Introduction to Machine Learning - Andrew Ng (Study Guide)

December 10, 2019

1 Introductory concepts

Most problems in machine learning can be described by one of two categories of problems:

- **Supervised learning** is the class of problems where we are given a set of inputs and outputs, and want to be able to predict outputs given future inputs. The category of supervised learning problems can further be broken down into two subcategories:
  - **Regression** is when the outputs lie on a continuous scale.
  - **Classification** is when the outputs are one of a discrete set of values.
- **Unsupervised learning** is the class of problems where we are simply given data and asked to find structure in it. One common way of finding structure is by identifying clusters in the data.

In most cases, we refer to the data that we start with as our *training data*.

2 Gradient Descent

In supervised learning, we typically define a model with parameters θ that we want to use to predict outputs from inputs and an objective function (also called a cost function) $J(\theta)$ that we can use to evaluate how well our model is performing with respect to our given inputs X and outputs y (Obviously the function J depends on the values of X and y). A lower value of J indicates that the model is performing better.

One common way of finding the values θ that minimize J is the method of *gradient descent*. In gradient descent, we start with an arbitrary initial parameterization θ and repeatedly update according to the following rule:

$$\theta := \theta - \alpha \nabla J,$$

where := denotes that this is an assignment operation rather than a statement of equality. Usually gradient descent will be run for a fixed number of iterations, or until the change in J from one iteration to the next falls below some threshold.

The value of α is called the *learning rate*, and can have a significant impact on the convergence of gradient descent: excessively high values of α can prevent the algorithm from converging, while excessively low values can slow down the algorithm.

3 Linear Regression

Linear regression is a form of supervised learning in which we want to fit a linear relationship between our inputs and our outputs. More formally, in linear regression, we are given inputs

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} & x_1^{(i)} & x_2^{(i)} & \dots & x_n^{(i)} \end{bmatrix}^T$$

and outputs $y^{(i)}$, and we want to learn a set of parameters

$$\theta = [\theta_0 \ \theta_1 \ \theta_2 \ \ldots \ \theta_n]^T$$

such that

$$\theta^T x^{(i)} = \sum_{j=0}^{n} \theta_j x_j^{(i)} = h_\theta\left(x^{(i)}\right)$$

predicts $y^{(i)}$ as well as possible. We measure the performance of our model by the objective function

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 = \frac{1}{2m}(X\theta - y)^T(X\theta - y),$$

where m is the number of training examples we have,

$$X = \begin{bmatrix} — & x^{(1)T} & — \\ — & x^{(2)T} & — \\ & \vdots & \\ — & x^{(n)T} & — \end{bmatrix}$$

is a matrix of all of our training inputs, and

$$y = \left[y^{(1)} \ y^{(2)} \ \ldots \ y^{(m)}\right]^T$$

is a vector of all of our training outputs. By convention, we typically define $x_0^{(i)} = 1$ for all i. We have

$$\nabla J = \frac{\partial J}{\partial \theta} = \frac{1}{2m} \frac{\partial}{\partial \theta} \left((X\theta - y)^T(X\theta - y)\right) = \frac{1}{m}(X\theta - y)^T X.$$

However, because θ is usually written as a column vector, it is also more convenient to write ∇J as a column vector, so we typically write

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

Thus we can perform gradient descent on θ using the assignment

$$\theta := \theta - \frac{\alpha}{m} X^T(X\theta - y).$$

3.1 Feature Scaling

When using gradient descent in practice, it is easier for the algorithm to converge if the values of the features are similar. Thus oftentimes, we will adjust the values in X:

$$x_j^{(i)} := \frac{x_j^{(i)} - \mu_j}{\sigma_j}, \quad 1 \le i \le m, \quad 1 \le k \le n,$$

where

$$\mu_j = \frac{1}{m} \sum_{k=1}^{m} x_j^{(k)}, \quad \sigma_j = \sqrt{\frac{1}{m-1} \sum_{k=1}^{m} (x_j^{(k)} - \mu_j)^2},$$

i.e. μ_j is the mean of all values of the j-th feature, and σ_j is their standard deviation.

3.2 Learning Rate

The behavior of gradient descent also depends on the choice of α, the learning rate. If α is too small, then gradient descent can be slow to converge, while if α is too large, gradient descent may fail to converge at all (or maybe also be slow to converge). A good way to evaluate whether a given value of α is working well is to plot the value of $J(\theta)$ against the number of iterations and check that it is converging quickly. One way to select a value of α is to try a few different values (in rough geometric sequence) and run gradient descent with them for a few iterations; then we can select the value that gives the most favorable plot of $J(\theta)$ against number of iterations (or to be conservative, we can select a slightly smaller value).

After selecting a value of α, we need a way to determine whether gradient descent has converged. One way is to simply eyeball the plot of $J(\theta)$ against iteration number; this generally works pretty well if the plot is easily obtainable, as the graph should appear to flatten out after some point. Another way is to declare that gradient descent has converged if $J(\theta)$ changes by less than some predefined threshold between iterations.

