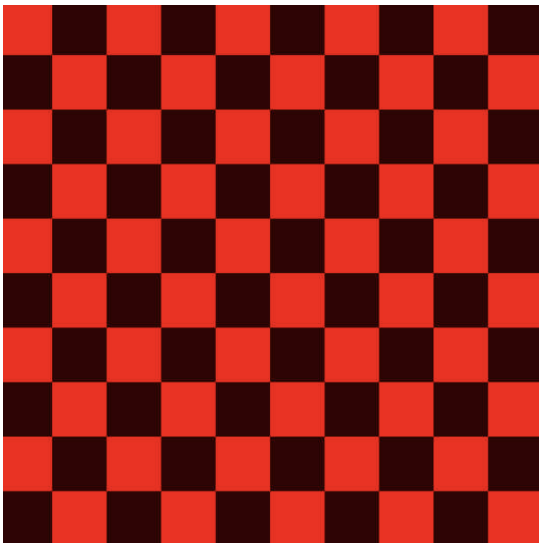
SUPERVISED LEARNING

RITIK JAIN



A 10 × 10 checkerboard. [1]

Date: July, 2024.

[1]Arguably the first machine learning algorithm, designed in 1959 by MIT computer scientist Arthur Samuel, was made to learn the game of checkers. You can find his original paper here.

Contents.

1. Introduction.

An innate characteristic of the human species is our ability to classify and understand the world around us based on the information gathered by our senses. We've all likely observed that stop signs tend to be painted red with white lettering, and that it tends to be warmer outside between the months of June and September.[2] We're also able to pick up more fuzzy patterns, like the fact that red tends to denote negatives, as in red lettering on negative bank balances, red lights, red error messages, etc.

As the world has grown more complex, so have the patterns we are required to understand. People 1000 years ago did not have to understand the effect of the money supply on inflation! Such problems motivated the development of statistics in the latter half of the second millennium, our first systemic attempt at understanding large-scale data. The field of *machine learning* is the most important offshoot of statistics – characterized by dynamic computer-powered models, which are capable of handling massive datasets and continuously improving over time. This new field has led to breakthroughs in almost every discipline imaginable – medicine, finance, biology, mathematics, and even literature! In this chapter, we will discuss a few algorithms from an extremely popular subfield of machine learning – supervised learning.

[2] If you live on the Northern Hemisphere!

1. Introduction

Here are two prototypical questions which can be answered using supervised learning.

Question 1.1. Determine whether or not a $\$100$ bill is real or counterfeit, given 1000 bills which are known to be real, and 1000 bills which are known to be counterfeit.

Question 1.2. Based on a dataset consisting of the square footage and price of 100 NYC apartments, how much should an apartment in New York cost based on its square footage?

The key goal of supervised learning is figuring out how to sort/label new data accurately, given some already-labelled data. In Question 1.1, we aim to label bills as real or counterfeit, and Question 1.2, we aim to label apartments with an accurate price, based on their square footage. However, the two questions involve different types of sorting. In the first case, the data is sorted into two classes – real or counterfeit. Problems such as these, where there are finitely many possible labels for the data, are called *classification problems*. In the second case, there are (theoretically) infinite possible prices (labels) for a given apartment. Such problems are called *regression problems*.

Some mathematical formalization is needed to investigate supervised learning more rigorously.

Definition 1.3. A *dataset* X is a finite subset of $\mathbb{R}^n$. Each element $\mathbf{x} \in X$ is called a *datapoint*. Each entry in a given $\mathbf{x} \in X$ is called a *feature* of $\mathbf{x}$. The dataset X is typically normalized such that the datapoints in X lie inside the unit sphere in $\mathbb{R}^n$, which can often aid in avoiding numerical issues.

We now formalize the notion of classifying data.

Definition 1.4. Given a dataset X which can be partitioned into subsets $\{C_i\}_{i \in I}$, a set $L \subset \mathbb{R}$ is called the *label space* of X if each subset $C_i \subset X$ can be associated with a unique $l \in L$.

Example 1.5. Every Amazon customer can be assigned a vector/point in $\mathbb{R}^3$, containing their age, annual income in thousands, and their annual expenditure on the website. The set of all such vectors V is a dataset, which can be normalized to lie within the unit sphere in $\mathbb{R}^3$ by subtracting the mean and dividing the maximum from each feature of every $\mathbf{v} \in V$.

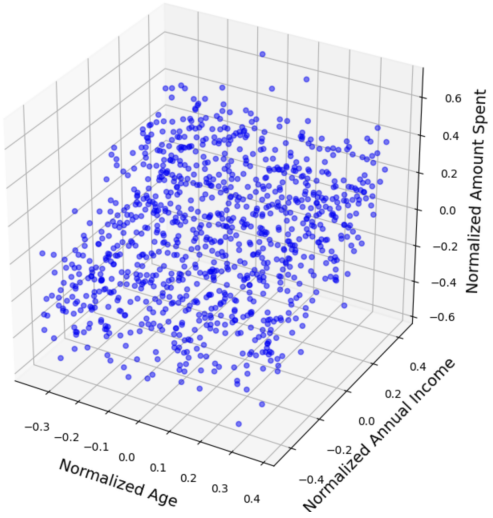


FIGURE 1. Normalized dataset of Amazon customers.

