where $|\mathcal{H}| = k$.

- First, for each hypothesis h_i we can bound the difference between the training error and generalization error using the Hoeffding inequality,

  $$\mathbb{P}(|\epsilon(h_i) - \hat{\epsilon}(h_i)| > \gamma) \leq 2\exp(-2\gamma^2 n).$$

  We can turn this into a bound on all the hypotheses using the union bound,

  $$\mathbb{P}(\text{for all } h \in \mathcal{H}, |\epsilon(h) - \hat{\epsilon}(h)| > \gamma) \leq 2k\exp(-2\gamma^2 n).$$

  This is a "uniform convergence" result since it applies to all the hypotheses.

- This result can be rewritten in several illustrative ways. Define

  $$\delta = 2k \exp(-2\gamma^2 n).$$

  Then we can give a "sample complexity" bound,

  $$\text{if } n \geq \frac{1}{2\gamma^2} \log \frac{2k}{\delta} \text{ then } |\epsilon(h) - \hat{\epsilon}(h)| < \gamma \text{ for all } h \in \mathcal{H} \text{ with probability at least } 1 - \delta.$$

  Alternatively, we can emphasize δ,

  $$\text{with probability at least } 1 - \delta, |\epsilon(h) - \hat{\epsilon}(h)| < \sqrt{\frac{1}{2n} \log \frac{2k}{\delta}} \text{ for all } h \in \mathcal{H}.$$

- We can use the above result twice to bound the generalization error of $\hat{h}$ to the actual best hypothesis $h^* = \operatorname{argmin}_{h \in \mathcal{H}} \epsilon(h)$. With probability at least $1 - \delta$,

  $$\epsilon(\hat{h}) \leq \hat{\epsilon}(\hat{h}) + \gamma \leq \hat{\epsilon}(h^*) = \gamma \leq \epsilon(h^*) + 2\gamma.$$

  This bound illustrates the bias-variance tradeoff, with $\epsilon(h^*)$ representing the bias, and 2γ representing the variance, because at fixed δ, increasing k increases γ.

- Unfortunately, many of our algorithms use infinite $|\mathcal{H}|$, since they have real-valued coefficients. Informally, real numbers have finite machine precision, so a hypothesis class parametrized with d real numbers really contains $(2^{64})^d$ hypotheses. Plugging this in above gives a sample complexity $n \geq O(d/\gamma^2)$.

- More rigorously, we can characterize the size of a hypothesis class using the Vapnik-Chervonenkis dimension $\mathrm{VC}(\mathcal{H})$. For binary classification, the VC dimension is defined as the largest n for which there exists a set of n points that the hypothesis class can shatter, i.e. if $\mathcal{H}$ contains hypotheses with all 2^n possible labelings of the points.

- For example, linear classifiers over two features have a VC dimension of 3, and discrete hypothesis classes have a VC dimension of at most $\lfloor \log_2 k \rfloor$. For most "reasonable" infinite hypothesis classes, the VC dimension is roughly equal to the number of real parameters.

- One of the most famous theorems of learning theory is that if $\mathrm{VC}(\mathcal{H}) = d$, then

  $$\text{with probability at least } 1 - \delta, |\epsilon(h) - \hat{\epsilon}(h)| \leq O\left(\sqrt{\frac{d}{n} \log \frac{n}{d} + \frac{1}{n} \log \frac{1}{\delta}}\right) \text{ for all } h \in \mathcal{H}.$$

4 Reinforcement Learning

4.1 Markov Decision Processes

Reinforcement learning (RL) solves qualitatively different problems than supervised learning.

- A typical RL problem is to drive a vehicle or maneuver a robot. In these cases, supervised learning is not useful because we do not know ahead of time what "good" and "bad" inputs are; we only can identify good and bad results. We leave it to the RL algorithm to figure out which inputs lead to good results.

- As a simple example, we will model the environment as a Markov decision process (MDP), containing a set S of states and a set A of actions. If one is at state s at some timestep and takes action a, then the probability distribution of the next state is P_{sa}. Finally, at each timestep there is a reward $R\colon S \times A \to \mathbb{R}$. For notational simplicity, we'll suppose that the reward only depends on the state, though this assumption can be lifted without much trouble.

- Given a sequence of states
$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots$$
we define the total payoff as
$$\sum_{i=0}^{\infty} \gamma^i R(s_i)$$
where $\gamma \in [0, 1)$ is called the discount factor. The goal of the RL algorithm is to maximize the expected payoff. The purpose of the discount factor is to make the sum well-behaved, and to give the algorithm a sense of urgency, getting positive rewards as quickly as possible and deferring negative rewards as long as possible.

