Coursera Notes

Siavash Aslanbeigi

April 2018

Contents

1 Introduction **2**
1.1 Supervised learning . . . 3
1.2 Unsupervised learning . . . 4

2 Linear regression with one variable **5**
2.1 Model representation . . . 6
2.2 Cost function . . . 6
2.3 Minimizing the cost function . . . 8

3 Linear regression with multiple variables **10**
3.1 Model representation . . . 10
3.2 Cost function . . . 11
3.3 Normal equation . . . 12
3.4 When the normal equation fails . . . 13
3.5 When the normal equation is infeasible . . . 15
3.6 Polynomial regression . . . 16
3.7 Probabilistic view . . . 17

4 Gradient Descent **20**
4.1 Impact of the learning rate . . . 21
4.2 Local minima and initialization . . . 22
4.3 Feature scaling . . . 23
4.4 Gradient descent for linear regression . . . 28
4.5 Line search . . . 28

A Linear Algebra **30**

B Singular-Value Decomposition **31**
B.1 Intuition . . . 31
B.2 Proof . . . 32

C Direction of steepest descent **34**

1 Introduction

Machine learning is a collection of techniques which allow computers to complete tasks by learning from data. It grew out of work in Artificial Intelligence (AI). Traditionally, we program computers to do what we want by giving them precise instructions. There are problems for which this technique doesn't work very well.

Consider the problem of recognizing hand-written digits: given an image of a hand-written digit, the computer is supposed to map it to the corresponding digit, i.e. pick correctly from $0, 1, \ldots, 9$. Figure 1 shows examples of hand-written digits. It is quite easy for a human to recognize these digits correctly, even though none of them have the exact same shape. Programming a computer to do the same turns out to be difficult. The reason is that it's impossible to explicitly tell the computer about all possible shapes; there are just too many of them. Machine learning approaches this problem differently: we present the algorithm with images like those shown in Figure 1 (input), as well as the correct digits they represent (output), and let the algorithm figure out for itself what the correct way of recognizing hand-written digits is. Once the algorithm is trained on the data, we can use it to recognize digits not in the dataset, i.e. to make predictions. This technique has proven more reliable than any other approach. It also seems more inline with how humans learn from experience.

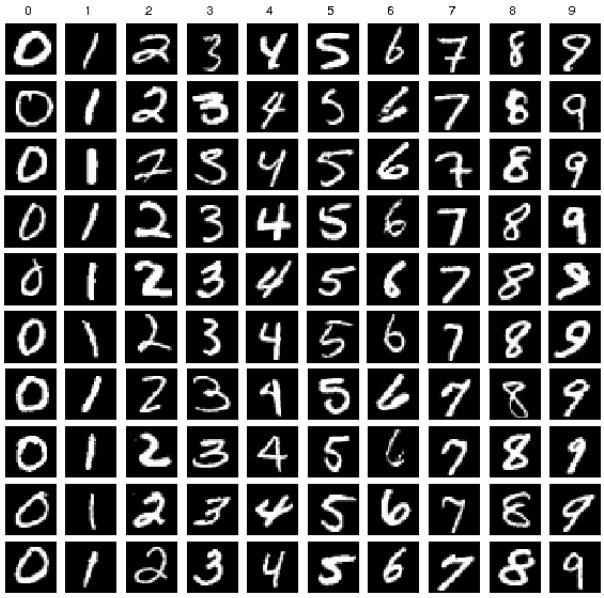


Figure 1: Examples of handwritten digits from the MNIST dataset.

Other modern applications of machine learning include:

- Spam filters: it is impossible to explicitly list all possible spam emails one

might receive.

- Database mining: growth of automation/web has resulted in much larger data sets which need to be understood. Examples include web click data, medical records, etc.
- Natural Language Processing (NLP), whose aim is to make computers understand text.
- Self-customizing programs, such as Netflix product recommendations.
- Understanding the human brain. How is it that we humans learn?

Here are two formal definitions of machine learning:

- Arthur Samuel (1959): Field of study that gives computers the ability to learn without being explicitly programmed. Samuel built a Checkers playing program where the computer would learn from playing tens of thousands of games against itself. This was the world's first self-learning program.
- Tom Mitchell (1998): A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

There are two main types of machine learning algorithms: supervised-learning and unsupervised learning.

1.1 Supervised learning

Supervised learning is the most common type of machine learning problem. The name **supervised** comes from the fact that the training samples (i.e. data set) contain the "right answers". In other words, for a set of inputs, we know what the output should be. Suppose we have the following data for prices of houses:

What is the price of a house which is 800ft^2? Although none of the houses in our data set have that size, we could make a prediction by fitting a curve through the data points, as shows in Figure 2.

Supervised learning algorithms fall in two broad categories:

- **Regression**: output values are continuous, for instance, predicting house prices, stock prices, etc.
- **Classification**: output values are discrete, for instance whether or not a tumor is malignant or benign.

Suppose we have a data set about breast tumors: we are given the size of each tumor, age of the patient, and whether or not it's malignant.

Given a new tumor, we'd like to be able to predict whether or not it's malignant or benign. This is an example of a classification problem, since the output values are discrete (we can assign 0 to benign tumors, and 1 to malignant ones.) The learning algorithm may decide that the line drawn in Figure 3 separates benign and malignant tumors.

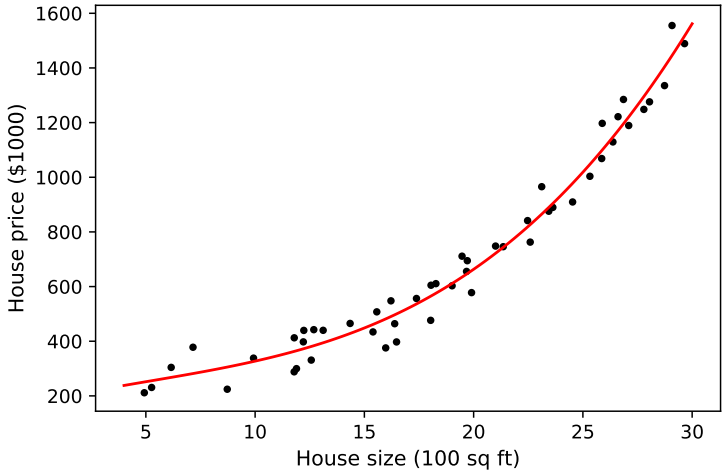


Figure 2: Predicting house prices.

1.2 Unsupervised learning

In supervised learning, the training samples tell us the correct outputs for a set of inputs. We then seeks to predict the output for new examples. Unsupervised learning is different in that we not given the right answers. It's more like, here's a data set, find some structure in it. For instance, a learning algorithm may tell us that in Figure 4 most of the data is concentrated in two different regions,those which are circled. This is an example of a clustering algorithm.

A real-life application of clustering is Google News, which looks at thousands of news and groups similar ones together. For instance, it would recognize that these articles from different sources are about the same topic and clusters them together:

- The Source: BP kills Macondo, but its legacy lives on.
- CNN: Well is dead, but much Gulf Coast work remains.
- Guardian: BP oil spill costs nearly $10bn dollars.

How remarkable is this? Here's another example of an unsupervised learning algorithm: the cocktail party problem, demonstrated in Figure 5.

Two people are talking at the same time. In microphone 1, the sound of speaker 1 is recorded more loudly than speaker 2, simply because speaker 1 is closer to microphone 1, and vice versa for microphone 2. If the volume offsets are pronounced enough, a human would easily recognize by listening to the recording of microphones 1 and 2 that there are two speakers, and even what each of them is saying. The amazing thing is that there are unsupervised

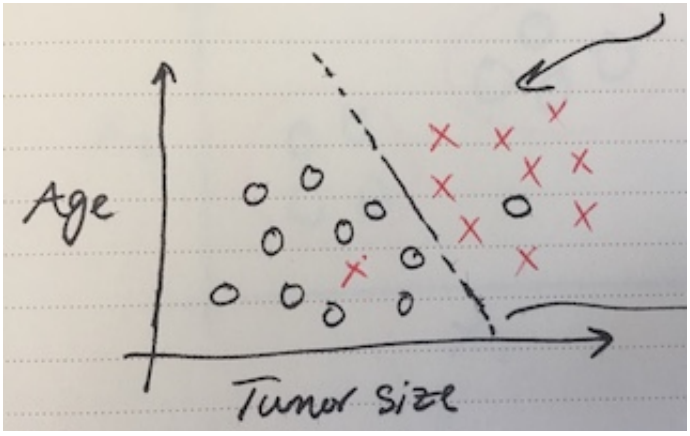


Figure 3: Predicting whether new tumor is malignant or benign. Circles denote benign tumors, and Xs malignant ones.

learning algorithms that are capable of doing the same. Based on the volume offsets, they recognize the structures of the two speakers sounds and are able to separate them out.

2 Linear regression with one variable

Consider a *training set* (data set which we'll use to fit parameters of our model, or *train* our model) for house prices in Portland, OR:

Size (ft^2) (x)	Price ($1000) (y)
2104	460
1416	232
1534	315
852	178
⋮	⋮

Our job is to learn from this data how to predict prices of houses as a function of their size. This is a supervised learning problem, because our training set contains the right answers: for every input (house size), we know the right output (house price). Moreover, this is a regression problem, because the output variable (house price) is continuous. Figure 6 shows the picture of how we will tackle this problem: given the training set, our learning algorithm will spits out a function h, called the *hypothesis* function, which can estimate the price of a house given its size.

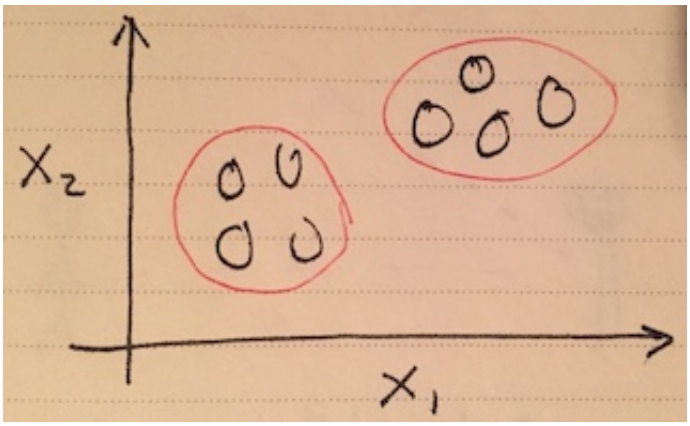


Figure 4: Clustering algorithm.