Example 1.6. In Question 1.1, the two classes are counterfeit and real. In such *binary* classification problems, the label space is usually taken to be $\{0, 1\}$. In this case, 0 may represent counterfeit, and 1 may represent real. In Question 1.2, supposing that the prices of NYC apartments range from $\$100\text{k} - \$100\text{M}$, a possible label space is the closed interval $L = [0.1, 100]$, where each $l \in L$ represents a possible apartment price in millions.

In general, L is finite in classification problems, and L is infinite in regression problems.

For a dataset X with a label space L, let $Y \subset X \times L$ be our given *labelled data*. The problem of labelling new data accurately can be formalized as the problem of finding a simple function $f : X \to L$ whose graph "agrees with" the labelled data Y. We call f a *model* of the data.

Example 1.7. Let V be the dataset of Amazon customers from before. Suppose we want to predict what rating a customer will give a particular ad from 1 to 5, given ratings from 1000 customers S on that ad. The initial information is the labelled data $Y = S \times \{1, 2, 3, 4, 5\}$. Then, we can model the ratings with a function $f : V \to \{1, 2, 3, 4, 5\}$, will take a customer as input, and output an accurate prediction for the rating they would give.

Given a dataset X with a label space L, and labelled data $Y \subset X \times L$, what should a good model f look like?

First, f should "agree" with Y. That is, for any $\mathbf{x} \in X$, $(\mathbf{x}, f(\mathbf{x})) \in \mathbb{R}^{n+1}$ should lie somewhere near the elements of Y whose first entry is $\mathbf{x}$. The model f should also be simple – meaning that it should be smooth, and its graph shouldn't have any unnecessary wiggles or curves. In machine learning terminology, we want to avoid *underfitting* (high bias) in the first case, and *overfitting* (high variance) in the second.

One common method for testing the fit of a given supervised learning model is *k-fold cross-validation*, which is done as follows. First, we randomly select k disjoint subsets $s_1, \cdots, s_k$ of Y, called *folds*. Then, for $1 \leq i \leq k$, we train the model on $s_1, \cdots s_{i-1}, s_{i+1}, \cdots, s_k$, fit the result to the fold s_i, and evaluate its accuracy. After averaging the results, we get a comprehensive accuracy score for the model, which tells us if it underfits or overfits the labelled data. The algorithm is most commonly performed with $k = 5$, in which case it is called *five-fold cross-validation*. It can be implemented in Python as follows.

```
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris

data = load_iris() # Load data.
X, y = data.data, data.target

model = # Insert your favorite model here.
cv_scores = cross_val_score(model, X, y, cv=5) # Five-fold cross-validation.

print(f'Cross-validation scores: {cv_scores}') # Output the results.
print(f'Average cross-validation score: {np.mean(cv_scores):.2f}')
```

In the next sections, we will go over some common models for finding models for regression and classification problems. We also discuss gradient descent, an important algorithm for implementing these models in practice.

2. LINEAR REGRESSION

Linear regression, a generalization of the "line-of-best-fit" in higher dimensions, is today one of the most widely-used regression models in machine learning for its simplicity and robustness. For a dataset $X \subset \mathbb{R}^n$, with a subset of m labelled datapoints $Y = \{(\vec{x}^{(1)}, y^{(1)}), \cdots, (\vec{x}^{(m)}, y^{(m)})\} \subset \mathbb{R}^{n+1}$, linear regression takes the model f to be the plane-of-best-fit to the points in Y. This function f is called the *regression plane*.

Note that, in the $n = 1$ case, f is simply the line-of-best-fit to the labelled datapoints in $\mathbb{R}^2$, also called the *regression line*. Planes have no wiggles in their graph, so overfitting is automatically avoided, and in the case that the data enjoys fairly linear relationships between its different features, underfitting is also negligible. The full assumptions for the use of linear regression are discussed in Appendix A.

Let's start the derivation of linear regression! First, f is a linear function, so it is of the form

$$f(x_1, \cdots, x_n) = w_0 + w_1 x_1 + \cdots + w_n x_n \tag{1}$$

for some $w_0, w_1, \cdots w_n \in \mathbb{R}$. The w_i are called the *weights* of the features x_i. Letting $\vec{w} = \langle w_1, \cdots, w_n \rangle$ and $\vec{x} = \langle x_1, \cdots, x_n \rangle$, we can rewrite (1) as

$$f(\vec{x}) = w_0 + \vec{w} \cdot \vec{x},$$

where $\cdot$ denotes the dot product.[3]

How can we choose the "right" weights w_0 and $\vec{w}$? One method of doing so is defining a *cost function*, which measures the error of a generic regression line f to the data in terms of its weights w_0, $\vec{w}$, such the ideal weights w_0, $\vec{w}$ minimize the cost function. We will derive the cost function for linear regression below.