3.3 Feature Addition

Sometimes the behavior of our target value cannot be captured by a linear combination of our features. In this case, we can define additional features that are nonlinear functions of existing features; for example, if we have two features x_1 and x_2, we may choose to define additional features $x_3 = x_1 x_2$ and $x_4 = \sqrt{x_2}$. This allows us to capture more sophisticated nonlinear relationships in our data. When doing this, however, it is important that we perform feature scaling so that gradient descent is able to perform well.

3.4 Normal Equation

In linear regression, we are attempting to find the value of θ that minimizes the value of our objective function

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

It can be shown that $J(\theta)$ is convex; therefore it has only one local minimum, and that local minimum is also a global minimum. Thus to find the value of θ that minimizes $J(\theta)$, it suffices to find the value of θ such that the gradient of $J(\theta)$ is zero. We have

$$\begin{aligned} \frac{\partial J}{\partial \theta} &= 0 \\ \Rightarrow \frac{1}{m} X^T(X\theta - y) &= 0 \\ \Rightarrow X^T X\theta &= X^T y \\ \Rightarrow \theta &= (X^T X)^{-1} X^T y. \end{aligned}$$

When taking the inverse of $X^T X$, we use the Moore-Penrose pseudoinverse rather than the "true" inverse as the former always exists (and is unique) while the latter does not if $X^T X$ is singular.

While the normal function may at a glance seem superior to gradient descent (it's very simple to implement, there's no need to choose hyperparameters like the learning rate, feature scaling is unnecessary), it's actually very slow if we have a large number n of features, as $X^T X$ has dimensions $n \times n$ and matrix inversion is a very expensive computation.

4 Logistic Regression

Logistic regression is a method that is commonly used to solve classification problems. The name here is a bit misleading, but makes more sense in context. In classification problems, we are typically given a set of inputs

$$x^{(i)} = \left[x_0^{(i)}\ x_1^{(i)}\ x_2^{(i)}\ \dots\ x_n^{(i)}\right]^T$$

and outputs $y^{(i)} \in \{0, 1, \dots, s\}$ for some $s \in \mathbb{N}$ and wish to predict the value of future outputs y from future inputs x. Unless otherwise specified, we will assume that $s = 1$.

A common way to make this prediction is to fit some θ-parameterized hypothesis function $h_\theta(x)$ (this is where the "regression" part comes in) with $0 \le h_\theta(x) \le 1$ such that the output of h_θ approximates our values y as well as possible and, for future inputs x, predict $y = 1$ if $h_\theta(x) \ge b$ for some boundary value b and $y = 0$ otherwise.

4.1 Logistic Regression Hypothesis and Objective Functions

Logistic regression makes use of the sigmoid function:

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

Note that

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

In logistic regression, we want to learn a set of parameters $\theta = [\theta_1, \theta_2, \dots, \theta_n]^T$ such that

$$\sigma(\theta^T x^{(i)}) = \sigma\left(\sum_{j=1}^{n} \theta_j x_j^{(i)}\right) = h_\theta(x^{(i)})$$

predicts $y^{(i)}$ as well as possible. We measure the performance of our model with the objective function

$$\begin{aligned} J(\theta) &= \frac{1}{m}\sum_{i=1}^{m}\left(-y^{(i)}\log(h_\theta(x^{(i)})) - (1 - y^{(i)})(\log(1 - h_\theta(x^{(i)})))\right) \\ &= \frac{1}{m}\left(-y^T \log(\sigma(X\theta)) + (1 - y)^T(-\log(1 - \sigma(X\theta))\right). \end{aligned}$$

We have

$$\begin{aligned} \nabla J &= \frac{\partial J}{\partial \theta} \\ &= \frac{1}{m}\left(-y^T \frac{\partial}{\partial \theta}\ \log(\sigma(X\theta)) - (1 - y)^T \frac{\partial}{\partial \theta}\ \log(1 - \sigma(X\theta))\right). \end{aligned}$$

Then,

$$\begin{aligned} \frac{\partial}{\partial \theta}\ \log(\sigma(X\theta)) &= \left(\frac{1}{\sigma(X\theta)}\right)^d \frac{\partial}{\partial \theta}\sigma(X\theta) \\ &= \left(\frac{1}{\sigma(X\theta)}\right)^d (\sigma(X\theta))^d(1 - \sigma(X\theta))^d X \\ &= (1 - \sigma(X\theta))^d X, \end{aligned}$$

where v^d for a vector $v = [v_1, v_2, \dots, v_r]^T$ denotes the diagonal matrix

$$\begin{bmatrix} v_1 & 0 & \cdots & 0 \\ 0 & v_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & v_r \end{bmatrix}.$$