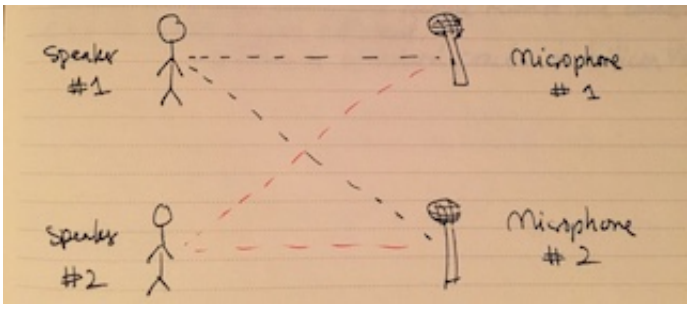


Figure 5: Cocktail party problem.

2.1 Model representation

How do we represent h? We start with the simplest function possible and consider more complex models later:

$$h_\theta(x) = \theta_0 + \theta_1 x. \tag{1}$$

This function assumes a linear relationship between the size of a house x and its price $h_\theta(x)$. We will come up with a learning algorithm that uses the training set to estimate the coefficients θ_0 and θ_1. With that at our disposal, we can predict the price of a house given its size. This learning algorithm is sometimes referred to as *univariate linear regression*, because $h_\theta(x)$ is a linear function of a single variable (size of house).

2.2 Cost function

What values of θ_0 and θ_1 provide the best fit to our data set? Let's start by establishing notation:

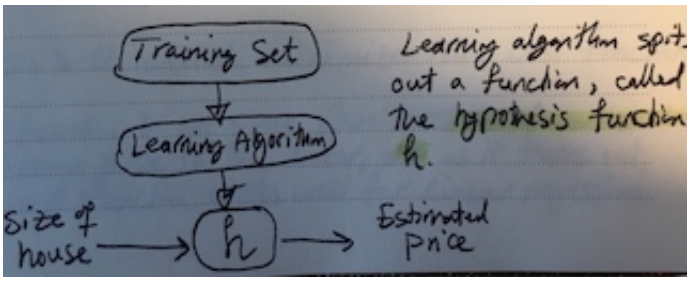


Figure 6: Big picture.

- m: number of training examples.
- x: input variable/feature (size of house).
- y: output/target variable (price of house).
- $(x^{(i)}, y^{(i)})$: ith training example (size and price of ith house).

For a given house size $x^{(i)}$ in our training set, we know the correct house price $y^{(i)}$. It then makes sense to pick θ_0 and θ_1 so that $h_\theta(x^{(i)})$ is as close to $y^{(i)}$ as possible. Realistically, we cannot always pick θ_0 and θ_1 so that $h_\theta(x^{(i)}) = y^{(i)}$ for all examples, since we only have two parameters to tune, but many more examples to fit, i.e. $m \gg 2$. As a result, we need to strike a balance across all examples, so that the collective prediction error $|h_\theta(x) - y|$ on the training set is minimized. To that end, consider the following function, called the *cost function*:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{2}$$

$$= \frac{1}{2m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2. \tag{3}$$

Given θ_0 and θ_1, it computes the sum of (squared) prediction errors across all training examples. (It also normalizes the sum by $2m$, but that's only for mathematical convenience.) The lower the value of J, the closer the model's prediction to the observed house prices in the training set. So it makes sense to pick θ_0 and θ_1 so that J is minimized. Figure 7 shows the correspondence between minimizing J and getting a better fit to to the training set.

Why cost function (3) and not some other one? There are plenty of other cost functions whose minimization would also lead to a good fit of data. For linear regression, though, (3) does have some desirable mathematical properties. First of all, J is a quadratic function of θ_0 and θ_1, i.e. it's a paraboloid. As a result, it doesn't have any local minima, only a global one. This makes minimization a much easier task. Secondly, as we will show in the next section, minimization of J can be carried out analytically.

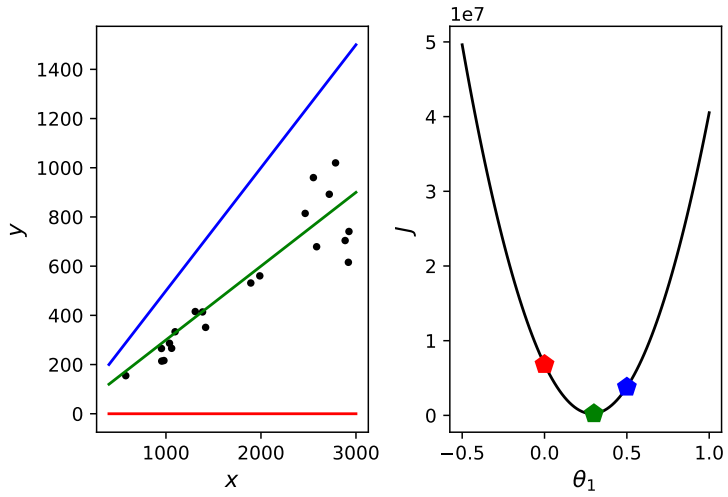


Figure 7: Correspondence between minimizing the cost function and getting a better fit to data. The following model is used to fit the data: $h_\theta(x) = \theta_1 x$. Black dots on the left show the training set. The right figure shows the cost function $J(\theta_1)$. Every dot on the right figure corresponds to a line of the same color on the left figure. We see that the closer θ_1 is to J's minimum, the better the fit to data is. Code used to generate this plot can be found here.

2.3 Minimizing the cost function

The global minimum of J occurs where its gradient is zero:

$$\frac{\partial J}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)}) = 0$$
$$\frac{\partial J}{\partial \theta_1} = \frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)})x^{(i)} = 0.$$

Thus, we have to solve the following system of linear equations

$$m\theta_0 + \left[\sum_{i=1}^{m} x^{(i)}\right]\theta_1 = \sum_{i=1}^{m} y^{(i)}$$
$$\left[\sum_{i=1}^{m} x^{(i)}\right]\theta_0 + \left[\sum_{i=1}^{m}\left(x^{(i)}\right)^2\right]\theta_1 = \sum_{i=1}^{m} x^{(i)}y^{(i)},$$

the solution to which is given by

$$\begin{pmatrix}\theta_0\\ \theta_1\end{pmatrix} = \begin{pmatrix} m & \sum_{i=1}^{m} x^{(i)} \\ \sum_{i=1}^{m} x^{(i)} & \sum_{i=1}^{m}\left(x^{(i)}\right)^2\end{pmatrix}^{-1}\begin{pmatrix}\sum_{i=1}^{m} y^{(i)} \\ \sum_{i=1}^{m} x^{(i)}y^{(i)}\end{pmatrix}. \tag{4}$$

We have solved the problem we posed in the beginning of this section. Given the training set, we can use (4) to estimate θ_1 and θ_2, and then make predictions using (1).

3 Linear regression with multiple variables

In the previous section, we wanted to predict the price of a house as a function of its size. Of course, there are many more factors that could affect house prices:

Size (ft^2) (x_1)	Number of bedrooms (x_2)	Number of floors (x_3)	Age of home (years) (x_4)	Price ($1000) (y)
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	10	178
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

The independent variables $x_1 - x_4$ (e.g. size of house, number of bedrooms, etc), which we think the output variable (house price) depends on, are called *features*. In this section, we will generalize univariate linear regression to incorporate more than just one feature.

3.1 Model representation

We generalize the univariate hypothesis (1) to a multivariate one as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n. \tag{5}$$

There are now n features, instead of just one. Let's establish notation, which will help us express (5) in matrix form:

- n: number of features.
- $x_j^{(i)}$: value of jth feature for the ith training example.
- $x_0 \equiv 1$. In other words, $x_0^{(i)} = 1$ for all examples i.
- $x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^{n+1}$ denotes the vector of features.
- $x^{(i)} = \begin{pmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \vdots \\ x_n^{(i)} \end{pmatrix} \in \mathbb{R}^{n+1}$ denotes the vector of features for the ith training example.

- $\theta = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{pmatrix} \in \mathbb{R}^{n+1}$ is the vector of parameters.

For data shows in Table 3: $n = 4$, $x_3^{(2)} = 2$, etc. We have created a fictitious feature x_0 whose value is always set to 1. This helps us rewrite the hypothesis function in matrix form:

$$h_\theta(x) = \theta^T x. \tag{6}$$

3.2 Cost function

We will use the same cost function (2), but with hypothesis (5):

$$\begin{aligned} J(\theta) &= \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \\ &= \frac{1}{2m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})^2. \end{aligned} \tag{7}$$

Now J is a function of $n+1$ variables $(\theta_0, \theta_1, \ldots, \theta_n)$, and we will need to find the vector θ that minimizes J. Before dealing with the minimization problem, let's rewrite J in a more compact form. Consider the following definitions:

- Design matrix X: $m \times (n+1)$ matrix whose elements are given by $X_{ij} = x_j^{(i)}$. Every row corresponds to a training example, and every column to a feature.
- Target vector Y: m-dimensional vector of all outputs in the training set, i.e. $Y_i = y^{(i)}$.