Suppose there are m training examples in $Y \subset \mathbb{R}^{n+1}$, denoted by

$$Y = \{(x^{(1)}, y^{(1)}), \cdots (x^{(m)}, y^{(m)})\}.$$

The error of our regression plane at a fixed $(x^{(i)}, y^{(i)})$ is the difference

$$f(x^{(i)}) - y^{(i)} = (w_0 + \vec{w} \cdot x^{(i)}) - y^{(i)}. \tag{2}$$

Note that (2) is a function of $\vec{w}$. In order to ensure that (2) is convex, or has a unique minimum, and is differentiable, we take the square to obtain

$$(f(x^{(i)}) - y^{(i)})^2.$$

What would be the best way to measure the total error over each $(x^{(i)}, y^{(i)}) \in Y$? The most straightforward method is taking the average error over each each point in Y

[3]The technique of using linear algebra to represent data, called *vectorization*, can often speed up numerical computation.

$$\frac{1}{m}\sum_{i=1}^{m}\left(f(x^{(i)})-y^{(i)}\right)^2=\frac{1}{m}\sum_{i=1}^{m}\left((w_0+\vec{w}\cdot x^{(i)})-y^{(i)}\right)^2. \tag{3}$$

Note that (3) is a function of w_0 and the weight vector $\vec{w}$. Moreover, the w_0 and vector $\vec{v}$ which minimizes the average error (3) corresponds to the ideal choice of coefficients for f in (1) as desired. Now, we are all set to define the cost function.

Definition 2.1. Define X, Y, and f as above. The *cost function* of the regression plane f is the function

$$J:\mathbb{R}^{n+1}\to\mathbb{R},\quad J(w_0,\vec{w})=\frac{1}{2m}\sum_{i=1}^{m}\left((w_0+\vec{w}\cdot x^{(i)})-y^{(i)}\right)^2. \tag{4}$$

The expression for the cost function is exactly the same as (3), except divided by 2. While this simplifies matters when taking partial derivatives, the global minima of (3) and (4) are identical.

Since the cost function J is the sum of squared terms, it is convex, and therefore has a unique global minimum $(w_0^*,\vec{w}^*)$. Again, the vector $(w_0^*,\vec{w}^*)$ corresponds to the ideal choice of weights for our regression line f, thus

$$f(\vec{x})=w_0^*+\vec{w}^*\cdot\vec{x}$$

is the desired regression plane to the labelled data Y.

Example 2.2. Suppose you conduct a study, measuring the heights and weights of 30 individuals, and gather the data pictured below.

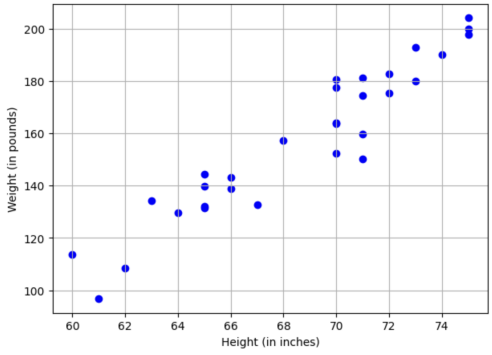


FIGURE 2. The height and weight of 30 individuals.

You want to use this data to predict a person's weight based on their height, and seeing that the data follows a roughly linear pattern, you decide to use linear regression.

Since the data has $n = 1$ features, the regression line f has the form

$$f(x) = w_0 + w_1 x$$

for some $w_0, w_1 \in \mathbb{R}$. For $1 \leq i \leq 30$, let $x^{(i)}$ and $y^{(i)}$ denote the weight and height of the ith individual respectively. Then, the cost function for the regression line is

$$J(w_0, w_1) = \frac{1}{60} \sum_{i=1}^{30} \left(\left(w_0 + w_1 x^{(i)} \right) - y^{(i)} \right)^2 . \tag{5}$$

By taking partial derivatives, we find that J is minimized when $w_0 \approx -273.47$ and $w_1 \approx 6.28$, thus

$$f(x) = -273.47 + 6.28x,$$

pictured below.

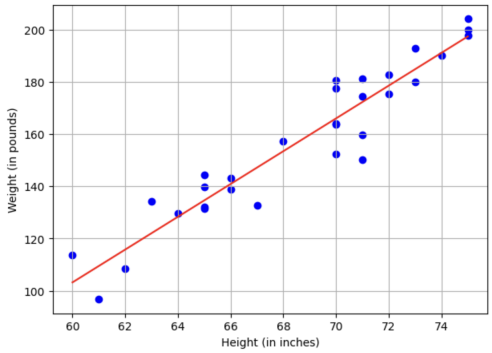


FIGURE 3. The height and weight of 30 individuals, with the regression line.

Note that f can be used to predict the weight of an individual, given their height. For example, according to the regression line, somebody who is five-foot nine, or 69 inches tall, should weigh

$$f(69) = -273.47 + 6.28(69) = 159.85 \text{ pounds},$$

which is a reasonable estimate.

But how can we actually minimize a given cost function? In the next section, we will review *gradient descent*, a numerical method for finding the minima of smooth multivariable functions. Afterwards, we will revisit the above example in greater detail.

3. Gradient descent

Imagine waking up one cold, foggy night on the hill pictured below.

FIGURE 4. 13th-century terraces in Shexian, China.

You see some flickering lights resembling a fire at the bottom of the hill, but can't see clearly further than a few steps ahead of yourself! How can you effectively descend the hill and reach the fire?

To start, you could feel if there's a slant from where you're standing, and take a step in the steepest direction downwards. Knowing that you're on a terrace, this step should be relatively small to avoid falling over! Then, you can repeat this procedure step-by-step until you eventually don't feel any slant underneath your feet. At this point, you'll be safely on the ground, and hopefully near the fire.

Gradient descent is essentially a mathematical version of this method, being used to numerically approximate the minima of convex differentiable functions $\mathbb{R}^n \to \mathbb{R}$. We will sketch how it works below.