Similarly,

$$\begin{aligned}
\frac{\partial}{\partial\theta}\ \log(1-\sigma(X\theta)) &= \left(\frac{1}{1-\sigma(X\theta)}\right)^d \frac{\partial}{\partial\theta}\ (1-\sigma(X\theta)) \\
&= -\left(\frac{1}{1-\sigma(X\theta)}\right)^d \frac{\partial}{\partial\theta}\ \sigma(X\theta) \\
&= -\left(\frac{1}{1-\sigma(X\theta)}\right)^d (\sigma(X\theta))^d(1-\sigma(X\theta))^d X \\
&= -\sigma(X\theta)^d X.
\end{aligned}$$

This yields

$$\begin{aligned}
\nabla J &= \frac{1}{m}\left(-y^T \frac{\partial}{\partial\theta}\ \log(\sigma(X\theta)) - (1-y)^T \frac{\partial}{\partial\theta}\ \log(1-\sigma(X\theta))\right) \\
&= \frac{1}{m}\left(-y^T(1-\sigma(X\theta))^d X - (1-y)^T(-\sigma(X\theta)^d X)\right) \\
&= \frac{1}{m}\left(-y^T(1-\sigma(X\theta))^d + (1-y)^T(\sigma(X\theta)^d)\right) X \\
&= \frac{1}{m}\left(-y^T + y^T\sigma(X\theta)^d + \sigma(X\theta) - y^T\sigma(X\theta)^d\right) X \\
&= \frac{1}{m}(\sigma(X\theta)^T - y^T)X.
\end{aligned}$$

The same points made in the last section about learning rate and feature addition apply to logistic regression as well.

4.2 Multiclass Classification

The previous section details how to use logistic regression to perform two-class classification. To perform K-class classification with $K > 2$, the most common way is to simply train K classifiers that predict whether an example displaying a set of inputs x belongs to class k or not, for each $k, 1 \le k \le K$. Then to classify a new example, we simply select the class of the classifier that returns the highest probability. (Note that this can be applied to other classification methods besides logistic regression as well.)

5 Regularization

A common problem in many machine learning applications is the problem of overfitting, where a trained model fits the training data very closely, but misses the overall trend in the data. This can occur, for (a rather contrived) example, if we have 8 training examples, each with a single numeric input and a single numeric output, that follow a rough linear relationship, but we fit a degree-7 polynomial to the data. This results in a perfect fit to the training data, but new data points that lie on the line are unlikely to be close to the model. On the other end of the spectrum, underfitting is also possible, where the model fails to respond to the training data or predicts something too simple to fit the training data well.

Oftentimes, we refer to models that overfit as models that have high *variance* (one way to think about this intuitively is that the models change very drastically when new training data is added), and models that underfit as models that have high *bias* (one way to think about this intuitively is to think about these models as having strong predefined notions of what the data should look like, and not changing their behavior when contrasting data is presented).

Regularization is one common method of dealing with overfitting. The key in regularization is to add a regularization term to our objective function, equal to

$$\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 = \frac{\lambda}{2m}\theta_r^T\theta_r,$$

where $\theta_r = [0, \theta_1, \theta_2, \ldots, \theta_n]^T$ is equal to θ except that θ_0 is replaced with 0. Here the parameter λ is called the *regularization parameter*. Regularization causes the model to prefer lower numeric values in θ; in practice, this causes the model to be simpler, as more complex models that overfit the data tend to have high parameter values. (This is not easy to prove, but makes some sense intuitively.) As would make sense, low values of λ tend to increase variance and decrease bias, while high values of λ tend to increase bias and decrease variance.

5.1 Regularization: Linear Regression

In regularized linear regression, our objective function becomes

$$J(\theta) = \frac{1}{2m}(X\theta - y)^T(X\theta - y) + \frac{\lambda}{2m}\theta_r^T\theta_r,$$

and we have

$$\nabla J = \frac{1}{m}(X\theta - y)^T X + \frac{\lambda}{m}\theta_r^T.$$

To calculate the normal equation solution to regularized linear regression, first let L be the matrix that is identical to I, the identity matrix, except that the top-left entry in L is equal to 0 instead of 1. Then we see that

$$\begin{aligned}
\frac{\partial J}{\partial \theta} &= 0 \\
\Rightarrow \frac{1}{m}X^T(X\theta - y) + \frac{\lambda}{m}\theta_r &= 0 \\
\Rightarrow X^T(X\theta - y) + \lambda L\theta &= 0 \\
\Rightarrow (X^TX + \lambda L)\theta &= X^Ty \\
\Rightarrow \theta &= (X^TX + \lambda L)^{-1}X^Ty.
\end{aligned}$$

5.2 Regularization: Logistic Regression

In regularized logistic regression, our objective function becomes

$$J(\theta) = \frac{1}{m}\left(-y^T\log(\sigma(X\theta)) + (1-y)^T(-\log(1-\sigma(X\theta))\right) + \frac{\lambda}{2m}\theta_r^T\theta_r,$$

and we have

$$\nabla J = \frac{1}{m}(\sigma(X\theta)^T - y^T)X + \frac{\lambda}{m}\theta_r^T.$$

6 Neural networks

Neural networks (often called neural nets) are a machine learning technique that can be used to learn complex nonlinear hypotheses between input variables. Linear regression and logistic regression only really work with linear models; linear regression directly learns a linear combination of the inputs, while logistic regression learns the sigmoid of a linear combination of the inputs. If we wanted our model to respond to anything besides a linear combination of our inputs, we had to implement nonlinear features manually, e.g. by providing a new feature that is equal to the product of some two other features. Neural nets can learn such nonlinear relationships independently, without us having to manually provide features.