- A policy (also called a controller, in the context of control theory) is a function $\pi\colon S \to A$ which prescribes an action for each state. The value function of a policy is the expected payoff,
$$V^\pi(s) = \mathbb{E}[R(s_0) + \gamma R(s_1) + \ldots | s_0 = s, \pi].$$
Using the definition of $V^\pi(s)$ on the right-hand side gives the Bellman equations,
$$V^\pi(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^\pi(s')$$
where the first term represents the immediate reward, and the second term represents the expected future reward. Focusing on discrete state spaces for now, the Bellman equations are a set of $|S|$ linear equations, which can be used to solve for $V^\pi(s)$.

- For each state, the optimal value function is the highest possible value function for any policy,
$$V^*(s) = \max_\pi V^\pi(s).$$
By definition, $V^*(s)$ obeys a version of the Bellman equations,
$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s').$$

- The first possibly unintuitive result is that $V^*(s)$ can be attained, for all states, by a *single* policy π^*. This policy is defined by

  $$\pi^*(s) = \operatorname*{argmax}_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s')$$

  and its value function is $V^{\pi^*(s)} = V^*(s)$ because it satisfies the Bellman equations for $V^*(s)$.

- This result is due to the "memoryless" nature of the reward function in an MDP. For example, if reward was instead given for having the learner return to the initial state, whatever that initial state was, then $V^*(s)$ for each s is attained by a strategy that tries to return to s, so there is no policy that attains $V^*(s)$ for all s simultaneously. (On the other hand, literally any environment including this one can in principle be viewed as an MDP in a larger state space; in this case the state space would be $S \times S$ where the first factor represents the initial state.)

- There are two useful iterative methods to compute π^*. In value iteration, we find V^* and use it to infer π^*. We initialize $V(s) = 0$ and update

  $$V(s) \coloneqq R(s) + \max_{a \in A} \gamma \sum_{s'} P_{sa}(s') V(s').$$

  We loop over the states repeatedly until convergence. This can be done synchronously (updating all the $V(s)$ at once after one full loop), or asynchronously (updating one $V(s)$ at a time).

- To understand why value iteration works, it's easier to think in terms of synchronous updates. Suppose time suddenly ends at $t = 0$. After the first iteration, $V(s) = R(s)$, which means $V(s)$ accounts for the immediate reward only; it thus computes the optimal value as if the agent started at $t = 0$. After the second iteration, $V(s)$ contains both the immediate reward and the best possible reward in the next timestep, so it's as if the agent started at $t = -1$. Thus, we are computing $V^*(s)$ in "dynamic programming" style. As long as the rewards are bounded, this process converges over many updates because of the discount factor γ.

- Formally, the update step defines a "Bellman backup operator" on the space of value functions. We can show convergence by demonstrating that it is a contraction mapping.

- In policy iteration, we initialize π to some random policy π_0 and update

  $$V(s) \coloneqq V^\pi(s), \quad \pi(s) \coloneqq \operatorname*{argmax}_{a \in A} \sum_{s'} P_{sa}(s') V(s')$$

  synchronously. In other words, $\pi(s)$ is set to be greedy with respect to the value function defined by the previous $\pi(s)$.

- The intuition for why policy iteration converges is similar to value iteration. We now suppose that at time $t = 0$, the agent is forced to begin executing π_0. After the first iteration, π is the best policy if the agent starts at $t = -1$ and knows this will happen at $t = 0$. After the second iteration, π is the best policy if the agent starts at $t = -2$, and so on.

- Unlike value iteration, policy iteration converges to the *exact* optimal policy, and hence the exact optimal value function, in a finite number of iterations. Policy iteration tends to be faster for small $|S|$, but for larger $|S|$ it becomes impractical because it requires computing $V^\pi(s)$ at every step, which naively takes $O(|S|^3)$ time. Most modern applications use value iteration.

The above discussion assumed that everything about the MDP was known in advance, but in reality many parameters are unknown.