For data shows in Table 3, the design matrix and target vector are given by

$$X = \begin{pmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 10 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}, \qquad Y = \begin{pmatrix} 460 \\ 232 \\ 315 \\ 178 \\ \vdots \end{pmatrix}. \tag{8}$$

With these definitions, we can show that $\theta^T x^{(i)}$ is the ith component of the the m-dimensional vector $X\theta$: $\theta^T x^{(i)} = \sum_{j=0}^{n} \theta_j x_j^{(i)} = \sum_{j=0}^{n} X_{ij}\theta_j = [X\theta]_i$. This

in turn helps us write J in terms of the squared norm of the vector $X\theta - Y$:

$$\begin{aligned} J(\theta) &= \frac{1}{2m}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2 \\ &= \frac{1}{2m}\sum_{i=1}^{m}([X\theta]_i - Y_i)^2 \\ &= \frac{1}{2m}\sum_{i=1}^{m}[X\theta - Y]_i[X\theta - Y]_i \\ &= \frac{1}{2m}(X\theta - Y)^T(X\theta - Y). \end{aligned} \tag{9}$$

Why should we care about writing J in matrix (or sometimes called *vectorized*) form? One reason is parallelization: matrix multiplication is a highly parallelizable operation. Consider $C = AB$, where A and B are 1000000×5 and 5×1 dimensional, respectively. All 1000000 components of C can be computed in parallel, because they do not depend on one another. Most programming languages have highly optimized libraries for matrix multiplication. In the following Python example, it's shown that a vectorized implementation of (9) is more than two orders of magnitude faster than a for-loop implementation of (7). This is not all due to parallelization; overhead of the for-loop itself is probably a major factor. Nevertheless, this demonstrates the importance of vectorized implementations for large data sets.

Writing expressions in matrix form also makes it easier to see when results from linear algebra could be relevant. For instance, later in this section we will use the eigenvalues/eigenvectors of $X^T X$ to show that minimization of J does not lead to a unique solution for θ when $m < n + 1$.

3.3 Normal equation

Just like the univariate case, J is a quadratic function of $\theta_0, \theta_1, \ldots, \theta_n$, and its global minimum occurs where its gradient is zero. Let's compute the gradient of J with respect to θ_j:

$$\begin{aligned} \frac{\partial J}{\partial \theta_j} &= \frac{1}{m}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})x_j^{(i)} \\ &= \frac{1}{m}\sum_{i=1}^{m}[X\theta - Y]_i X_{ij} \\ &= \frac{1}{m}[X^T(X\theta - Y)]_j \end{aligned} \tag{10}$$

or equivalently

$$\nabla J = \frac{1}{m}X^T(X\theta - Y), \tag{11}$$

where ∇J is the gradient of J:

$$\nabla J = \begin{pmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \vdots \\ \frac{\partial J}{\partial \theta_N} \end{pmatrix}. \tag{12}$$

As a sanity check, note that the dimensions of the two sides match: $\nabla_\theta J$ is an $(n+1)$-dimensional vector, and $X^T(X\theta - Y)$ is a multiplication between an $(n+1) \times m$ matrix and an m-dimensional vector, which again is an $(n+1)$-dimensional vector. Setting the gradient to zero, we arrive at the value of θ that minimizes J:

$$\theta = \left(X^T X\right)^{-1} X^T Y. \tag{13}$$

This is called the *normal* equation. You should check for yourself that (13) reduces to (4) when $n = 1$.

So there we have it. Given the training set in the form of the design matrix X and the target vector Y, we can estimate θ via (13) and then use (6) to make predictions for new examples.

3.4 When the normal equation fails

The normal equation (13) can only work if $X^T X$ is invertible. Let's look at scenarios where this is not the case, and understand the reasons intuitively and mathematically. Consider the following design matrix, constructed using the first two examples in Table 3:

$$X = \begin{pmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \end{pmatrix}. \tag{14}$$

It can be checked that $\det(X^T X) = 0$. Therefore, $X^T X$ is not invertible. Is there something special about this dataset? As it turns out, this result holds no matter how we tweak the numbers. This may seem a bit surprising. However, let's consider what it would've meant if $X^T X$ were invertible. The dataset contains five features (including x_0), so the normal equation will solve for five parameters $\theta_0, \theta_1 \ldots \theta_4$, but using only two examples. In short, there are two equations and five unknowns. There's not enough data to fix the parameters of the model.This in turn manifests itself through the non-invertibility of $X^T X$.

More generally, we should expect that $X^T X$ is not invertible when $m < n+1$, i.e. when there are fewer examples than features. Let's prove this. We will make use of the Singular-Value Decomposition (SVD), which states that any matrix $m \times n$ can be written as

$$A = UDV^T, \tag{15}$$

where:

- U is an $m \times m$ matrix which satisfies $U^T U = UU^T = 1_m$, where 1_m is the m-dimensional identity matrix.

- V is an $n \times n$ matrix which satisfies $V^T V = V V^T = 1_n$, where 1_n is the n-dimensional identity matrix.
- D is an $m \times n$ matrix with zero elements everywhere except $D_{ii} \geq 0$ where $i = 1, \ldots, \min(m, n)$. So D is diagonal and has at most $\min(m, n)$ elements on the diagonal. These elements are called the singular values of A.

Appendix B provides a detailed explanation of SVD and its geometric meaning. Consider now the SVD of the design matrix $X = UDV^T$ (remember that X is $m \times (n+1)$ dimensional, so D and V are $m \times (n+1)$ and $(n+1) \times (n+1)$ dimensional, respectively):

$$\begin{aligned} X^T X &= (UDV^T)^T (UDV^T) \\ &= VD^T U^T UDV^T \\ &= VD^T DV^T, \end{aligned}$$

where in the third equality we've used $U^T U = 1_m$. Taking the determinant of both sides

$$\begin{aligned} \det(X^T X) &= \det(VD^T DV^T) \\ &= \det(D^T DV^T V) \\ &= \det(D^T D), \end{aligned}$$

where the second line uses the linear algebra identity $\det(AB) = \det(BA)$, and the last equality follows from $V^T V = 1_{n+1}$. $D^T D$ is an $(n+1) \times (n+1)$ diagonal matrix. When $m < n+1$, $D^T D$ will definitely have zero elements on the diagonal, because D itself has at most $\min(m, n+1)$ non-zero elements. If $D^T D$ has zero elements on the diagonal, its determinant will be zero, and so will $X^T X$, concluding the proof.

We've shown that we need more examples than features for the normal equation to work. We should be careful not to cheat, though. Let's say we've been given a design matrix, where the same example has been repeated by mistake

$$X = \begin{pmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 2104 & 5 & 1 & 45 \\ 1 & 2104 & 5 & 1 & 45 \\ 1 & 2104 & 5 & 1 & 45 \\ 1 & 2104 & 5 & 1 & 45 \end{pmatrix}. \tag{16}$$

This design matrix has as many examples as features, so should we expect $X^T X$ to be invertible? Probably not, since our dataset is highly redundant. Indeed, it can be checked that $\det(X^T X) = 0$, and that D only has one non-zero element. Therefore, we need the number of *unique* examples to be greater than the number of feature.

Redundant features can also make $X^T X$ non-invertible. In the following design matrix, x_2 is a copy of x_1:

$$X = \begin{pmatrix} 1 & 2104 & 2104 \\ 1 & 1416 & 1416 \\ 1 & 1534 & 1534 \\ 1 & 852 & 852 \end{pmatrix}. \tag{17}$$

It can be checked that $\det(X^T X) = 0$. In fact, it's more subtle than that. The following design matrix also satisfies $\det(X^T X) = 0$:

$$X = \begin{pmatrix} 1 & 2104 & 1053 \\ 1 & 1416 & 709 \\ 1 & 1534 & 768 \\ 1 & 852 & 427 \end{pmatrix}. \tag{18}$$

This looks like a perfectly innocent dataset. What's going on? The reason is that x_2 is a linear combination of x_0 and x_1: $x_2 = x_0 + 0.5x_1$.

In summary, if $X^T X$ is not invertible, we can look for the following in the dataset:

- Are there more unique examples than features? Of course, if possible, we should aim for many more examples than features, maybe $m \gtrsim 10n$ to be safe.
- Are there any redundant features? This is harder to check, because one needs to determine which feature is a linear combination of the others.

3.5 When the normal equation is infeasible

Using the normal equation requires inverting $X^T X$, which is an $(n+1) \times (n+1)$ matrix. When n is large, this may not be feasible in practice. Matrix inversion is typically an $\mathcal{O}(n^3)$ operation. This is demonstrated in Figure 8, where a matrix inversion routine in Python is applied to random matrices with different dimensions.

If inverting $X^T X$ is not possible in practice, how else can we estimate the parameters θ? Remember we obtained the normal equation by minimizing the cost function $J(\theta)$ (see (7) or (9)). In the next section we will describe an algorithm, called Gradient Descent, which directly minimizes $J(\theta)$ and is faster for large values of n. How large is too large for the normal equation? About $n \simeq 10,000$. That seems like an unreasonably large number of features. For instance, it's hard to imagine there could be $10,000$ features relevant for predicting house prices. Consider, though, making predictions based on images. If every pixel is considered a feature, which is often the case for image recognition problems, having $10,000$ features is equivalent to processing images that are only 100px $\times$ 100px; not large by any means!

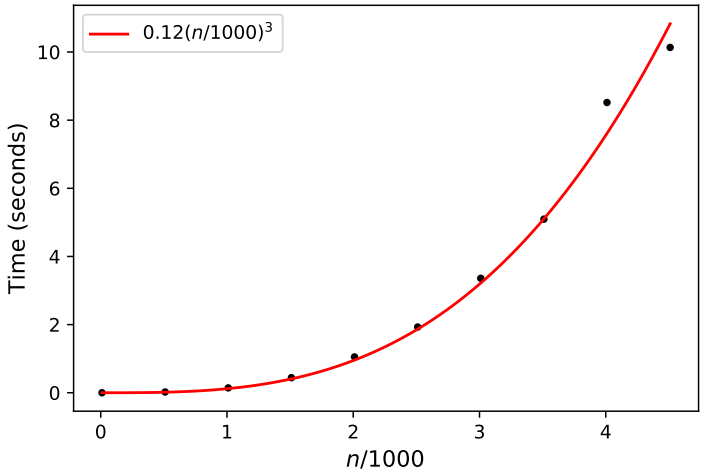


Figure 8: How matrix inversion scales with dimension. Code used to generate this plot can be found here.