6.1 Mathematical representation

A neuron in a neural network has an input vector x, a parameter vector θ, and an activation function f, and produces an output (also called an activation) $a = f(\theta^T x)$. It is usually implied that the input and parameter vectors contain bias terms $x_0 = 1$ and θ_0. A common activation function f is the sigmoid function

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

Given this, it is perhaps obvious that neural nets are typically used to solve classification problems (although it is possible to use them to solve regression problems); unless otherwise stated, we will assume that any given neural net has a sigmoid activation function (also called a logistic activation function) and is being used to solve a classification problem.

The neurons in a neural net are typically arranged in layers, with the first layer being called the input layer, the last layer being called the output layer, and any layers in between being called the hidden layers. The input layer is special in that its neurons have no inputs, and their outputs (i.e. activations) are simply the values of the system inputs, i.e. the feature values. For the simple case where every neuron in a layer depends only on activation values of the previous layer and take all of those activation values as inputs (which is the only case we will cover in this document), if we denote the activation of the j-th neuron in layer i as $a_j^{(i)}$, its parameter vector as $\theta_j^{(i-1)}$, and the number of neurons in layer l as s_l, then we have

$$a_j^{(i)} = \sigma\left(\sum_{k=0}^{s_{i-1}} \theta_{jk}^{(i-1)} a_k^{(i-1)}\right),$$

for all j, i.e.

$$a^{(i)} = \sigma\left(\Theta^{(i-1)} a^{(i-1)}\right),$$

where

$$a^{(i)} = [a_0^{(i)}, a_1^{(i)}, \ldots, a_{s_i}^{(i)}]^T,$$
$$\Theta^{(i)} = \begin{bmatrix} — & \theta_0^{(i)T} & — \\ — & \theta_1^{(i)T} & — \\ & \vdots & \\ — & \theta_{s_{i+1}}^{(i)T} & — \end{bmatrix},$$

and it is understood that $a_0^{(i)} = 1$ for all i. It is also common to denote the inverse sigmoid of $a_j^{(i)}$ as $z_j^{(i)}$, i.e. $a_j^{(i)} = \sigma(z_j^{(i)})$. This process of calculating the values of the next layer from the previous layer is called *forward propagation.*

6.2 Binary Versus Multiclass Classification

For binary classification, a neural net needs only a single output neuron that outputs its predicted probability $h(x)$ that an input presenting features x belongs to the class $y = 1$. For K-class classification, the neural net needs K output neurons, where the k-th output neuron indicates the neural net's predicted probability that an input presenting features x belongs to class k.

6.3 Gradient Descent with Neural Networks

Suppose we have a neural network with L layers, s_l neurons in each layer, and K classes. Let Θ represent its entire set of parameters, and let $h_\Theta(x)_k$ denote the k-th output (i.e. the probability predicted by the neural net that the input belongs to the k-th class). Let $y_k^{(i)} = 1$ if the i-th training example belongs to class k, and 0 otherwise. Then our objective function is

$$\begin{aligned} J(\Theta) &= -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left(y_k^{(i)}\log\left(h_\Theta(x^{(i)})_k\right) + (1-y_k^{(i)})\log\left(1-h_\Theta(x^{(i)})_k\right)\right) + \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{s_{l+1}}\sum_{j=1}^{s_l}(\Theta_{ij}^{(l)})^2 \\ &= -\frac{1}{m}\sum_{k=1}^{K}\left(y_k^T\log(h_\Theta(X)_k) + (1-y_k)^T\log(1-h_\Theta(X)_k)\right) + \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{s_{l+1}}\sum_{j=1}^{s_l}(\Theta_{ij}^{(l)})^2 \end{aligned}$$

To calculate the gradient of this objective function, the *backpropagation* algorithm is typically used. The proof of correctness of backpropagation is pretty complex, so we will not spell it out here. To perform backpropagation, begin with values $\Delta_{ij}^{(l)}$ set to 0 for all i, j, l. Then for each training example $t = 1, \dots, m$:

1. Set $a^{(1)} = x^{(t)}$, and perform forward propagation to calculate $a^{(l)}$ for each $1 \le l \le L$.
2. Compute $\delta^{(L)} = a^{(L)} - y^{(t)}$.
3. For each $L-1 \ge l \ge 1$, compute $\delta^{(l)} = \sigma'(z^{(l)})^d(\Theta^{(l)T}\delta^{(l+1)}) = a^{(l)d}(1-a^{(l)})^d(\Theta^{(l)T}\delta^{(l+1)})$.
4. Perform $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$.