Recall the following definition from multivariable calculus:

Definition 3.1. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function in n variables $x_1, \cdots, x_n$, such that for all $1 \leq i \leq n$, the partial derivative $\frac{\partial f}{\partial x_i}$ exists. The *gradient* of f is defined as the function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$,

$$\nabla f = \left\langle \frac{\partial f}{\partial x_1}, \cdots, \frac{\partial f}{\partial x_n} \right\rangle.$$

The gradient is a key part the algorithm because, for any point $\mathbf{p} \in \mathbb{R}^n$, $-\nabla f(\mathbf{p})$ is the "direction of fastest decrease" of f at $\mathbf{p}$.[4] By starting at some initial point and moving step-by-step down the direction of $-\nabla f(\mathbf{p})$, we can obtain a good approximation of the minimum of f after sufficiently many iterations.

[4]This fact is proved in Appendix B.

The iterative equation for gradient descent is as follows. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a convex, differentiable function, and let $\mathbf{x_0} \in \mathbb{R}^n$ be an arbitrary point. For all $i \in \mathbb{Z}_{>0}$, the $(i+1)$th step of the gradient descent is defined as

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i),$$

where $\alpha \in (0, \infty)$ is a fixed step size, or *learning rate*. Going back to the analogy at the start of the section, this process can be thought of as travelling down a hill f from a starting position $\mathbf{x_0}$ by taking α-sized steps down the steepest direction $-\nabla f$.

Provided that ∇f has continuous partial derivatives[5], with an appropriate choice of α, the sequence

$$\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_i, \cdots$$

is guaranteed to converge to the global minimum of f. If the learning rate α is larger, the algorithm will move more aggressively down the graph of f. On the one hand, this may result in a quicker convergence, but on the other, it can cause us to overshoot the minimum, and end up oscillating around it. If α is smaller, the algorithm will be more cautious, yielding a slower but more guaranteed convergence. In order to balance speed and precision, we often start with a larger learning rate α, run the gradient descent, and reduce α until convergence occurs.

We can apply gradient descent to numerically minimize the cost function of a given regression plane to a set of datapoints. Let's take a closer look at Example 2.2. Our cost function, given in (5), was of the form

$$J(w_0, w_1) = \frac{1}{60} \sum_{i=1}^{30} \left(\left(w_0 + w_1 x^{(i)} \right) - y^{(i)} \right)^2.$$

Taking partial derivatives, we have

$$\frac{\partial J}{\partial w_0} = \frac{1}{30} \sum_{i=1}^{30} \left(\left(w_0 + w_1 x^{(i)} \right) - y^{(i)} \right),$$

$$\frac{\partial J}{\partial w_1} = \frac{1}{30} \sum_{i=1}^{30} \left(\left(w_0 + w_1 x^{(i)} \right) - y^{(i)} \right) x^{(i)}.$$

therefore

$$\nabla J(w_0, w_1) = \left\langle \frac{1}{30} \sum_{i=1}^{30} \left(\left(w_0 + w_1 x^{(i)} \right) - y^{(i)} \right), \frac{1}{30} \sum_{i=1}^{30} \left(\left(w_0 + w_1 x^{(i)} \right) - y^{(i)} \right) x^{(i)} \right\rangle.$$

Note that J is convex and ∇J has continuous partial derivatives, thus gradient descent is applicable. Starting at a fixed $\mathbf{x}_0 \in \mathbb{R}^2$, the $(i+1)$th step is given by

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla J(\mathbf{x}_i).$$

[5]More generally, ∇f must be *Lipschitz continuous*.

To demonstrate how the algorithm might be carried out in practice, let's implement gradient descent in pseudocode.

```
def J(w_0, w_1): # This is our cost function
    # Compute and sum the squared errors at each datapoint.
    J(w_0, w_1) = sum i between 1 and 30:
        ((w_0 + w_1 * heights[i]) - weights[i])^2 / 2n
    return J(w_0, w_1) # Return the cost function.

def d_dw0(w_0,w_1): # This is the partial derivative of J with respect to w_0.
    # Compute the partial derivative.
    d_dw0 = sum i between 1 and 30:
        ((w_0 + w_1 * heights[i]) - weights[i]) / n
  return d_dw0

def d_dw1(w_0,w_1): # This is the partial derivative of J with respect to w_1.
    # Compute the partial derivative.
    d_dw1 = sum i between 1 and 30:
        ((w_0 + w_1 * heights[i]) - weights[i]) * heights[i] / n
  return d_dw1
```

Above, we computed the cost function $J(w_0, w_1)$ with its partial derivatives with respect to w_0 and w_1. Using these functions, we will run the gradient descent.

```
def grad_descent(d_dw0, d_dw1, learning_rate, num_iterations, w_0, w_1):

    for i between 1 and num_iterations:  # This loop will run the gradient descent.
        # Update w_0, w_1 simultaneously.
        temp_w_0 = w_0 - learning_rate * d_dw0(w0,w1)
        temp_w_1 = w_1 - learning_rate * d_dw1(w0,w1)
        w_0 = temp_w_0
        w_1 = temp_w_1

    return w_0, w_1 # Return the final values of w_0, w_1.
```

The process of gradient descent on (5) with $100,000$ iterations, a learning rate of $\alpha = 0.0003$, from the initial point $\mathbf{x}_0 = (0, 0)$, is visualized below. Every dot represents the values for (w_1, w_0) after each thousandth iteration.