3.6 Polynomial regression

What if our data is better modeled as a polynomial? For instance, fitting a linear model to data shown in Figure 9 doesn't seem appropriate. As it turns out, the following model provides a much better representation:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 \tag{19}$$

Can we use the machinery of linear regression to estimate the coefficients $\theta_0 \dots \theta_3$? Consider the following definitions:

$$x_1 = x, \qquad x_2 = x^2, \qquad x_3 = x^3. \tag{20}$$

using which (19) takes the form (5), or equivalently (6) where the feature vector is given by:

$$\begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \end{pmatrix}.$$

By transforming non-linear terms into new features, we can carry out linear regression as prescribed in the previous section. This trick can be used for far more complicated models, such as

$$h_\theta(x) = \theta_0 + \theta_1 x^2 e^{-2x} + \theta_2 \sqrt{x}. \tag{21}$$

Given that we can treat such complicated functions, what's with the word *linear* in linear regression? It refers to the parameters θ. An example of a model that

cannot be treated using linear regression is $h_\theta(x) = e^{-\theta x}$, because it's not a linear function of θ.

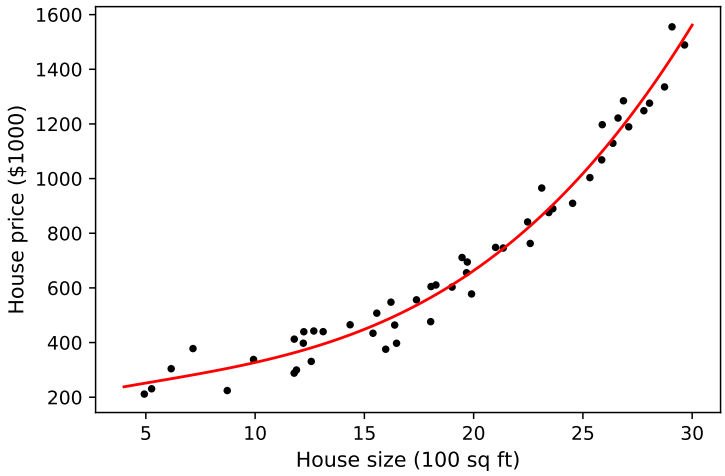


Figure 9: Non-linear relationship between house price and size. Code used to generate this plot can be found here.

3.7 Probabilistic view

Suppose we look at houses with similar characteristics such as number of bedrooms, number of floors, age, etc. Even though they are similar, they will likely not have the exact same price. For instance, one owner may be under time pressure and is willing to sell below market price. We should always expect a certain degree of randomness that our features cannot capture. We can, however, hope to design a model that makes the right predictions *on average*. This suggests a probabilistic approach for modeling. For a given set of features x, we can think of the house price y as a random variable with a certain probability distribution, whose mean is given by the hypothesis $h_\theta(x)$.

For simplicity, let's say y has a normal distribution with standard deviation σ, and that all observations are independent. What is the probability of observing the training set $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$, assuming that our hypothesis $h_\theta(x)$ is correct on average? The probability that we observe a house with features $x^{(i)}$ and price $y^{(i)}$ is

$$P^{(i)} = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y^{(i)} - h_\theta(x^{(i)}))^2}{2\sigma^2}\right]. \tag{22}$$

Since we're assuming all observations are independent, the probability of ob-

serving the training set is given by

$$P = \prod_{i=1}^{m} P^{(i)} = \frac{1}{(2\pi\sigma^2)^{\frac{m}{2}}} \exp\left[-\sum_{i=1}^{m} \frac{(y^{(i)} - h_\theta(x^{(i)}))^2}{2\sigma^2}\right]. \tag{23}$$

The quantity in the exponent can be rewritten in terms of $J(\theta)$ (see (7)):

$$P = \frac{1}{(2\pi\sigma^2)^{\frac{m}{2}}} \exp\left[-\frac{mJ(\theta)}{\sigma^2}\right]. \tag{24}$$

Interesting! The probability of observing the training set can be written in terms of the cost function. In fact, the parameters θ that maximize the probability of observing the training set are precisely those that minimize the cost function $J(\theta)$. Picking the parameters of a model to maximize the probability of observing the dataset is called *maximum likelihood estimation*. Before we picked the cost function based on heuristic arguments; now we see there's a deeper principle from which it can be arrived at.

Note that going from (23) to (24) assumes all observations are independent and have the same standard deviation σ. If every $y^{(i)}$ has its own standard deviation σ_i, the following cost function should be minimized:

$$\tilde{J}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \frac{1}{\sigma_i^2} (h_\theta(x^i) - y^{(i)})^2. \tag{25}$$

The wider the distribution of $y^{(i)}$ (i.e. the larger σ_i), the less it contributes to the overall cost. This makes sense: if σ_i is very large compared to the other examples, we do not have much confidence in $y^{(i)}$, so we shouldn't assign too much weight to it.

We can further generalize to the case where $y^{(1)}, y^{(2)}, \ldots, y^{(m)}$ are distributed as a multivariate Gaussian with covariance matrix Σ. Probability of observing the training set is then given by

$$P = \frac{1}{(2\pi|\Sigma|)^{\frac{m}{2}}} e^{-m\tilde{J}(\theta)}, \tag{26}$$

where

$$\begin{aligned}
\tilde{J} &= \frac{1}{2m} \sum_{i,i'=1}^{m} \Sigma^{-1}_{ii'} (y^{(i)} - h_\theta(x^{(i)}))(y^{(i')} - h_\theta(x^{(i')})) \\
&= \frac{1}{2m} \sum_{i,i'=1}^{m} \Sigma^{-1}_{ii'} [Y - X\theta]_i [Y - X\theta]_{i'} \\
&= \frac{1}{2m} (X\theta - Y)^T \Sigma^{-1} (X\theta - Y).
\end{aligned} \tag{27}$$

This expression is very similar to (10), except for the appearance of Σ. Again, maximizing P is equivalent to minimizing $\tilde{J}$. Following the same steps as in

section 3.3, it can be shown that

$$\nabla_\theta \tilde{J} = \frac{1}{m} X^T \Sigma^{-1} (X\theta - Y). \tag{28}$$

Setting $\nabla_\theta \tilde{J}$ to zero and solving for θ:

$$\theta = (X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} Y. \tag{29}$$

Figure 10 shows an example of using (10) vs. (29) to fit house prices. When the training set is large, the difference is likely quite small because variation in the data captures the underlying distribution quite well. Indeed, we cannot know the distribution of $\{y^{(i)}\}$ in the first place if we don't collect a lot of data. In physical experiments, the distribution of $y^{(i)}$ depends largely on the measuring apparatus. No measurement is ever 100% accurate and it is crucial to understand the noise.

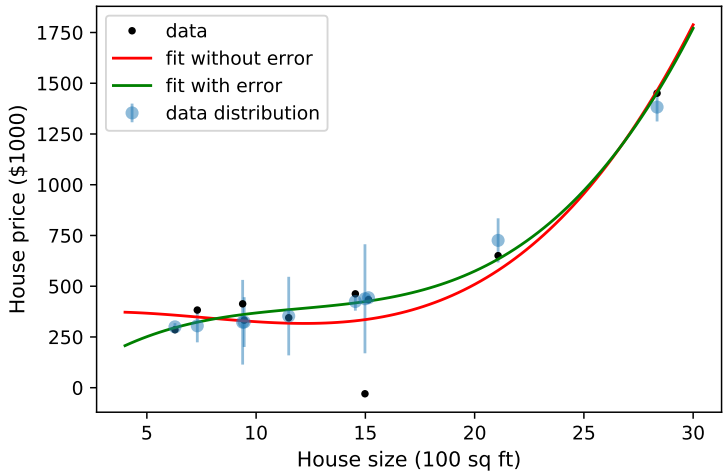


Figure 10: Fitting house prices when $y^{(i)}$ are drawn from different normal distributions. Blue dots show the mean of $y^{(i)}$ and the error bars show the standard deviation. Black dots are sampled house prices. The red fit uses (10), while the green fit uses (29). Code used to generate this plot can be found here.

4 Gradient Descent

We saw in the previous section that training linear regression involves minimizing a cost function $J(\theta_1, \ldots, \theta_N)$. We were able to carry out the minimization analytically and arrived at the normal equation. That is only possible because the hypothesis function $h_\theta(x)$ is linear in the fitting parameters $\theta_1, \ldots, \theta_N$, which is not the case for most other learning algorithms. Even for linear regression, as we explained in section 3.5, the normal equation becomes infeasible when there are a large number of features. Therefore, we need an algorithm for minimizing a generic cost function $J(\theta_1, \ldots, \theta_N)$. *Gradient descent* is one such algorithm. To make the notation simpler, we will use θ to denote the vectors of parameters $\theta_1, \ldots, \theta_N$ and simply write the cost function as $J(\theta)$.

The general idea of gradient descent is to start at some value of θ, and then keep changing θ to reduce J as much as possible, until we hopefully end up at the minimum. How do we change θ? One way is to look for the direction along which J decreases the fastest locally, and take a small step in that direction. This makes sense, since our goal is to minimize J and we want to arrive at the minimum as quickly as possible. What *is* the direction along which J decreases the fastest? The answer turns out to be the opposite direction of the gradient ∇J. To see why this makes sense, consider the Taylor expansion of J about θ:

$$J(\theta + \delta\theta) = J(\theta) + \nabla J \cdot \delta\theta + \cdots . \tag{30}$$

If we make a small move in the opposite direction of the gradient, i.e. $\delta\theta = -\epsilon \nabla J$ where ϵ is a sufficiently small positive number, the cost function is guaranteed to decrease:

$$\Delta J = J(\theta + \delta\theta) - J(\theta) \approx -\epsilon \nabla J \cdot \nabla J < 0. \tag{31}$$

This doesn't quite prove that J decreases the *fastest* in this direction. To see why that is, note that the first-order change in J is given by the inner product of the vectors ∇J and $\delta\theta$: $\Delta J \approx \nabla J \cdot \delta\theta$. If we constrain the magnitude of $\delta\theta$, its inner product with ∇J is most negative when it is pointing in the opposite direction. A formal proof of this result is presented in Appendix C.