Then we have

$$\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = \begin{cases} \frac{1}{m}\left(\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)}\right), & \text{if } j \neq 0, \\ \frac{1}{m}\Delta_{ij}^{(l)}, & \text{if } j = 0. \end{cases}$$

6.4 Implementation details

When implementing backpropagation (which you probably shouldn't do in practice; libraries exist that already implement it far more efficiently than you can), there are a few things to keep in mind.

- To be able to use the built-in optimization functions in most languages, we need to unroll the parameters that we are optimizing into a single long vector. This means that the matrices $\Theta^{(l)}$ all need to be reshaped into a vector and then concatenated.
- For backpropagation to work correctly, we need to initialize the values in our matrices Θ to random small values in some interval $[-\epsilon, \epsilon]$. Initializing to zero will cause backpropagation to not update any parameters (except those in the last layer).
- It can also help to check symbolically computed gradients against numerically computed gradients:

$$f'(x) \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}.$$

7 Hyperparameter tuning and model evaluation

In any machine learning model, we have parameters, which we have usually denoted as θ or Θ. However, some machine learning models also have hyperparameters, which specify some underlying property of the model itself that isn't fit to the training data. An example of a hyperparameter is the regularization parameter λ in regularized linear or logistic regression. (Formally, a hyperparameter is a parameter of a prior distribution; it's helpful to understand what this means if you're thinking about machine learning from a statistical perspective.) However, it's not always clear how to tune a hyperparameter; for example, we will always minimize our objective function in regularized linear regression by setting λ to zero, even though this is clearly not always the correct approach.

One way to approach hyperparameter tuning is to randomly split your training data into two sets of data (with some pre-set proportion): the training set and the cross-validation (CV) set (a common proportion is 70% training and 30% CV). To select an optimal hyperparameter value (we will use the regularization parameter λ here for sake of example), we can train models θ for many different values of λ, compute the objective function that each model θ yields on the CV set, and then select the model (and thus the value of λ) that corresponds to the lowest CV value. When computing CV cost, however, it is important to compute simply the raw objective function (so in the case of optimizing λ, we should compute the unregularized CV cost), as the objective here is to find the model that fits new data most closely, regardless of how it was trained (i.e., in our example, regardless of the value of λ).

After this, supposing we want to report the accuracy of our model, we should report the value of the objective function that our model yields on a dataset. However, we should not use either the training set or the CV set here, as our model's parameters and hyperparameters have been optimized to give the lowest possible values on those particular sets. We should thus use a dataset that is independent of those two, commonly called a test set. One way to obtain a test set is to, instead of splitting our training data into only two sets in the beginning (training vs CV), split the data into three sets (training, CV, and test; a common proportion is 60% training and 20% for CV/test).

7.1 Bias or Variance?

When a model is not working well, it's good to evaluate whether the model is overly biased, or overly variant. This is because things that will help an overly biased model perform better will not help an overly variant model, and vice versa. Things that can help a model with high bias are:

- Add more features
- Decrease the regularization parameter λ
- Add more layers of neurons (in a neural net)

Things that can help a model with high variance are:

- Reduce the feature space
- Increase the regularization parameter λ
- Get more training data
- Remove layers of neurons (in a neural net)

We can diagnose whether the problem is high bias or high variance by looking at the training and cross-validation errors (unregularized): if the issue is high bias (underfitting), both training and CV error will be high, and if the issue is high variance (overfitting), training error will be much lower than CV error. When tuning hyperparameters of our model (number of features, regularization parameter, number of neurons in a neural network), we can usually approach the issue by minimizing CV error. However, when deciding whether or not more training data would be helpful, we can plot learning curves, which plot training and CV error as a function of the amount of training data used (so we can train our model using only one training example, plot training/CV error, then train with two examples, then three, etc.). If training and CV error approach each other closely early on, then a high error is likely to be a bias issue and more data will not help; if training and CV approach each other slowly and are still far apart even when using the whole training set, then additional data will probably be useful.

7.2 Error analysis

One thing that can be helpful in solving machine learning problems is to train a model on a data set, evaluate it against a cross-validation set, and manually look at some of the examples where it made the largest errors. Based on this, we can get some ideas about what things might be useful to work on next, implement those ideas, and see if our algorithm performed any better.

7.3 A new evaluation metric

When evaluating the performance of our algorithm, it is important to have a single real-number metric to measure it by; this gives us an unambiguous way to measure whether the algorithm is performing better or not. For regression problems, we might use the squared error (i.e. the usual error function); for classification problems, we might use the classification accuracy rate.

7.3.1 Skewed data

One case where the classification accuracy rate may not be a good metric for classifier performance is if we are building a classifier for a rare event. Suppose, for example, that we are trying to predict whether or not a patient has cancer based on an x-ray, and that only 0.5% of patients actually have cancer. Then if we had an extremely naive classifier that always predicted "no cancer", we'd have 99.5% classification accuracy, which might actually perform better than a lot of sophisticated models; however, this clearly doesn't indicate that our model is better.