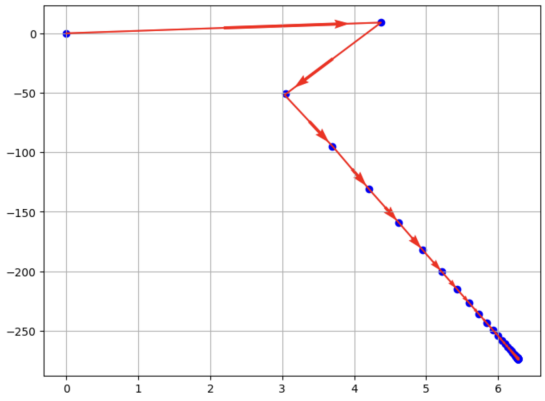


FIGURE 5. Convergence of the gradient descent algorithm.

As pictured in Figure 5, with each iteration, (w_1, w_0) approaches $(6.28, -273.47)$, and the changes become increasingly small. This is a good sign that the algorithm is converging to the true minimum of the cost function J as desired.

Remark 3.2. The above algorithm is actually a specific type of gradient descent called *batch gradient descent*. The name is justified since it considers the error at each point in the entire "batch" of training examples. There are two other important variations called *minibatch gradient descent* and *stochastic gradient descent*.

The first algorithm, minibatch gradient descent, only considers the error with respect to a "minibatch" of k randomly chosen training examples at each step. Stochastic gradient descent is precisely minibatch gradient descent with $k = 1$. Batch gradient descent is slow but accurate, stochastic gradient descent is fast but sometimes inaccurate, and minibatch gradient descent is a compromise between the two.

4. Least-squares regression

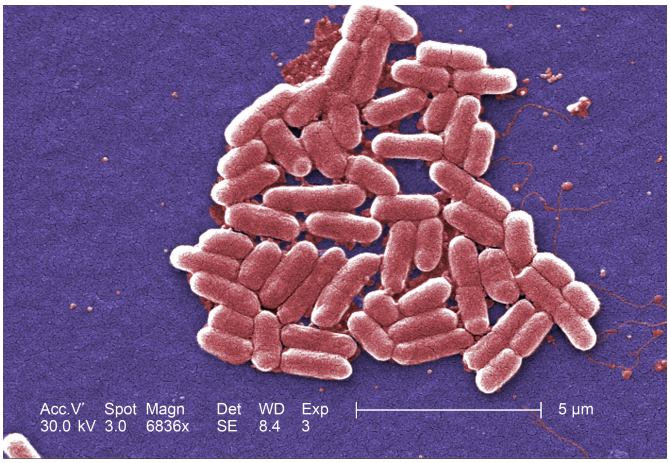


FIGURE 6. E. coli cluster, magnified 6836x. To be discussed in Example 4.1.

While linear regression is a powerful and widely applicable model for many regression problems, it is not an accurate fit for data which exhibit "non-linear" relationships. In order to model such data, we consider a generalization of linear regression called *least-squares regression*.

Recall the basic setup for linear regression. We are given a dataset $X \subset \mathbb{R}^n$, with a subset of m labelled datapoints $Y = \{(\vec{x}^{(1)}, y^{(1)}), \cdots, (\vec{x}^{(m)}, y^{(m)})\} \subset \mathbb{R}^{n+1}$, and want to find the "ideal" weights $w_0, \vec{w} = \langle w_1, \cdots, w_n \rangle$ such that the regression plane

$$f : \mathbb{R}^n \to \mathbb{R}, \qquad f(\vec{x}) = w_0 + \vec{w} \cdot \vec{x}$$

is a good fit to labelled data Y. We did this by minimizing the cost function (4)

$$J : \mathbb{R}^{n+1} \to \mathbb{R}, \quad J(w_0, \vec{w}) = \frac{1}{2m} \sum_{i=1}^{m} \left(f(x^{(i)}) - y^{(i)} \right)^2,$$

by using gradient descent. Using the framework of the cost function, however it is possible to fit all sorts of curves and surfaces to our data, regardless of whether or not they are linear. For example, suppose our dataset X has $n = 1$ features. If we were using linear regression, the regression line would be taken to be $f(x) = w_0 + w_1 x$, where (w_0, w_1) is the minimum of the cost function

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

But what if the labelled data $Y \subset \mathbb{R}^2$ looked more like a parabola than a straight line? In this case, we could take our model to be of the form $f(x) = w_0 + w_1 x + w_2 x^2$, where

(w_0, w_1, w_2) minimizes the cost function

$$J(w_0, w_1, w_2) = \frac{1}{2m} \sum_{i=1}^{m} \left(\left(w_0 + w_1 x^{(i)} + w_2 x^{(i)^2} \right) - y^{(i)} \right)^2.$$

By the same reasoning as Section 2, the resulting model $f(x)$ will be a good fit for the data. In general, the process of minimizing the cost function of a regression model f is called *least-squares regression*. In the special case that f is linear, it is called *ordinary least-squares regression*. While least-squares regression is more computationally intensive when f is nonlinear, it can provide a better model for our data.

Example 4.1. Suppose you want to model the average growth rate of 50 E. coli cells over the course of one hour, grown in a lab. In order to gather some data, you set up a Petri dish containing 50 E. coli cells, and record the number of cells every two minutes.