One step of gradient descent moves us along the direction of steepest descent as follows

$$\theta := \theta - \alpha \nabla J, \tag{32}$$

where $\alpha > 0$ is called the *learning rate* and determines how long a step we take along the direction $-\nabla J$. Concretely, this is how gradient descent works:

1. Start from some value $\theta^{(0)}$ and compute $\nabla J(\theta^{(0)})$.
2. $\theta^{(1)} = \theta^{(0)} - \alpha \nabla J(\theta^{(0)})$.
3. $\theta^{(2)} = \theta^{(1)} - \alpha \nabla J(\theta^{(1)})$.
4. ...

Figure 11 demonstrates this using a simple cost function: $J(\theta) = \theta^2$. With an appropriate learning rate $\alpha = 0.1$ and starting point $\theta^{(0)} = 2$, gradient descent is able to find the minimum after about 20 steps. It's a bit like rolling a ball down a valley; gravity will work its magic and the ball will eventually settle at the bottom. Note that even though the learning rate is constant, the steps keep getting smaller. The reason is that the closer to the minimum we get, the smaller the gradient becomes: at $\theta = 2$ we would take a step of size $-\alpha J'(2) = -0.4$, while at $\theta = 0.5$ the step size would only be -0.1. [1]

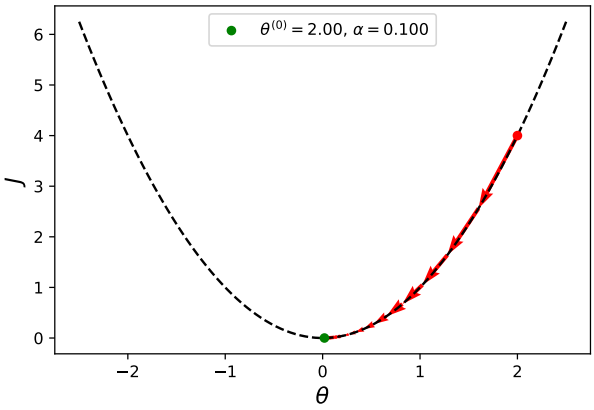


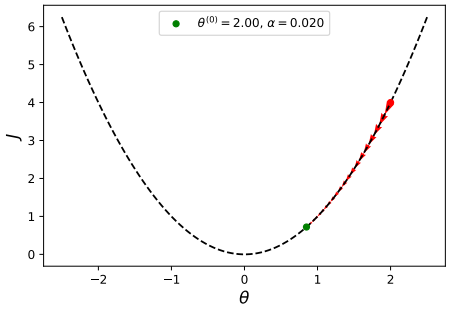
Figure 11: Gradient descent applied to $J(\theta) = \theta^2$. The starting and end points are shown by red and green dots, respectively. Code used to generate this plot can be found here.

4.1 Impact of the learning rate

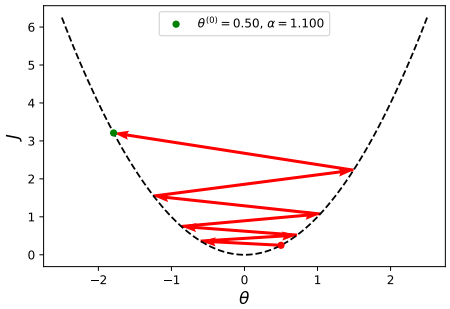
Each step of gradient descent moves θ by $-\alpha \nabla J(\theta)$. If α is too small, each step will make little progress and gradient descent will take a long time to converge to the minimum. This is demonstrated in Figure 12a, where the learning rate has been reduced $\alpha = 0.02$.

On the other hand, if α is too large, gradient descent might not converge at all. This is demonstrated in Figure 12b, where $\theta^{(0)} = 0.5$ and $\alpha = 1.1$. The first step overshoots the minimum and ends up at $\theta = -0.6$, where the gradient is even higher. This keeps on repeating and the steps keep getting larger. Earlier we showed that moving in the opposite direction of ∇J should decrease J. Why, then, does J increase in this example? The reason is that moving from θ to $\theta - \alpha \nabla J$ is only guaranteed to decrease J if $J(\theta - \alpha \nabla J) \approx J(\theta) - \alpha \nabla J \cdot \nabla J$, for which α needs to be sufficiently small.

[1] The ball-rolling analogy breaks down here, since the ball speeds up as it gets to the bottom. In fact, the ball will oscillate about the minimum until it comes to a stop due to friction.



(a) If α is too small, gradient descent will take long to converge.



(b) If α is too large, gradient descent might not converge.

Figure 12: Impact of the learning rate on convergence. The cost function used is $J(\theta) = \theta^2$. Code used to generate this plot can be found here.

It is hard to guess the appropriate learning rate *a priori*. In general, a good way of checking if gradient descent is working is by plotting J after every step. With a sufficiently small value of α, J should decrease at every step. If it is increasing, we can try decreasing α. If it is decreasing too slowly, we can try increasing α. Figure 13 shows how $J(\theta) = \theta^2$ changes for gradient descent runs in Figures 11, 12a, and 12b. In case of Figure 11, J converges to the minimum after about 10 iterations. For the run shown in Figure 12b, J increases after every step, which means the learning rate is too large.

This way of tracking the progress of gradient descent is easily generalizable when there are many features, i.e. when J is a function of many variables. That's not the case for Figures such as 11, where we plot J as a function of θ.

4.2 Local minima and initialization

Let's now look at a more complicated function:

$$J(\theta) = \frac{12}{5}\theta^4 + \frac{4}{5}\theta^3 - \frac{18}{5}\theta^2 + 2. \tag{33}$$

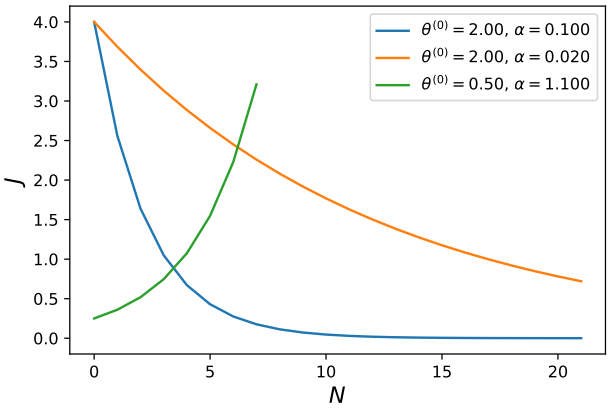


Figure 13: Tracking the progress of gradient descent applied to $J(\theta) = \theta^2$. Code used to generate this plot can be found here.

It has a global minimum at $\theta = -1$, a local minimum at $\theta = +3/4$, and a local maximum at $\theta = 0$. Will gradient descent always be able to find the global minimum? Unfortunately, no. It depends on the starting point. This is demonstrated in Figure 14. For any starting point $\theta^{(0)} > 0$, gradient descent will converge to the local minimum, while if $\theta^{(0)} < 0$, it will find the global minimum.

Another interesting feature of (33) is that it has a local maximum at $\theta = 0$. If we initialize gradient descent at $\theta = 0$, it will not make any progress at all, since $J'(0) = 0$. Even if we initialize close to the local maximum, progress will be slow, since $J'(\theta)$ is small there. Figure 15 demonstrates this, where the learning rate is kept the same as that in Figure 14, but the starting point is changed to $\theta = -0.01$ and the number of iterations is increased by a factor of 3. Even with so many more steps, gradient descent is not able to converge to the global minimum.

4.3 Feature scaling

So far we've been looking at 1d functions. Let's now look at a 2d example:

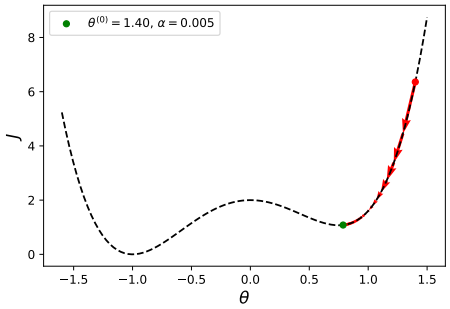
$$J(\theta_1, \theta_2) = 9(\theta_1 - 2)^2 + (\theta_2 - 1)^2. \tag{34}$$

Its minimum occurs at

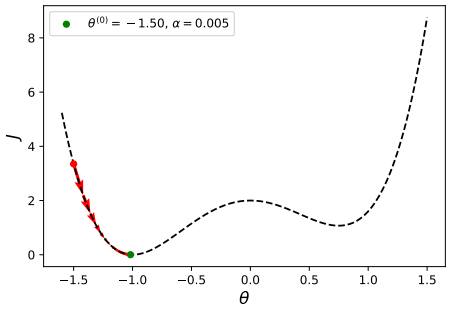
$$\theta = \begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix},$$

and its gradient is given by

$$\nabla J = \begin{pmatrix} 18(\theta_1 - 2) \\ 2(\theta_2 - 1) \end{pmatrix},$$



(a) Converging to the local minimum.



(b) Converging to the global minimum.

Figure 14: Impact of initialization on convergence. Cost function is given by (33). Number of gradient descent steps in each case is 30. Code used to generate this plot can be found here.

so the gradient descent update rule is

$$\begin{pmatrix}\theta_1\\ \theta_2\end{pmatrix} := \begin{pmatrix}\theta_1\\ \theta_2\end{pmatrix} - \alpha \begin{pmatrix}18(\theta_1 - 2)\\ 2(\theta_2 - 1)\end{pmatrix}.$$

In Figure 16a, we've applied gradient descent to (34), with $\alpha = 0.1$ and starting point $\theta_1 = \theta_2 = 5$. It makes large zig-zag moves about $\theta_1 = 2$ and approaches the minimum slowly along the θ_2 direction. The reason is that the gradient in the θ_1 direction is much larger. This can be seen from the contours of J; they are elongated vertically. The first step, for instance, goes from $\begin{pmatrix}5\\ 5\end{pmatrix}$ to $\begin{pmatrix}-0.4\\ 4.2\end{pmatrix}$, decreasing θ_1 by 5.4 and θ_2 by only 0.8.

One way to avoid the zig-zag moves is to reduce α, so that we do not overshoot $\theta_1 = 2$. This is done in Figure 16b, where $\alpha = 0.05$. Gradient descent takes one big step along the θ_1 direction, and many smaller ones along θ_2 towards the minimum. Since we decreased α to avoid large moves in the θ_1 direction, steps along θ_2 are now much smaller than they need to be.