- For example, consider learning to play a videogame. The RL agent knows only the actions A ahead of time. The states, transition probabilities, and rewards must be learned from experience.
- These constraints make RL nontrivial, especially when the rewards are sparse. In this case, the agent faces an "explore/exploit" tradeoff, where it must balance the benefit of accruing known rewards with exploring the possibility of potential higher rewards. It also faces the "credit assignment problem": if the reward is a binary win or loss, then it can be challenging for the agent to understand why it won or lost. These issues are exacerbated by the potentially huge size of the state space, which can be subject to the curse of dimensionality, and the fact that the agent might not even know exactly what state it's in.
- As a first step, let's consider the case where P_{sa} is unknown. Then the RL agent can infer it from experience using a maximum likelihood estimate,

  $$P_{sa}(s') = \frac{\text{number of times we took action } a \text{ in state } s \text{ and got to } s'}{\text{number of times we took action } a \text{ in states}}.$$

  These probabilities can be efficiently updated over time if we keep track of the counts in the numerator and denominator. (For 0/0, we assume a uniform distribution.) This is a basic example of how RL algorithms can use supervised learning ideas as subroutines.
- Thus, one can use a modified value iteration. After initializing π randomly, we repeat the following until convergence:

  1. Execute π in the MDP for some number of trials.
  2. Using the accumulated experience, update our estimate for P_{sa}.
  3. Using these P_{sa}, apply value iteration to get an estimated value function V.
  4. Set π to be the greedy policy with respect to V.

  For performance, it's useful to save V between iterations, so that value iteration doesn't have to start over from scratch.
- To increase the amount of exploration, we could use "ϵ-greedy" exploration, where in step 1 we execute a random action with probability ϵ. However, for larger ϵ convergence will be slower. Alternatively, in "Boltzmann" exploration, the chance of exploring a seemingly less promising option is proportional to a Boltzmann factor $e^{V/T}$. Over time, ϵ and T can be annealed. Another idea is "intrinsic motivation", where the RL agent is rewarded for reaching new states.

Another complication is that often the state space is continuous.

- For example, an inverted pendulum on a cart has a $4d$ state space (position, linear velocity, angle, angular velocity) and a $1d$ action space (acceleration). A helicopter has a $12d$ state space (six for translation, six for rotation) and a $4d$ action space.
- The state/action space can depend on the level of description. For example, if we want to navigate a car over a huge distance, we can use a $2d$ state space (its current position) and a $2d$ action space (its velocity). But for navigating a specific road, we actually want to include the

velocity in the state space; the action space would instead be the position of the gas pedal and orientation of the steering wheel. If we wanted to model avoiding a car crash, we might need to account for the time it takes to turn the wheel, so its orientation would go into the state space and its angular velocity would be in the action space, and so on.

- A simple idea is to discretize the state space. This works well when the state space is low-dimensional (in practice, $d \leq 3$, though higher d can be made to work with clever choices of discretization), but then degrades due to the curse of dimensionality. Discretizations can approximate any function, just like neural networks, but they are inefficient and generalize poorly. For example, they give no prediction at all for bins containing no observations, even if they are surrounded by bins with observations, because they ignore this structure.
- Instead, we will consider approximating V^* directly, using "fitted value iteration". As a first step, we assume we have a model/simulator for the environment, which is a black box that takes the current state and action and outputs a next state.
- In some cases, such as in some games, the model is just the known P_{sa} themselves. In other cases, such as in robotics, the model can be derived from the laws of physics. Or, the model can be inferred from experience as described above, in which case constructing a model is just a regression problem. The model's output could be deterministic, or it could be stochastic, e.g. by adding Gaussian noise. This noise helps prevent the RL agent from learning "brittle" strategies.
- For concreteness, we suppose S is continuous but $|A|$ is small. Recall that in value iteration, we perform the update

  $$V(s) := R(s) + \gamma \max_a \mathbb{E}[V(s')|s' \sim P_{sa}].$$

  The main idea of fitted value iteration is to use the model to estimate the right-hand side.
- For concreteness, $V(s)$ can be taken to be linear in some feature mapping of s,

  $$V(s;\theta) = \theta^T \phi(s).$$

  In each step of fitted value iteration, we start with a set of coefficients θ. We sample n states $s^{(i)}$ and use the model to estimate $\mathbb{E}[V(s';\theta)|s' \sim P_{s^{(i)}a}]$ for each action a, giving values $y^{(i)}$ for the right-hand side. Finally, we set the new θ by fitting it to the data points $(s^{(i)}, y^{(i)})$ as usual in linear regression.
- Fitted value iteration doesn't have the same kinds of convergence guarantees as value iteration, but it works well for many problems in practice.
- The same idea holds for different forms of $V(s;\theta)$. For instance, "deep RL" essentially consists of making $V(s;\theta)$ a neural network.
- Once we have a suitable approximation for V^*, we can use the model again to infer π^*. Note that if the model has stochastic noise, then we can turn the noise off for this step to save time. This doesn't make the model brittle, since the noise already has suitably smoothed V^*.