In these cases, it is helpful to introduce the concepts of *precision* and *recall.* To define precision and recall, first let us define T_{ab} as the number of examples for which b was the correct classification, and our model predicted a, where $a, b \in \{0, 1\}$. Then, defining p as precision and r as recall, we have

$$p = \frac{T_{11}}{T_{11} + T_{10}}$$
$$r = \frac{T_{11}}{T_{11} + T_{01}}.$$

We can think about precision and recall intuitively as:

- Precision: out of all the things we classified positively, how many were actually positive? (Are we classifying too many things positively?)
- Recall: out of all the things that are positive, how many did we classify positively? (Did we get everything that's actually positive?)

This introduces a problem however - now we have two metrics instead of one, and many things that boost one metric almost necessarily hurt the other (e.g. raising or lowering the positivity threshold on a classifier). To combat this issue, we often use the F_1 score instead:

$$F_1 = \frac{2pr}{p + r}.$$

Note that the F_1 score will penalize both a low precision and a low recall heavily, even if the other metric is perfect.

7.4 When are more examples useful?

Oftentimes, machine learning algorithms can perform pretty much arbitrarily well if given enough training examples, but it may not be obvious that a lack of examples is the issue. A good way to tell if collecting more examples might help (as opposed to e.g. developing new features) is to ask: Given the data being provided to the algorithm, could a human expert predict the correct answer? If yes, then more training examples will likely be helpful; if no, then more feature engineering may be in order.

8 Support Vector Machines

Support vector machines (SVMs) are a technique commonly used for classification. To define them mathematically, we must first define a function β:

$$\beta(z) = \max(0, bx + b)$$

for some constant b. Note that this function is 0 for $b \leq -1$ and positive for $b \geq -1$. Then the objective function for a SVM is

$$J(\theta) = C \sum_{i=1}^{m} \left(y^{(i)} \beta(-\theta^T x^{(i)}) + (1 - y^{(i)}) \beta(-\theta^T x^{(i)}) \right) + \theta_r^T \theta_r.$$

Note that the regularization parameter is encapsulated in C here, which acts like $1/\lambda$, and that there is no $1/m$ factor. Furthermore, SVMs do not predict probabilities - they simply predict $h_\theta(x) = 0$ or 1 depending on whether $\theta^T x$ is negative or positive.

When presented with data to classify, SVMs will usually settle on a decision boundary with the property that the data is maximally distant from the decision boundary. This is for two reasons:

1. The SVM is incentivized to fit $\theta^T x < -1$ for examples where $y = 0$, and $\theta^T x > 1$ for examples where $y = 1$ (by the β function, which is a constant 0 for $z \geq 1$), and

2. The SVM is incentivized by the regularization term to find the θ that comes closest to doing this with the smallest possible magnitude.

Intuitively, these two items are significant because the vector θ is the normal vector of the decision boundary, and $\theta^T x$ measures the distance of x from the decision boundary (multiplied by the magnitude of θ). Then a decision boundary where the data is maximally distant from it allows us to achieve the first condition with a parameter vector θ of minimial magnitude.

8.1 Kernels

A common technique with SVMs is to define a kernel function f, that measures "similarity" between two vectors (many kernel functions are possible), with landmarks $l^{(1)}, l^{(2)}, \ldots, l^{(v)}$ and parameterize θ with respect to the vector

$$\left[f(x, l^{(1)}), f(x, l^{(2)}), \ldots, f(x, l^{(3)}) \right]$$

instead of x. A common kernel function f is the Gaussian kernel, which is defined as

$$f(x, l) = \exp\left(-\frac{||x - l||^2}{2\sigma^2} \right)$$

for some constant σ.

When using kernels in practice, a common way to define landmarks is simply to define a landmark at each training input vector, i.e. $l^{(i)} = x^{(i)}$ for $i = 1, 2, \ldots, m$.

8.1.1 Bias and variance with Gaussian-kernel SVMs

Typically, when using an SVM with a Gaussian kernel, small values of C and large values of σ^2 lead to a model with high bias and low variance, while large values of C and small values of σ^2 lead to the opposite.

9 K-means

K-means is an unsupervised learning algorithm that is used for clustering, which refers to the problem of grouping unlabeled data into clusters. Mathematically, K-means takes a set of unlabled training data $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ and finds K centroids $\mu_1, \mu_2, \ldots, \mu_K$ such that

$$J(c^{(1)}, c^{(2)}, \ldots, c^{(m)}, \mu_1, \mu_2, \ldots, \mu_K) = \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

is minimized, where $c^{(i)} \in \{1, 2, \ldots, K\}$ corresponds to a centroid index. Intuitively, minimizing this function attempts to find the most "sensible" way to split the data into K groups. The usual algorithm for accomplishing this minimization is as follows:

1. Begin with K randomly initialized centroids $\mu_1, \mu_2, \ldots, \mu_K$.
2. Repeat until convergence, i.e. until further repetitions do not change any of the $c^{(i)}$ or the μ_k:
   (a) For each $1 \leq i \leq m$, set $c^{(i)}$ to be the index of the centroid μ_k that minimizes $||x^{(i)} - \mu_k||^2$. This minimizes J with respect to the $c^{(i)}$ while holding the μ_k constant.
   (b) For each $1 \leq k \leq K$, set μ_K to the average of the $x^{(i)}$ satisfying $c^{(i)} = k$. This minimizes J with respect to the μ_k while holding the $c^{(i)}$ constant.