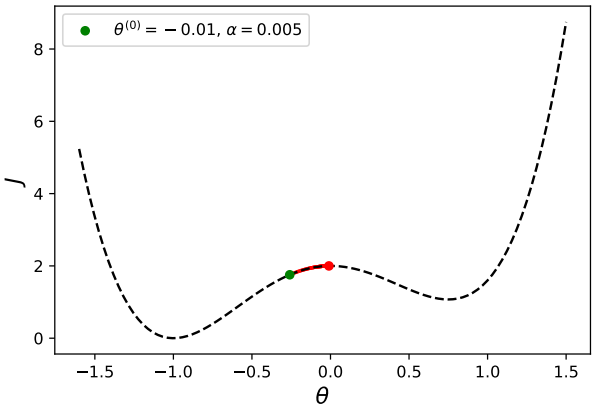


Figure 15: Gradient descent takes very small steps in regions where the gradient is small. Cost function is given by (33). Code used to generate this plot can be found here.

The problem we're facing is that J has a much larger gradient along θ_1 than θ_2. As it turns out, we can get around this issue by a simple redefinition:

$$\tilde{\theta}_1 = 3(\theta_1 - 2), \qquad \tilde{\theta}_2 = \theta_2 - 1, \tag{35}$$

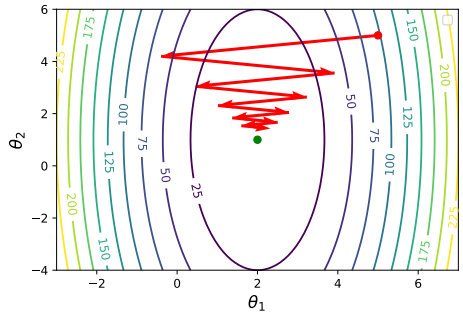
so that $J = \tilde{\theta}_1^2 + \tilde{\theta}_2^2$. We can now use gradient descent to minimize J w.r.t $\tilde{\theta}_1$ and $\tilde{\theta}_2$, and then simply use (35) to find the minimum w.r.t θ. [2] The crucial point is that the gradient of J along $\tilde{\theta}_1$ and $\tilde{\theta}_2$ is the same, so gradient descent can now move directly towards the minimum. This is demonstrated in Figure 17. Contours of J w.r.t $\tilde{\theta}_1$ and $\tilde{\theta}_2$ are circular, and we can now use a larger learning rate ($\alpha = 0.2$) to get much closer to the minimum after the same number of steps.

Let's consider the implications of what we have just seen for linear regression. Suppose we are considering two features for predicting house prices: size of house x_1 (ft^2) and number of bedrooms x_2. The hypothesis function h and cost function J are given as usual:

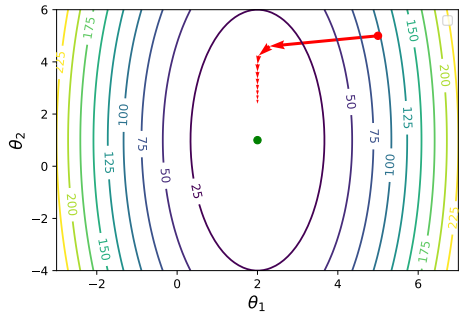
$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2, \qquad J = \frac{1}{2m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} - y^{(i)})^2. \tag{36}$$

A reasonable range of values for these features is $x_1 \sim 100-1000$ and $x_2 \sim 1-10$. Because x_1 is much larger than x_2, J changes a lot more quickly along θ_1 than θ_2. We just saw that running gradient descent on functions like this is terribly

[2] Since J is simple enough, we can check this analytically. The minimum of $\tilde{\theta}_1^2 + \tilde{\theta}_2^2$ occurs at $\tilde{\theta}_1 = \tilde{\theta}_2 = 0$, and with the rescaling (35), we recover the original minimum: $\theta_1 = \tilde{\theta}_2/3 + 2 = 2$, $\theta_2 = \tilde{\theta}_1 + 1 = 1$.



(a) $\alpha = 0.1$.



(b) $\alpha = 0.05$.

Figure 16: Gradient descent applied to (34). The starting point is $\theta_1 = \theta_2 = 5$ and the first 10 steps are shown. Code used to generate this plot can be found here.

inefficient, because it will spend most of the time zig-zagging in the θ_1 direction. Again, we can get around this problem by a redefinition

$$\tilde{x}_1 = \frac{x_1 - \mu_1}{\sigma_1}, \qquad \tilde{x}_2 = \frac{x_2 - \mu_2}{\sigma_2} \tag{37}$$

where

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}, \qquad \sigma_j = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2. \tag{38}$$

We are subtracting the mean μ_j of every feature x_j in the training set and scaling it by the standard deviation σ_j. As a result, the new features $\tilde{x}_1$ and $\tilde{x}_2$ take on values in a similar range (in most cases somewhere between -3 and 3). This procedure is called *feature scaling*. There are other ways of transforming the original features so they take on similar values. For instance, instead of dividing $x_j - \mu_j$ by the standard deviation σ_j, we could divide by the range of x_j, i.e. $r_j = \max(x_j) - \min(x_j)$.

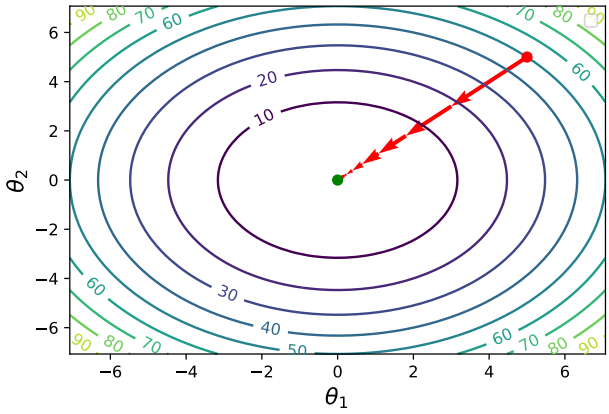


Figure 17: Gradient descent applied to $\theta_1^2 + \theta_2^2$, with $\alpha = 0.2$ and starting point $\theta_1 = \theta_2 = 5$. The first 10 steps are shown. Code used to generate this plot can be found here.

We can now run linear regression on the normalized features. Concretely, we will consider the hypothesis function

$$h = \tilde{\theta}_0 + \tilde{\theta}_1 \tilde{x}_1 + \tilde{\theta}_2 \tilde{x}_2, \tag{39}$$

and minimize the following cost function for $\tilde{\theta}_0, \tilde{\theta}_1$, and $\tilde{\theta}_2$:

$$J = \frac{1}{2m} \sum_{i=1}^{m} (\tilde{\theta}_0 + \tilde{\theta}_1 \tilde{x}_1^{(i)} + \tilde{\theta}_2^{(i)} \tilde{x}_2^{(i)} - y^{(i)})^2. \tag{40}$$

Gradient descent we will not zig-zag along $\tilde{\theta}_1$, since $\frac{\partial J}{\partial \tilde{\theta}_1}$ is not much larger than $\frac{\partial J}{\partial \tilde{\theta}_2}$. (Contours of J w.r.t $\tilde{\theta}_1$ and $\tilde{\theta}_1$ should be more or less circular.)

After finding the optimal parameters $\tilde{\theta}$ using gradient descent, we can predict the price of an unseen house x by following two steps: (i) transform x to $\tilde{x}$ via (37), (ii) plug $\tilde{x}$ in (39) to predict the house price. In other words:

$$h(x) = \tilde{\theta}_0 + \tilde{\theta}_1 \frac{x_1 - \mu_1}{\sigma_1} + \tilde{\theta}_2 \frac{x_2 - \mu_2}{\sigma_2}. \tag{41}$$

Had we optimized using the original features x, we would find that

$$\theta_0 = \tilde{\theta}_0 - \frac{\mu_1}{\sigma_1}\tilde{\theta}_1 - \frac{\mu_2}{\sigma_2}\tilde{\theta}_2, \qquad \theta_1 = \frac{\tilde{\theta}_1}{\sigma_1}, \qquad \theta_2 = \frac{\tilde{\theta}_2}{\sigma_2}.$$

It is important to include the bias term when using feature scaling. Try the following experiment: ignore θ_0 and $\tilde{\theta}_0$ in the analysis above, run linear regression once on the original features to find the optimal θ, and once on the normalized

features to find $\tilde{\theta}$. Then consider a new house x and predict its price once using $\theta_1 x_2 + \theta_2 x_2$, and another time using $\tilde{\theta}_1(x_1 - \mu_1)/\sigma_1 + \tilde{\theta}_2(x_2 - \mu_2)/\sigma_2$. You will find that the answers are different. Can you see what went wrong?

4.4 Gradient descent for linear regression

The gradient descent update rule for linear regression is given by (see (11))

$$\theta := \theta - \frac{\alpha}{m} X^T (X\theta - Y), \tag{42}$$

or equivalently (see (10))

$$\theta := \theta - \frac{\alpha}{m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)}) x^{(i)}. \tag{43}$$

4.5 Line search

We've seen so far that picking the learning rate is quite tricky. If it's too large, gradient descent might not converge. If it's too small, convergence might be very slow. Is there a way of automatically picking the optimal learning rate? Let's remember what the learning rate is actually doing. One step of gradient descent moves us in the direction $-\nabla J$ by $\alpha \nabla J$: $\theta \to \theta - \alpha \nabla J(\theta)$. So α determines how long a step we take in the direction $-\nabla J$. How about, then, tuning α to reduce J as much as possible along the direction $-\nabla J$? After all, our goal is to minimize J! What exactly would this involve? We can start with one value of α, move to $\theta - \alpha \nabla J(\theta)$, and compute the value of the cost function there: $J(\theta - \alpha \nabla J(\theta))$. We can repeat this for all possible values of α and ultimately pick the value that minimizes $J(\theta - \alpha \nabla J(\theta))$. This means we need to minimize the 1-dimensional function

$$f(\alpha) = J(\theta - \alpha \nabla J(\theta)) \tag{44}$$

at every step of gradient descent. Solving for $f'(\alpha) = 0$, we find

$$\frac{df}{d\alpha} = -\nabla J|_\theta \cdot \nabla J|_{\theta - \alpha \nabla J} = 0. \tag{45}$$

I have used the notation $\nabla J|_\theta$ to explicitly state where the gradient is being valued. This equation is telling us that the optimal learning rate moves us to a point where the gradient vector is orthogonal to the one at the current step. In other words, the inner product between $\nabla J|_\theta$ and $\nabla J|_{\theta - \alpha \nabla J}$ is zero. Assuming $\nabla J|_\theta \neq 0$ (so we're not sitting on a local optimum), this implies that $J|_{\theta - \alpha \nabla J}$ should either be zero, meaning we've reached a (local) minimum, or it should be orthogonal to $\nabla J|_\theta$.