- This approach is “model based”, but there are also “model free” approaches that never construct P_{sa} at all; the contrast is like that between generative and discriminative supervised learning algorithms. Both approaches are used in practice. Model free approaches are more general, but take much longer to train, so model based approaches are more common in robotics.

- For example, Q-learning is a model-free approach where everything is phrased in terms of the Q-function $Q(s, a)$, which gives the expected future reward assuming we take action a in state s. The Q-function is learned directly; no reference is made to what future states a leads to.

4.2 Policy Gradient

5 Unsupervised Learning

5.1 Clustering

Consider the grouping of data $x^{(i)}$ into cohesive clusters. This is an unsupervised learning problem, as no labels $y^{(i)}$ are given.

- In the k-means clustering algorithm, we initialize k cluster centroids μ_j randomly. We assign each data point to its nearest cluster centroid, so that the cluster labels $c^{(i)}$ are

  $$c^{(i)} := \underset{j}{\operatorname{argmin}} \|x^{(i)} - \mu_j\|^2.$$

  Then we reassign each cluster centroid to the centroid of the data points inside,

  $$\mu_j := \frac{\sum_i 1\{c^{(i)} = j\}\, x^{(i)}}{\sum_i 1\{c^{(i)} = j\}}.$$

  This is repeated until convergence.

- To see that this converges, note that we can define a distortion/cost function

  $$J(\mu_j, c^{(i)}) = \sum_i \|x^{(i)} - \mu_{c^{(i)}}\|^2.$$

  The two update steps in k-means perform coordinate ascent, minimizing this cost function with respect to the $c^{(i)}$ and μ_j, respectively. Thus, the cost function decreases monotonically, though it may be trapped in a local minimum. To address this, we can run the algorithm multiple times. Future data points are assigned to clusters by whichever one has the least cost.

- We can also get variations on k-means by changing the distance function. However, for a general distance function, the μ_j update step can't be done in closed form. Instead we can use k-medoids, where the μ_j are required to be "exemplars", i.e. members of the data set. When we update μ_j, we thus only need to minimize over the data set.

- In general, the cost function will be lower the higher k is, but taking k arbitrarily high would be useless. The "best" value of k depends on what the clustering is being used to do, but an old rule of thumb is that we should stop when $J_{\text{min}}(k)$ hits an "elbow", after which it stops falling as steeply.

- A more principled approach is "gap statistics", where we estimate how large $J_{\text{min}}(k)$ is expected to be for a hypothetical data set without structure, and choose the k so that the actual $J_{\text{min}}(k)$ looks best in comparison.

- Alternatively, in some cases the clustering might be "semi-supervised", where a small number of data points come with labels. In this case, we can split the labeled points into training and validation sets, fix the labels of the points in the training set while training on the unlabeled points, and choose k to minimize the validation error.

5.2 Expectation Maximization

The expectation maximization (EM) algorithm is a general technique for fitting a maximum likelihood model in the presence of latent variables. As an example, we consider the mixture of Gaussians model, which can be used for density estimation, i.e. estimating the probability distribution that generated some given data, which can be useful for anomaly detection.

- The simplest technique for density estimation would just be fitting a Gaussian to the whole data set. But often this is not general enough, because the data will have clusters. The next simplest thing to do is to fit a mixture of Gaussians.
- Specifically, we assume that
$$x^{(i)}|z^{(i)} \sim \mathcal{N}(\mu_j, \Sigma_j), \quad z^{(i)} \sim \text{Multinomial}(\phi)$$
where the $z^{(i)}$ indicate which Gaussian to use. (Statisticians would call the individual $z^{(i)}$ "categorical" distributed, and the set of total counts of each $z^{(i)}$ value "multinomial" distributed; the distinction between the two is analogous to that between the Bernoulli and binomial distributions. However, we will treat the two as interchangeable.)
- We wish to maximize the log-likelihood,
$$\ell(\phi, \mu, \Sigma) = \sum_i \log \sum_{z^{(i)}} p(x^{(i)}|z^{(i)}; \mu, \Sigma) p(z^{(i)}; \phi).$$
This is similar to GDA, but the $z^{(i)}$ are now unobserved ("latent") variables, rather than provided labels. This makes it impossible to maximize the log-likelihood in closed form.
- Instead, we use a two-step procedure as in k-means. In the E-step, we set
$$w_j^{(i)} = p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma).$$
This can be computed with Bayes' rule; suppressing parameters,
$$p(z^{(i)} = j|x^{(i)}) = \frac{p(x^{(i)}|z^{(i)} = j)p(z^{(i)} = j)}{\sum_\ell p(x^{(i)}|z^{(i)} = \ell)p(z^{(i)} = \ell)}.$$
- In the M-step, we treat these weights $w_j^{(i)}$ as soft GDA labels and update the parameters as
$$\phi_j = \frac{1}{m}\sum_i w_j^{(i)}, \quad \mu_j = \frac{\sum_i w_j^{(i)} x^{(i)}}{\sum_i w_j^{(i)}}, \quad \Sigma_j = \frac{\sum_i w_j^{(i)} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_i w_j^{(i)}}.$$
We can think of the first step as computing our "expectation" for the $z^{(i)}$, and the second step as carrying out a "maximization".
- Like k-means, the mixture of Gaussians model can get stuck in local optima. One particularly bad possibility is if one of the clusters centers on only one data point; in this case it will shrink indefinitely. This can be prevented by regularization.
- Probabilistic unsupervised algorithms can be tested for overfitting by using a training and testing set. We maximize the log-likelihood over the training set, and evaluate it on the testing set; if the two are dramatically different, the model is overfit.