Typically, the randomly initialized μ_k are just K items randomly selected from the training data. Because the algorithm can sometimes get stuck in local optima that are not global optima, the algorithm is usually run multiple times with different initializations, and the result giving the lowest value of J is selected.

One thing that is not obvious when using K-means is how to choose the number K of clusters. This problem is usually beyond the scope of the mathematics, and the number K should typically be chosen based on the larger problem that you are trying to solve (e.g. if you are trying to do market segmentation, it is helpful to find out the number of segments that maximizes sales versus effort; the algorithm itself isn't going to be able to tell you this).

10 Principal Component Analysis

Principal Component Analysis, or PCA, is an unsupervised learning algorithm that is used to perform dimensionality reduction. Dimensionality reduction is exactly what it sounds like: it refers to finding a projection of an n-dimensional data set X onto a k-dimensional ($k < n$) subspace while retaining most of the information in X, resulting in a new k-dimensional data set Z.

The foundation of PCA is a mathematical technique called the eigendecomposition of a normal positive semidefinite matrix. This refers to factoring a normal positive semidefinite matrix M into the form $Q\Lambda Q^{-1}$, where each column of Q is a distinct eigenvector of M and Λ is a diagonal matrix such that the k-th entry is the eigenvalue corresponding to the k-the column of Q. Eigendecomposition can also be generalized to singular value decomposition (SVD) for matrices that are not normal positive semidefinite. Most implementations of eigendecomposition and SVD are such that the matrix Λ is sorted with $\Lambda_{ii} \geq \Lambda_{jj}$ for $i < j$.

The PCA algorithm for reducing from n dimensions to k dimensions is as follows:

1. Given a data set X_r, first perform feature scaling and mean normalization to get a normalized data set X.
2. Compute the covariance matrix Σ of X:
$$\Sigma = \frac{1}{m} X^T X.$$
3. Compute the eigendecomposition $\Sigma = Q\Lambda Q^{-1}$, and ensure that Λ is sorted with $\Lambda_{ii} \geq \Lambda_{jj}$ for $i < j$.
4. Let U be the matrix comprised of the first k columns of Q. Then our reduced-dimension data set is $Z = XU$.

Often it is informative to reconstruct the approximation of X represented by Z; this can be done with $X' = ZU^T$. One use of this reconstruction is to measure the amount of information retained after the dimensionality reduction; this is usually done using the metric

$$v = 1 - \frac{\frac{1}{m}\sum_{i=1}^{m} ||x^{(i)} - x'^{(i)}||^2}{\frac{1}{m}\sum_{i=1}^{m} ||x^{(i)}||^2},$$

where v is typically called the variance retained. Oftentimes we want to select the smallest k such that $v \geq v_0$ for some benchmark v_0; common values of v_0 are 0.95 and 0.99. Luckily, this is easy to do; it turns out that

$$v = \sum_{j=1}^{k} \Lambda_{jj},$$

where k is the number of dimensions that we reduced to.

11 Anomaly detection

Anomaly detection is a pretty conceptually simple problem: given a data set $\{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$ and a new example x, is x normal or anomalous?

This is typically done by fitting a probability distribution p to the data set and then evaluating whether $p(x) > \epsilon$ for some threshold ϵ. If $p(x) > \epsilon$, then x is normal; otherwise, it is anomalous. One way to fit this probability distribution is to fit a Gaussian distribution to each of the features x_i, so that

$$p(x) = \prod_{j=1}^{n} p_j(x),$$

where

$$p_j(x) = \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(\frac{-(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$

with

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}, \quad \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j - \mu_j)^2.$$

Another way is to fit a multivariate Gaussian distribution to the dataset as a whole, so that

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

where

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

Some pros and cons with using separate Gaussians versus the multivariate Gaussian are:

- The multivariate Gaussian is more computationally expensive to generate.
- To capture correlations with separate single-variable Gaussians, you would need to design new features, e.g. the ratio between two existing features. The multivariate Gaussian captures these correlations automatically.
- The multivaraite Gaussian requires there to be more training examples than features; otherwise Σ will be noninvertible.