In general, we will need to carry out this optimization numerically. In the special case where J is a quadratic function of θ, however, this can be done analytically. Consider the cost function

$$J(\theta) = \frac{1}{2} \theta^T A \theta + \theta^T B + C, \tag{46}$$

where A is an $N \times N$ positive-definite matrix, B is an N-dimensional vector, and C is a number. This describes an N-dimensional paraboloid. The cost function for linear regression takes precisely this form. Comparing this equation to (9): $A = \frac{1}{m}X^T X$, $B = -\frac{1}{m}X^T Y$, and $C = \frac{1}{m}Y^T Y$. Since J is quadratic in θ, ∇J is linear:

$$\nabla J = A\theta + B. \tag{47}$$

Using this, it can be checked that

$$\nabla J|_{\theta - \alpha \nabla J} = \nabla J|_\theta - \alpha A \nabla J|_\theta. \tag{48}$$

We can now solve (45) for the optimal learning rate:

$$\alpha = \frac{(\nabla J)^T \nabla J}{(\nabla J)^T A (\nabla J)} = \frac{(A\theta + B)^T (A\theta + B)}{(A\theta + B)^T A (A\theta + B)}. \tag{49}$$

Let's look at what this optimal learning does for us in some special cases. Suppose J is one-dimensional ($N = 1$), so that A and B are numbers. The optimal learning rate is simply $\alpha = 1/A$. The first step of gradient descent will take us from θ to $\theta - \alpha J'(\theta) = \theta - \frac{1}{A}(A\theta + B) = -B/A$, which is precisely where the global minimum is! No matter where we start, gradient descent will take us directly to the minimum in just one step, if we use the optimal learning rate. Consider now another case, where $J = \theta_1^2 + \theta_2^3 + \cdots + \theta_N^2$, i.e. $A = 1_N$, $B = 0$, and $C = 0$. The optimal learning rate is $\alpha = 1$. The first step of gradient descent takes us from θ to $\theta - \alpha(A\theta + B) = 0$, which is the global minimum. Again, it only takes one step to find the minimum.

More generally, if ∇J is along the direction of an eigenvector of A, it will only take one step to get to the minimum. To see why, let v be a normalized eigenvector of A with eigenvalue λ: $Av = \lambda v$, $v^T v = 1$. If ∇J points along the direction of v, then $\nabla J = ||\nabla J|| v$ and $A(\nabla J) = \lambda ||\nabla J|| v = \lambda \nabla J$. Plugging this back in (49) we find $\alpha = 1/\lambda$. One step of gradient descent would take us from θ to $\theta^{new} = \theta - \alpha \nabla J = \theta - \frac{1}{\lambda}\nabla J$. The gradient at θ^{new} is given by $A\theta^{new} + B = A\theta - \frac{1}{\lambda}A\nabla J + B = A\theta - \nabla J + B = 0$, which means θ^{new} is the minimum.

A Linear Algebra

Here we present definitions and proofs of most linear algebra results used in the notes.

Definition A.1. The conjugate transpose A^* of an $m \times n$ complex matrix A is an $n \times m$ matrix defined as follows: $A^*_{ij} = \overline{A_{ji}}$. In other words, A^* is obtained by first transposing A, and then take taking the complex conjugate of every entry, as shown in the example below:

$$A = \begin{pmatrix} 1 & 1+i & 2i \\ 3+2i & 2 & 5 \end{pmatrix}, \qquad A^* = \begin{pmatrix} 1 & 3-2i \\ 1-i & 2 \\ -2i & 5 \end{pmatrix}. \tag{50}$$

Definition A.2. A square complex matrix A is called Hermitian matrix if it is equal to its conjugate transpose $A = A^*$. The following is an example of an Hermitian matrix:

$$A = \begin{pmatrix} 1 & 1+i \\ 1-i & 1 \end{pmatrix}. \tag{51}$$

Note that a real Hermitian matrix is symmetric.

Theorem A.1. *Eigenvalues of a Hermitian matrix are real.*

Proof. Let A be a Hermitian matrix and consider its eigenvalues and eigenvectors:

$$Av_n = \lambda_n v_n. \tag{52}$$

Applying v_n^* to both sides:

$$\lambda_n |v_n|^2 = v_n^* A v_n = v_n^* A^* v_n = (Av_n)^* v_n = (\lambda_n v_n)^* v_n = \overline{\lambda_n} |v_n|^2 \tag{53}$$

where $|v_n|^2 = v_n^* v_n > 0$. Therefore, $\lambda_n = \overline{\lambda_n}$, which means λ_n is real. □

Theorem A.2. *Eigenvectors of a Hermitian matrix A with different eigenvalues are orthogonal, i.e.*

$$\lambda_n \neq \lambda_m \to v_n^* v_m = 0. \tag{54}$$

Proof.

$$\lambda_m v_n^* v_m = v_n^* (\lambda_m v_m) = v_n^* A v_m = v_n^* A^* v_m = (Av_n)^* v_m = \lambda_n v_n^* v_m, \tag{55}$$

where the last one uses the fact that λ_n is real. Thus, we've shown $(\lambda_m - \lambda_n) v_n^* v_m = 0$, and since we're assuming $\lambda_n \neq \lambda_m$, it follows that $v_n^* v_m = 0$. □

Definition A.3. A Hermitian matrix A is called *positive definite* if it satisfies the following condition for all vectors v:

$$v^* A v > 0. \tag{56}$$

Note that $v^* A v$ is real, because A is Hermitian: $\overline{v^* A v} = (v^* A v)^* = v^* A^* v = v^* A v$. Similarly, A is called positive *semi*-definite if $v^* A v$ is positive or zero: $v^* A v \geq 0$.

Theorem A.3. *All eigenvalues of a positive-definite matrix are positive. Similarly, all eigenvalues of a positive semi-definite matrix are non-negative.*

Proof. Let A be a positive definite matrix and consider its eigenvalues and eigenvectors:

$$Av_n = \lambda_n v_n. \tag{57}$$

Applying v^* t both sides: $\lambda_n v_n^* v_n = v_n^* A v_n > 0$, which implies $\lambda_n > 0$ since $v_n^* v_n > 0$. The same argument shows that any non-zero eigenvalue of a positive semi-definite matrix is positive. □

B Singular-Value Decomposition

The singular-value decomposition (SVD) is a powerful result that applies to any matrix. In this section, we will derive the SVD and explain what it means intuitively. The SVD states that any $m \times n$ matrix M can be written as

$$M = UDV^T, \tag{58}$$

where

- U is an $m \times m$ unitary matrix, i.e. it satisfies $U^T U = UU^T = 1_m$, where 1_m is the m-dimensional identity matrix.
- V is an $n \times n$ unitary matrix, i.e. satisfies $V^T V = VV^T = 1_n$, where 1_n is the n-dimensional identity matrix.
- D is an $m \times n$ matrix with zero elements everywhere except $D_{ii} \geq 0$ where $i = 1, \ldots, \min(m, n)$. So D is diagonal and has at most $\min(m, n)$ elements on the diagonal. These elements are called the singular values of X.

B.1 Intuition

Before delving into the proof, let's go through a simple example to understand what the matrices V, D, and U are doing. Consider the following matrix

$$M = \frac{1}{2}\begin{pmatrix} -\sqrt{3} & 3 \\ 5 & -\sqrt{3} \end{pmatrix}, \tag{59}$$

which admits the SVD

$$U = \frac{1}{2}\begin{pmatrix} -1 & \sqrt{3} \\ \sqrt{3} & 1 \end{pmatrix}, \qquad D = \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix}, \qquad V = \frac{1}{2}\begin{pmatrix} \sqrt{3} & 1 \\ -1 & \sqrt{3} \end{pmatrix}. \tag{60}$$

Let's look at the action of V^T on unit vectors u_x and u_y. This is shown in Figure 18a. It rotates the two vectors counterclockwise by $\pi/6$. We shouldn't

be surprised that V^T is acting as a rotation, since it is unitary. Let's remember why this is. A unitary matrix A preserves inner products between vectors:

$$(Av_1)^T(Av_2) = v_1^T(A^TA)v_2 = v_1^Tv_2. \tag{61}$$

This means the angle between v_1 and v_2 is the same as that between Av_1 and Av_2. It also means that the norm of v_1 and v_2 remain unchanged under the action of A. Rotations and reflections have these properties. Figure 18b shows the application of D, after V^T has been applied. We see that the vectors are being stretched ($D_{11} = 3$) times along the x-axis, and remain unchanged along the y-axis, because $D_{22} = 1$. Finally, Figure 18c shows the application of U after DV^T has been applied. The vectors are being reflected along the line that makes $\pi/3$ angle with the x-axis. Note that UDV^Tu_x and UDV^Tu_y are the first and second columns of M, respectively.

SVD is telling us that any matrix can be decomposed into a rotation/reflection, scaling along the coordinate axes, and another rotation/reflection. This result applies even when M is not a square matrix. For instance, if M is 2×3: V^T is a rotation/reflection in $\mathbb{R}^3$, D projects the resulting vector into $\mathbb{R}^2$ and stretches it along the coordinate axes, and U applies a final rotation/reflection in $\mathbb{R}^2$.