Just as for k-means, we would like a more general way of viewing this algorithm which lets us establish convergence.

- The EM algorithm is the analogue of maximum likelihood estimation in the presence of latent variables. It alternates between computing expectations for the distribution of the latent variables given the parameters, and maximizing the likelihood over the parameters given those latent variable distributions.
- Jensen's inequality states that for a convex function f and random variable X,

  $$\mathbb{E}[f(X)] \geq f(\mathbb{E}[X])$$

  and if f is strictly convex, equality holds if and only if $X = \mathbb{E}[X]$ with probability one.
- Now, for each data point i, let $Q_i(z)$ be a probability distribution over the z's. Abbreviating all the parameters as θ, the log-likelihood is

  $$\ell(\theta) = \sum_i \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta) = \sum_i \log \sum_{z^{(i)}} Q_i(z^{(i)}) \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}.$$

  This has the form of the logarithm of the expectation value of $p(x^{(i)}, z^{(i)}; \theta)/Q_i(z^{(i)})$, distributed according to Q_i. Since the logarithm is concave, Jensen's inequality holds in reverse, giving

  $$\ell(\theta) \geq \sum_{i, z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \equiv J(Q, \theta).$$

  Thus, for each choice of Q_i, we have a lower bound on the log-likelihood. The quantity $J(Q, \theta)$ is also called the evidence lower bound (ELBO).
- The equality case of Jensen's inequality is achieved if the argument of the logarithm does not depend on $z^{(i)}$. This is achieved for the probability distribution

  $$Q_i(z^{(i)}) = p(z^{(i)} | x^{(i)}; \theta)$$

  which we abbreviate as Q_0. Then $\ell(\theta_0) = J(Q_0, \theta_0)$.
- Now the EM algorithm can be phrased as follows. If the current parameters are θ_0, then in the E-step we construct Q_0, and in the M-step we set the new value of θ to

  $$\theta_1 \equiv \underset{\theta}{\operatorname{argmax}}\, J(Q_0, \theta).$$

  In each pair of E and M-steps, the log-likelihood increases, because

  $$\ell(\theta_1) \geq J(Q_0, \theta_1) \geq J(Q_0, \theta_0) = \ell(\theta_0)$$

  thus establishing convergence. However, we are not guaranteed to converge to the global maximum of ℓ.
- Alternatively, we can choose to view $J(Q, \theta)$ as the cost function. In this case the analogy with k-means is very close: the E-step maximizes over Q, while the M-step maximizes over θ.

- The EM algorithm is also useful for semi-supervised learning, which is a supervised learning task where only some of the data points have labels. For the other data points, the labels are treated as latent variables.

Note. We can also think of $J(Q, \theta)$ as the KL divergence $D_{\mathrm{KL}}(Q \| p)$. This is useful because we can also write it in the forms

$$D_{\mathrm{KL}}(Q\|p) = \mathbb{E}_{z\sim Q}[\log p(x|z;\theta)] - D_{\mathrm{KL}}(Q\|p_z) = \log p(x) - D_{\mathrm{KL}}(Q\|p_{z|x}).$$

The first form tells us that the M-step maximizes J by optimizing the first term, i.e. by maximizing the log-likelihood. The second form tells us that the E-step maximizes J by optimizing the second term, i.e. by setting $Q = p_{z|x}$.