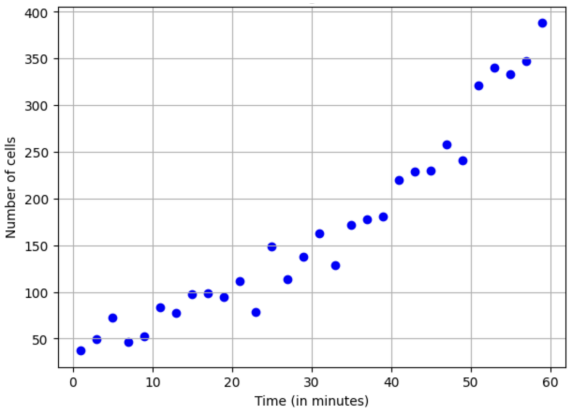


FIGURE 7. Number of E. coli cells in the Petri dish over time.

The problem of modelling the growth rate can be formalized as a regression problem. We can consider the closed interval $[0, 60]$ to be our dataset, and the times for which we know the number of E. coli cells as the labelled data. Since the average number of E. coli cells in the dish is always a positive real number, we can take $(0, \infty)$ to be the label space.

As shown in Figure 7, the number of cells seems to be growing exponentially, and doubling every twenty minutes. Will linear regression give a good model for this data? Your intuition should be telling you "no", since the data very much does not look like a line. For the sake of curiosity, we can perform an ordinary least-squares regression anyways, and obtain

$$f(t) = 1.76 + 5.53t \tag{6}$$

as our regression line.

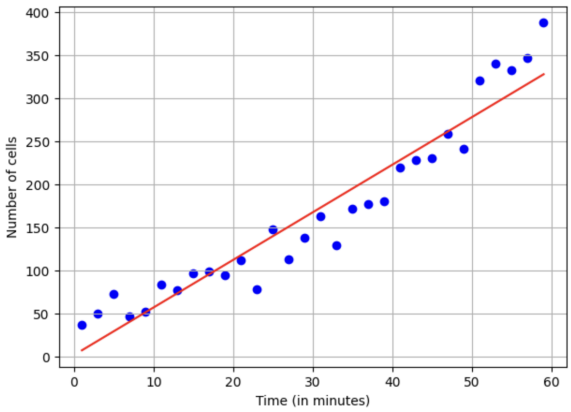


FIGURE 8. Number of E. coli cells, with the regression line.

As shown in Figure 8, the regression line underfits the data at several points. For example, even though there are roughly 50 cells in the dish at the time $t = 0$, our line predicts that there are only $f(0) = 1.76$ cells!

Since our data is growing exponentially, a more appropriate model might be, you guessed it, an exponential function! In particular, we can take $f(x) = Ae^{bt}$ as our regression model. Since $f(0) = A = 50$, we can simply take

$$f(t) = 50e^{bt}$$

to be our regression model, where b is an unknown parameter. Now, there $m = 30$ labelled datapoints, so our cost function $J : \mathbb{R} \to \mathbb{R}$ will be of the form

$$J(b) = \frac{1}{60}\sum_{i=1}^{30}\left(f(x^{(i)}) - y^{(i)}\right)^2 = \frac{1}{60}\sum_{i=1}^{30}\left(50e^{bx^{(i)}} - y^{(i)}\right)^2.$$

The graph of our cost function on the interval $b \in [0, 0.05]$ is shown below.

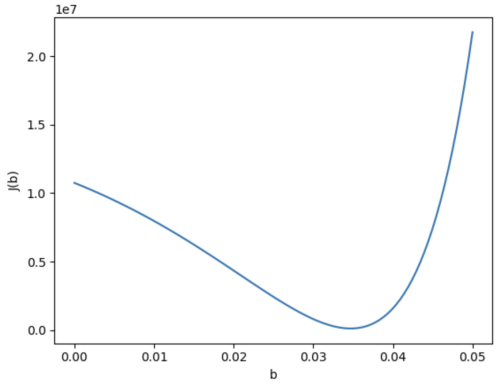


FIGURE 9. Graph of the cost function.

Since $J(b)$ is convex for the same reasons discussed in Section 2, it is strictly decreasing before attaining its minimum, and strictly increasing afterwards. From Figure 9, the point $b \approx 0.035$ is therefore the global minimum of the cost function J, hence our regression model is given by

$$f(t) = 50e^{0.035t}.$$

To see if $f(t)$ is a good fit, let's plot it with our data.

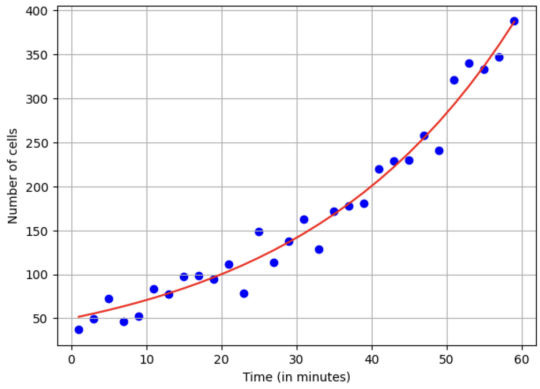


FIGURE 10. Number of E. coli cells, with $f(t)$.

It appears that the model $f(t)$ is indeed a good match for our data.