B.2 Proof

Let's now prove the singular-value decomposition. Consider the following two matrices

$$\Sigma = M^TM, \qquad W = MM^T. \tag{62}$$

Note that Σ is $n \times n$, W is $m \times m$, and they are both symmetric and positive semi-definite (see A.3):

$$\begin{aligned} v^T\Sigma v &= v^TM^TMv = (Mv)^T(Mv) = \|Mv\|^2 \geq 0, \\ v^TWv &= v^TMM^Tv = (M^Tv)^T(M^Tv) = \|M^Tv\|^2 \geq 0. \end{aligned}$$

Therefore, they can be diagonalized and all of their eigenvalues are non-negative (see A.3):

$$\Sigma v^{(i)} = \lambda_i v^{(i)}, \qquad (v^{(i)})^Tv^{(j)} = \delta_{ij}, \qquad \lambda_i \geq 0 \tag{63}$$

where $i, j = 1, \ldots n$ and similarly for W:

$$Wv^{(i)} = \xi_i w^{(i)}, \qquad (w^{(i)})^Tw^{(j)} = \delta_{ij}, \qquad \xi_i \geq 0, \tag{64}$$

where $i, j = 1, \ldots m$.

The following results show that the eigenvalues and eigenvectors of Σ and W are closely related.

Lemma B.1. *When $\lambda_i > 0$, $Mv^{(i)}$ is an eigenvector of W with eigenvalue λ_i. Also $\|Mv^{(i)}\|^2 = \lambda_i$.*

Proof.

$$W(Mv^{(i)}) = MM^TMv^{(i)} = M\Sigma v^{(i)} = M(\lambda_i v^{(i)}) = \lambda_i(Mv^{(i)}). \tag{65}$$

Consider now the norm of $Mv^{(i)}$:

$$(Mv^{(i)})^T(Mv^{(i)}) = (v^{(i)})^TM^TMv^{(i)} = (v^{(i)})^T\Sigma v^{(i)} = \lambda_i. \tag{66}$$

Note that when $\lambda_i = 0$, the norm of $Mv^{(i)}$ is zero, which means $Mv^{(i)}$ is the zero vector. □

Lemma B.2. *When $\xi_i > 0$, $M^Tw^{(i)}$ is an eigenvector of Σ with eigenvalue ξ_i. Also $\|M^Tw^{(i)}\|^2 = \xi_i$.*

Proof.

$$\Sigma(M^Tw^{(i)}) = M^TMM^Tw^{(i)} = M^TWw^{(i)} = M^T(\xi_i w^{(i)}) = \xi_i(M^Tw^{(i)}). \tag{67}$$

Consider now the norm of $M^Tw^{(i)}$:

$$(M^Tw^{(i)})^T(M^Tw^{(i)}) = (w^{(i)})^TMM^Tw^{(i)} = (w^{(i)})^TWw^{(i)} = \xi_i. \tag{68}$$

Note that when $\xi_i = 0$, the norm of $M^Tw^{(i)}$ is zero, which means $M^Tw^{(i)}$ is the zero vector. □

Corollary B.2.1. *Σ and W share the same set of non-zero eigenvalues, of which there are at most $min(m, n)$.*

Proof. Every non-zero eigenvalue of Σ is also an eigenvalue of W (see B.1). But can W have more non-zero eigenvalues than Σ? No, because every non-zero eigenvalue of W is also an eigenvalue of Σ (see B.2). Since Σ and W share the same set of non-zero eigenvalues, there cannot be more than $\min(m, n)$ of them. For instance, if $m = 2$ and $n = 3$, W can have at most two non-zero eigenvalues (because it's 2×2), and since Σ cannot have more non-zero eigenvalues than W, it is guaranteed to have at least one zero eigenvalue, which means it is not invertible. □

Let $\lambda_1, \ldots, \lambda_k$ ($k \leq \min(m, n)$) denote the strictly positive eigenvalues of Σ and W. Also let $v^{(1)}, \ldots, v^{(k)}$ be the corresponding eigenvectors of Σ, and $w^{(1)}, \ldots, w^{(k)}$ those of W. It follows from B.1 that

$$w^{(i)} = \frac{1}{\sqrt{\lambda_i}}Mv^{(i)} \qquad i = 1, \ldots, k. \tag{69}$$

Let $v^{(k+1)}, \ldots, v^{(n)}$ and $w^{(k+1)}, \ldots, w^{(m)}$ be orthonormal eigenvectors corresponding to zero eigenvalues of Σ and W, respectively. Now we define V, D, and U as follows

$$\begin{aligned} V_{ij} &= v_i^{(j)}, \qquad i, j = 1, \ldots, n \\ U_{ij} &= w_i^{(j)}, \qquad i, j = 1, \ldots, m \\ D_{ii} &= \sqrt{\lambda_i}, \qquad i = 1, \ldots, k, \end{aligned}$$

and $D_{ij} = 0$ otherwise. Note that columns of V are eigenvectors of Σ, columns of U are eigenvectors of W, and D is zero everywhere except for the first k diagonal elements. (Check for yourself that $V^T V = 1_n$ and $W^T W = 1_m$, and that from these it follows $VV^T = 1_n$ and $WW^T = 1_m$.) Finally:

$$\begin{aligned}
[UDV^T]_{ij} &= \sum_{i'=1}^{m} U_{ii'} \sum_{j'=1}^{n} D_{i'j'} V_{jj'} \\
&= \sum_{i'=1}^{k} w_i^{(i')} \sum_{j'=1}^{k} \sqrt{\lambda_{j'}} \delta_{i'j'} v_j^{(j')} \\
&= \sum_{i'=1}^{k} w_i^{(i')} \sqrt{\lambda_{i'}} v_j^{(i')} \\
&= \sum_{i'=1}^{k} [Mv^{(i')}]_i v_j^{(i')} \\
&= \sum_{i'=1}^{k} [Mv^{(i')}]_i v_j^{(i')} + \sum_{i'=k+1}^{n} [Mv^{(i')}]_i v_j^{(i')} \\
&= \sum_{i'=1}^{n} [Mv^{(i')}]_i v_j^{(i')} \\
&= [MVV^T]_{ij} \\
&= M_{ij}.
\end{aligned}$$

The second equality uses the definitions of W, V, D, and the fact that $D_{ij} = 0$ for all $i, j > k$. The fourth equality uses (69), and the fifth equality follows from the fact that $Mv^{(i)} = 0$ for all $i > k$ (see (B.1)).

C Direction of steepest descent

Here we present a formal proof that $J(\theta)$ decreases the fastest (locally) in the opposite direction of the gradient ∇J. Consider a vector n and its corresponding unit vector $\hat{n} = n/\sqrt{n \cdot n}$. The rate of change of J at point θ along $\hat{n}$ is

$$\sum_{j=1}^{N} \hat{n}_j \frac{\partial J}{\partial \theta_j} = n \cdot \nabla J.$$

(You should convince yourself of this, by considering $J(\theta_1 + \epsilon \hat{n}_1, \ldots, \theta_N + \epsilon \hat{n}_N)$.) We'd like to find the direction $\hat{n}$ along which the rate of change of J is the highest. This is equivalent to minimizing the function

$$f(n) = \sum_{j=1}^{N} \frac{n_j}{\sqrt{n \cdot n}} \frac{\partial J}{\partial \theta_j} = \frac{n \cdot \nabla J}{\sqrt{n \cdot n}}.$$

Let's compute ∇f:

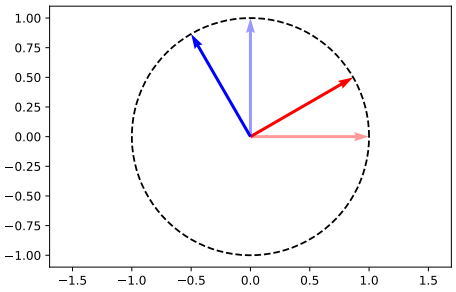
$$\frac{\partial f}{\partial n_i} = \sum_{j=1}^{N} \left(\frac{\delta_{ij}}{\sqrt{n \cdot n}} - \frac{n_i n_j}{(n \cdot n)^{3/2}} \right) \frac{\partial J}{\partial \theta_j} \tag{70}$$

$$= \frac{1}{\sqrt{n \cdot n}} \left(\frac{\partial J}{\partial \theta_i} - \frac{n \cdot \nabla J}{n \cdot n} n_i \right). \tag{71}$$

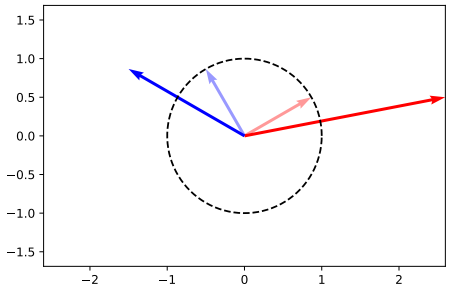
It can be checked that $\nabla f = 0$ when $n = \pm \nabla J$, which means either $+\nabla J$ or $-\nabla J$ minimizes f. To figure out which one, we plug both back into f:

$$f(\pm \nabla J) = \pm \sqrt{\nabla J \cdot \nabla J},$$

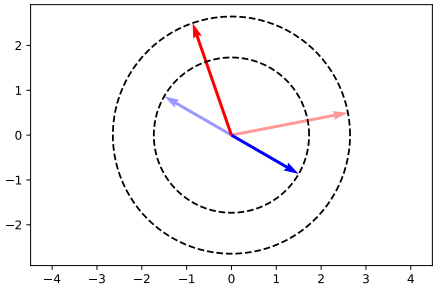
which tells us $n = -\nabla J$ is the direction along which J decreases the fastest.



(a) First step: applying V^T to unit vectors u_x (light red) and u_y (light blue). $V^T u_x$ and $V^T u_y$ are the darker red and blue arrows, respectively.



(b) Second step: applying D to $V^T u_x$ (light red) and $V^T u_y$ (light blue). $DV^T u_x$ and $DV^T u_y$ are the darker red and blue arrows, respectively.



(c) Third step: applying U to to $DV^T u_x$ (light red) and $DV^T u_y$ (light blue). $UDV^T u_x$ and $UDV^T u_y$ are the darker red and blue arrows, respectively.

Figure 18: Applying SVD of M step by step. Code used to generate this plot can be found here.