Example. Let's properly derive the mixture of Gaussians model, starting from this notation. The E-step is the same, if we identify $w_j^{(i)} = Q_i(z^{(i)} = j)$. In the M-step, we maximize

$$J(Q_0, \theta) = \sum_{ij} w_j^{(i)} \log \frac{\phi_j \, \exp(-(x^{(i)} - \mu_j)^T \Sigma_j^{-1} (x^{(i)} - \mu_j)/2)/(2\pi)^{n/2} |\Sigma_j|^{1/2}}{w_j^{(i)}}.$$

To set the μ_j, we note that

$$\frac{\partial J}{\partial \mu_\ell} = -\frac{1}{2} \frac{\partial}{\partial \mu_\ell} \sum_{ij} \left(w_j^{(i)} (x^{(i)} - \mu_j)^T \Sigma_j^{-1} (x^{(i)} - \mu_j) \right) = \sum_i w_\ell^{(i)} \Sigma_\ell^{-1} (x^{(i)} - \mu_\ell)$$

and setting this to zero recovers our previously stated result. To set the ϕ, note that

$$J(Q_0, \theta) \supset \sum_{ij} w_j^{(i)} \log \phi_j.$$

To maximize this, we use Lagrange multipliers to account for the constraint $\sum_j \phi_j = 1$, which again recovers the previous result. The maximization over Σ_j is a bit more complicated, but similar.

As another example of the EM algorithm, we consider factor analysis.

- Suppose we have n data points $x^{(i)} \in \mathbb{R}^d$ that we wish to fit a Gaussian to. The maximum likelihood estimators are simply

  $$\mu = \frac{1}{n} \sum_i x^{(i)}, \quad \Sigma = \frac{1}{n} \sum_i (x^{(i)} - \mu)(x^{(i)} - \mu)^T.$$

  However, in some cases we have $d > n$, which means Σ is singular. Geometrically, all the probability is concentrated on the hyperplane containing the data points.

- We could avoid singular Σ by restricting it, such as by making it diagonal or even proportional to the identity. But this prevents us from modeling correlations between the variables. We could account for this by letting some off-diagonal elements of Σ be nonzero, but we don't know ahead of time which – if we already knew, then we wouldn't have to fit Σ in the first place!

- As for the mixture of Gaussians model, the resolution is to introduce a latent random variable which parametrizes the correlated degrees of freedom. In factor analysis, we let these be a set of k-dimensional "factors",
$$z \sim \mathcal{N}(0, I), \quad z \in \mathbb{R}^k$$
where $k < n$, and model the data as generated by
$$x = \mu + \Lambda z + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \Psi)$$
where Ψ is diagonal with positive entries, and Λ is a $d \times k$ matrix.

- Using standard properties of Gaussians and the definition of the covariance,
$$\begin{pmatrix} z \\ x \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} 0 \\ \mu \end{pmatrix}, \begin{pmatrix} I & \Lambda^T \\ \Lambda & \Lambda\Lambda^T + \Psi \end{pmatrix} \right).$$
The marginal distribution of x is
$$x \sim \mathcal{N}(\mu, \Lambda\Lambda^T + \Psi)$$
and the log-likelihood is
$$\ell(\mu, \Lambda, \Psi) = \log \prod_{i=1}^{n} \frac{1}{(2\pi)^{d/2}|\Lambda\Lambda^T + \Psi|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu)^T (\Lambda\Lambda^T + \Psi)^{-1} (x^{(i)} - \mu) \right).$$

- To train the model, we maximize the log-likelihood. Now, the latent variable z does not appear in the likelihood. In fact, we could have skipped introducing it entirely, and just said that we are restricting Σ to the form $\Lambda\Lambda^T + \Psi$. But maximizing the log-likelihood is hard! We introduced z not because it's necessary to formulate the model, but because it allows us to use the EM algorithm, which is the best maximization method for the job.

- For brevity, we suppress the training label i. In the E-step, we set
$$Q(z) = p(z|x; \mu, \Lambda, \Psi) = \frac{1}{(2\pi)^{k/2}|\Sigma_{z|x}|^{1/2}} \exp\left(-\frac{1}{2}(z - \mu_{z|x})^T \Sigma_{z|x}^{-1} (z - \mu_{z|x}) \right).$$
Using our earlier results for conditionals of Gaussians,
$$\mu_{z|x} = \Lambda^T(\Lambda\Lambda^T + \Psi)^{-1}(x - \mu), \quad \Sigma_{z|x} = I - \Lambda^T(\Lambda\Lambda^T + \Psi)^{-1}\Lambda.$$