A typical way to evaluate an anomaly detection model (especially the parameter ϵ) is to gather a set of both known normal and known anomalous examples, find the examples that the model classifies as anomalous, and then compute the F_1 score; we can also use a training and cross-validation set to optimize ϵ. This makes anomaly detection seem a lot like classification on a skewed data set, but typically we use a classification algorithm when we have enough positive examples to know what positive examples generally look like; anomaly detection is more suitable when we have very few positive examples and do not have a good idea of what future positive examples will look like. Furthermore, in a training set for anomaly detection, we do not include anomalous examples, as we are trying to model a probability distribution of normal examples.

12 Recommender systems

For the problem of recommender systems, it helps to have an example for the sake of better intuition. Suppose that we are Netflix, and that we have n_m movies and n_u users on our platform. Each user has watched/rated some subset of the movies on our platform, and we want to predict how much they would enjoy the other movies on our platform (more formally, how they would rate those movies). Let $r(i,j)$ be 1 if user j has rated movie i, and 0 otherwise, and let $y^{(i,j)}$ be the rating assigned by user j to movie i (if it exists). Furthermore, let $x^{(i)}$ be the feature vector for movie i (more on this later), and $\theta^{(j)}$ be the parameter vector for user j. You can imagine this as a linear regression problem, where our model predicts that user j will give movie i a rating of $\theta^{(j)T}x^{(i)}$.

One way to approach this problem is to try to devise features x and acquire ground-truth values for all of them, and simply train a linear regression model for each user. Formally, this can be expressed as trying to minimize the objective function

$$J(\Theta) = \frac{1}{2}||X\Theta^T - Y||^2 + \frac{\lambda}{2}||\Theta||^2,$$

where

$$\Theta = \begin{bmatrix} — & \theta^{(1)T} & — \\ — & \theta^{(2)T} & — \\ & \vdots & \\ — & \theta^{(n_u)T} & — \end{bmatrix}$$

and $||A||^2$ represents the sum of the squares of the entries of the matrix A. However, this is pretty difficult to do, as these features may not be easily measured (e.g. a feature might be how action-packed a movie is; it's difficult to represent this with a number). A more feasible approach is to use a technique called collaborative filtering, where we learn the features x and the parameters θ simultaneously. In collaborative filtering, we use the objective function

$$J(\Theta) = \frac{1}{2}||X\Theta^T - Y||^2 + \frac{\lambda}{2}||\Theta||^2 + \frac{\lambda}{2}||X||^2.$$

The gradient of this function is not difficult to calculate but is pretty difficult to notate, so we will not go over the computation here.

A typical implementation of collaborative filtering will use random initialization (as with neural networks), and will also perform gradient descent with respect to X and Θ separately. (So it performs one step of gradient descent with respect to X, then one step with respect to Θ, then one with respect to X, etc.) It can also help to use mean normalization on Y to get better predictions for users who have not rated movies; without mean normalization, we would predict these users to give every movie a rating of zero, while with mean normalization, we would predict these users to give every movie its average rating (which is a more useful prediction). Lastly, when performing collaborative filtering, we typically get rid of bias terms, as the algorithm is free to learn its own features; if the data would be better fit with a bias term, the algorithm will learn such a feature on its own.

13 Gradient descent, revisited

Consider the computation of the gradient when using gradient descent to perform linear regression:

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

Another way to write this is

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}.$$

We use this in gradient descent to update the value of θ:

$$\theta_j := \theta_j - \alpha \frac{\partial J}{\partial \theta_j} = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}.$$

However, if m is very large (say, 10^8), this computation can be very slow even if we have a good vectorized implementation, as we have to sum over m terms. However, note that each summand is independent of the next with respect to the training data; we could write

$$\frac{\partial J^{(i)}}{\partial \theta_j} = (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)},$$

implying

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial J^{(i)}}{\partial \theta_j}.$$

The key insight then is that instead of using the sum of all m partials of the $J^{(i)}$ on each iteration of updating θ, we could use only some of them on each iteration. The number that we use on each iteration is called the *batch size*; the normal variant of gradient descent has a batch size of m and is (somewhat confusingly) known as *batch gradient descent*. This allows us to do a few things:

- **Stochastic gradient descent.** This uses a batch size of 1, and the training examples are shuffled before starting this algorithm (so that each example is considered in a random order). Sometimes the elements are cycled through multiple times to get a better convergence result. Furthermore, this algorithm does not strictly converge in the same way that gradient descent does (its movements are more noisy); convergence can be evaluated using learning curves but examining the average of the objective function over the last n iterations for some $n > 1$. One way to achieve true convergence, however, is to gradually decrease α over time.

- **Mini-batch gradient descent.** This is similar to stochastic gradient descent; it refers to using any other batch size besides 1 and m.

- **Online learning.** This is similar to stochastic gradient descent, except that we use it in situations where we explicitly have a stream of training examples (as opposed to a well-defined set of them). One advantage of online learning is that the parameters θ will adapt over time if the underlying property that we are trying to learn changes over time.

- **Map-reduce.** Because the gradient is a summation of independent terms, we can compute the terms in parallel with many machines and combine the results output by those machines, which can significantly improve runtime.