Remark 4.2. In some cases, it is not possible to visualize or predict what the regression model should look like. In such cases, we can take it to be some combination of polynomials and exponential functions. For example, consider a dataset X with a single feature and some labelled data $Y \subset X \times [0, 1]$. We could take the regression model to be

$$f(x) = w_0 + w_1 x + w_2 e^{w_3 x}.$$

Suppose that, after performing least-squares regression, we have

$$f(x) = 10 + 0.005x + 30e^{5x}.$$

Since the weight w_1 for the linear term is very small relative to the coefficients w_2, w_3 for the exponential term, we can take the model to simply be $f(x) = 10 + 30e^{5x}$. While being a bit slower, this method of starting with a regression model containing many terms, then removing terms with small coefficients after performing least-squares regression, can often be useful. For example, if our data has three features, the labelled data will appear in four-dimensional space, making it a bit hard to eyeball!

In the next section, we will take a look at another important supervised learning algorithm, called *k-nearest-neighbors*. As we will see, k-nearest-neighbors requires far fewer assumptions than either linear or least-squares regression, and can be used to solve both regression and classification problems.

5. K-NEAREST-NEIGHBORS

The k-nearest-neighbors (KNN) algorithm is one of the most powerful tools in supervised learning for its elegance and wide applicability. In contrast to least-squares regression, KNN does not require that the labelled data follows any particular pattern to work, and can be easily adjusted to avoid overfitting and underfitting. We will demonstrate how it works for classification problems through an example.

Example 5.1. Suppose we are given a dataset of ninety Delta customers, together with their age, number of miles with Delta, and whether they generally fly in economy, business-class, or first-class. Given the age and number of miles of a set of new customers, how can we best predict which type of ticket they will purchase? First, let's visualize our given data, normalized such that the age and Delta miles of each customer is a real number between zero and one. We additionally represent each customer's preference of flight class by a color: bronze for economy, silver for business, and gold for first-class.

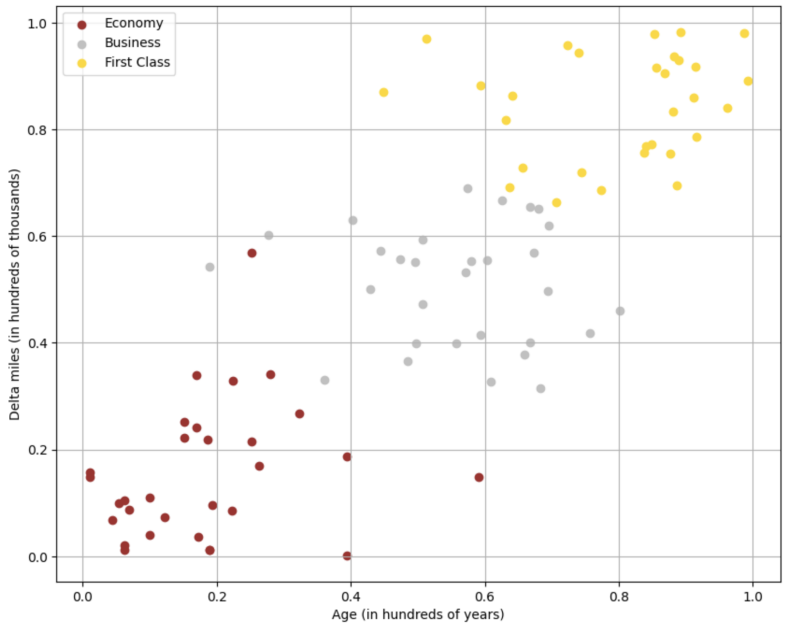


FIGURE 11. Dataset of Delta customers.

Note that there are a finite number of labels; economy, business, and first-class, so this is a classification problem. The k-nearest-neighbors algorithm is one way of solving this, and relies on the following simple idea. For every new customer c, we can look at the k customers which are most similar in age and Delta miles to c, whose travelling preferences are known to us. These are the customer's k-nearest-neighbors. Then, we can decide where to sort c by majority voting–if more than one-third of c's k-nearest-neighbors

generally travel in a particular class, we sort c into that class as well. In the rare case that there is a tie between two or more classes, we can just sort c randomly into one of them.

The algorithm can be implemented in general as follows. Take a dataset $X \subset \mathbb{R}^n$ with a subset Y of labelled datapoints, with label space $\{0, 1, \cdots, \ell\}$ for some positive integer ℓ. For each $\mathbf{x} \in X$, calculate the Euclidean distance[6] from $\mathbf{x}$ to each $\mathbf{y} \in Y$, and choose k-nearest-neighbors $\mathbf{y}_1, \cdots, \mathbf{y}_k \in Y$ with the smallest distance to $\mathbf{x}$. Then, denoting the labels of the $\mathbf{y}_1, \cdots, \mathbf{y}_k$ by $l_1, \cdots, l_k$, we predict the label of $\mathbf{x}$ to be the mode of $l_1, \cdots, l_k$, that is, the most frequently occurring label.

From the KNN algorithm with $k = 4$, for instance, a fifty-year-old customer with 10,000 Delta miles (represented by the point $(0.5, 0, 1)$) is predicted to travel in economy, and a thirty-year-old with 90,000 Delta miles is predicted to travel in first-class.

Now, let's use KNN with $k = 4$ in this example to sort a new batch of 500 customers, which are represented as black points on the graph before they are sorted.