- In the M-step, we need to maximize an integral rather than a sum,
$$J(Q, \mu, \Lambda, \theta) = \int dz\, Q(z) \log \frac{p(x, z; \mu, \Lambda, \Psi)}{Q(z)}.$$
We'll show only the maximization with respect to Λ. The argument can be broken down as
$$\log p(x|z) + \log p(z) - \log Q(z)$$
and only the first term depends on Λ. Using our earlier results for conditionals of Gaussians, or just common sense,
$$\mu_{x|z} = \mu + \Lambda z, \quad \Sigma_{x|z} = \Psi.$$
It is thus equivalent to maximize
$$\mathbb{E}_{z \sim Q}[\log p(x|z)] = \mathbb{E}_{z \sim Q}\left[-\frac{1}{2}\log|\Psi| - \frac{n}{2}\log(2\pi) - \frac{1}{2}(x - \mu - \Lambda z)^T \Psi^{-1} (x - \mu - \Lambda z) \right].$$

- Only the last term depends on Λ. Differentiating with respect to Λ,

$$\frac{1}{2}\nabla_\Lambda\left[(x-\mu-\Lambda z)^T\Psi^{-1}(x-\mu-\Lambda z)\right] = -\Psi^{-1}(x-\mu-\Lambda z)z^T$$

  which means that at the maximum,

$$\mathbb{E}_{z\sim Q}\left[\Psi^{-1}(x-\mu-\Lambda z)z^T\right] = 0.$$

  Rearranging this gives

$$\Lambda = \left((x-\mu)\,\mathbb{E}_{z\sim Q}[z^T]\right)\left(\mathbb{E}_{z\sim Q}[zz^T]\right)^{-1}.$$

  This is very similar to the normal equations for least squares regression; the only difference is the introduction of the latent variable.

- Using the definition of the covariance, the expectations above are

$$\mathbb{E}_{z\sim Q}[z^T] = \mu_{z|x}^T, \quad \mathbb{E}_{z\sim Q}[zz^T] = \mu_{z|x}\mu_{z|x}^T + \Sigma_{z|x}$$

  Restoring the data labels, the M-step update for Λ is

$$\Lambda = \left(\sum_i (x^{(i)}-\mu)\mu_{z^{(i)}|x^{(i)}}^T\right)\left(\sum_i \mu_{z^{(i)}|x^{(i)}}\mu_{z^{(i)}|x^{(i)}}^T + \Sigma_{z^{(i)}|x^{(i)}}\right)^{-1}.$$

- One can easily show that

$$\mu = \frac{1}{n}\sum_i x^{(i)}.$$

  Since μ doesn't depend on the $z^{(i)}$, it can be calculated in the first step and need not be updated.

- With a bit more effort, we can show that $\Psi_{ii} = \Phi_{ii}$, where

$$\Phi = \frac{1}{n}\sum_i x^{(i)}x^{(i)^T} - x^{(i)}\mu_{z^{(i)}|x^{(i)}}^T\Lambda^T - \Lambda\mu_{z^{(i)}|x^{(i)}}x^{(i)^T} + \Lambda(\mu_{z^{(i)}|x^{(i)}}\mu_{z^{(i)}|x^{(i)}}^T + \Sigma_{z^{(i)}|x^{(i)}})\Lambda^T.$$

- Note that the M-step in principle depends on the *full* distribution of the $z^{(i)}$, rather than just the expected value $\mu_{z^{(i)}|x^{(i)}}$, as can be seen from the $\Sigma_{z^{(i)}|x^{(i)}}$ terms above. Some sources will incorrectly state that we only need the expectation value $\mathbb{E}[z^{(i)}]$ from the E-step. This is a misconception which arises because of the name of the E-step, and because this happens to be true in the simplest examples, such as the Gaussian mixture model.

5.3 Principal and Independent Component Analysis

Principal component analysis (PCA) identifies a low dimensional subspace in which the given feature vectors approximately lie. However, unlike factor analysis, it is not based on a probabilistic model; it corresponds to factor analysis just as K-means corresponds to the Gaussian mixture model. PCA is simpler than factor analysis, and can be performed in closed form, without the EM algorithm.

- As a first step, we should normalize the data by replacing the features with their z-scores. If all the features are known to be similar, e.g. if they are the pixel values in an image, then we only have to subtract their means.

- The variance of the data points is $\sum_i {x^{(i)}}^T x^{(i)}/n$. The first principal component is the direction that captures the most of this variance, i.e. the unit vector u that maximizes

  $$\frac{1}{n}\sum_i (u^T x^{(i)})^2 = u^T \left(\frac{1}{n}\sum_i x^{(i)} {x^{(i)}}^T\right) u \equiv u^T \Sigma u.$$

  Equivalently, it is the direction that minimizes the sums of the squared distances to the first principal axis. Note that it is essential to normalize the data, or else the features with larger values will always dominate.

- As we showed earlier, the critical points of this optimization problem are when u is an eigenvector of Σ. Thus the first principal component corresponds to the eigenvector of Σ with the largest eigenvalue, the second to the second largest, and so on.

- If there are d features, finding all the eigenvectors of Σ takes $O(d^3)$ time. However, finding the first principal component only takes $O(d^2)$ time by power iteration: if we start with any vector and repeatedly apply Σ and normalize the vector, then it will converge to the desired eigenvector. We can then restrict to the orthogonal subspace and repeat to find the second principal component, so finding k principal components takes $O(d^2 k)$ time.

- Finally, given the unit vectors u_i corresponding to the k chosen principal components, we can define the reduced feature vectors

  $$y^{(i)} = \begin{pmatrix} u_1^T x^{(i)} \\ \vdots \\ u_k^T x^{(i)} \end{pmatrix}.$$

- Another way to phrase the above results is that PCA finds the eigenvectors of $X^T X$, where X is the design matrix; this is equivalent to performing the singular value decomposition of X, with the principal components corresponding to the right singular vectors of X.

- PCA does not model how the data was generated; it is strictly for dimensionality reduction. For example, it can be used for data compression, or to reduce the dimensionality of input data by identifying redundant features. It can also be used by human begins to gain insight.

- PCA can also be used for noise reduction, by projecting data down into a lower-dimensional subspace. For example, the “eigenfaces” method uses PCA to identify a subspace of the set of images of faces. Face matching is performed using the Euclidean distance in this subspace.

Independent component analysis (ICA) can find a basis where the features are independent.

- A classic application of ICA is the cocktail party problem: suppose that we have a room with d microphones and d speakers. The speakers produce a vector of outputs s at every time, whose components are iid. The microphone readings are $x = As$ where A is the unknown mixing matrix. The purpose of ICA is to find the “unmixing” matrix $W = A^{-1}$ and hence recover the original speakers.

- ICA is practically used in EEG measurements, where it can separate out artifacts such as blinking and heartbeats, leaving clean data. Also, when ICA is applied to natural images, the independent components are essentially “edges”.

- Properly speaking, we aren't looking for a matrix W, but for an unordered normalized basis in which the features are independent. This is because the order of the speakers and their scalings are not defined. Thus, W is not exactly determined, though this isn't important in practice.
- Let w_j be the rows of W, so that
$$s_j = w_j^T x.$$
Then the ambiguities mentioned above correspond to permutations and scalings of the w_j.
- A more serious problem is if the speakers produce Gaussians. In this case, the joint distribution is $s \sim \mathcal{N}(0, cI)$, which is rotationally invariant, which means W is completely undetermined. This problem is unique to Gaussians, and is related to the central limit theorem.
- Let the distribution of each source s_j have probability density p_s, so the joint distribution is
$$p(s) = \prod_j p_s(s_j).$$
The distribution of $x = W^{-1}s$ is
$$p(x; W) = (\det W) \prod_j p_s(w_j^T x)$$
where the determinant is a Jacobian factor.
- Summing over the training examples and assuming they are independent, the log-likelihood is
$$\ell(W) = \sum_i \left(\log \det W + \sum_j \log p_s(w_j^T x^{(i)}) \right).$$
Taking the derivative with respect to W gives the stochastic gradient ascent update rule,
$$W := W + \alpha \left(\begin{pmatrix} p_s'(w_1^T x^{(i)})/p_s(w_1^T x^{(i)}) \\ \vdots \\ p_s'(w_d^T x^{(i)})/p_s(w_d^T x^{(i)}) \end{pmatrix} x^{(i)^T} + (W^T)^{-1} \right).$$
- To continue we must postulate a form for p_s, which can come from domain knowledge. In the absence of such knowledge, one option is to normalize the data and assume a logistic distribution, i.e. a sigmoid cdf, $p_s(s) = g'(s)$, which will generally perform decently. In this case, p'/p simplifies to $1 - 2g$.
- For time series data, nearby training examples are often not independent, but this is not an issue in practice. Counterintuitively, if we have such data, then randomly reshuffling the data points can accelerate the convergence of stochastic gradient descent.
- ICA also works perfectly well if there are more microphones than speakers, though the formalism above has to be slightly generalized to let A be non-square. When there are fewer microphones than speakers, it doesn't work; we would need to use additional domain knowledge to have a chance of separating all the speakers. For example, for one microphone and a male and female speaker, we can separate by frequency.