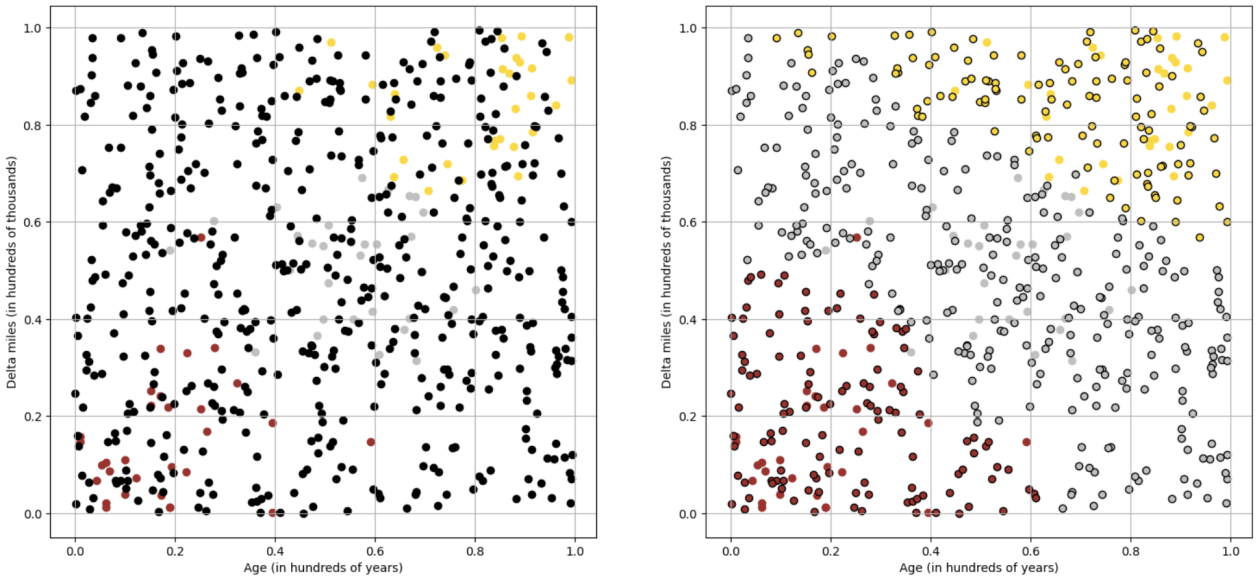


FIGURE 12. Dataset of new customers before and after KNN.

As shown in the above figure, KNN seems to do a good job at classifying new customers, and we can test it rigorously using five-fold cross-validation.

While KNN is most often used for classification problems, the framework can easily be extended to regression problems as follows. Take a dataset $X \subset \mathbb{R}^n$ with a subset Y of labelled datapoints, and let the label space be an interval $L \subset \mathbb{R}$. For each $\mathbf{x} \in X$,

[6]There are also other forms of "distance" which are used for KNN, depending on how the datapoints are distributed in $\mathbb{R}^n$ For example, people sometimes calculate nearest-neighbors using the Manhattan metric or the more general Minowski metric.

calculate the k-nearest-neighbors $\mathbf{y}_1, \cdots, \mathbf{y}_k \in Y$ to $\mathbf{x}$, which have respectively have labels $l_1, \cdots, l_k \in L$. Rather than do a "majority vote", this time, we take the predicted label of $\mathbf{x}$ to be the average

$$\frac{1}{k}\sum_{i=1}^{k} l_k$$

of the labels of the k-nearest-neighbors.

Remark 5.2. The number of nearest-neighbors k can be any positive integer: in general, if k is small, there is a higher risk of overfitting, and if k is large, there is a higher risk of underfitting. In practice, the "right" value of k is usually chosen by trial-and-error.

Remark 5.3. Unlike least-squares regression, note that KNN does not need to be "trained" on the labelled data; for example, no cost functions are being minimized. Instead, KNN computes the predicted labels for each datapoint one-by-one. Therefore, KNN is part of a class of models called *lazy learning* models. Since the entire dataset needs to be stored, or "memorized", lazy learning models are also called *memory-based* models.

In the next section, we will look *logistic regression*, an algorithm is designed especially for solving classification problems with two classes. Such problems are ubiquitous in applications, making logistic regression an essential part of any machine learning practitioner's toolbox.

6. Logistic Regression

Arguably the most important problem in machine learning is that of *binary classification*, the problem of sorting a dataset into two classes. For example, we might want to classify patients as sick or healthy, emails as "spam" or "not spam", or $100 bills as real or counterfeit as in Question 1.1. In general, one class asserts that the data has a certain property, and the other class asserts that it does not. The label space representing the two classes is usually taken to be $\{0, 1\}$.

One nice quirk of binary classification problems is they can be reinterpreted as regression problems with the label space $[0, 1]$. Instead of assigning an unlabelled datapoint x to a class in $\{0, 1\}$, we instead assign it a *probability*, representing chance of x being in class 1, or conversely, one minus the chance of it being in class 0. For example, if x is given the label $0.62 \in [0, 1]$, then x has a 62% chance of being in class 1, and a $100 - 62 = 38\%$ chance of being in class 0. Since this interpretation of binary classification problems allows for a more precise characterization of the data, it often avoids underfitting.

Since understanding logistic regression and its underpinnings is important for a deeper grasp of machine learning, we will derive it mathematically rather than skip straight to an example, despite it requiring some knowledge of probability and statistics. However, skimming the details when necessary is fine, since logistic regression can usually be implemented in code by using a preexisting library.

That being said, let's begin the derivation! For a dataset $X \subset \mathbb{R}^n$, suppose we have a binary-labelled subset

$$Y = \left\{(x^{(1)}, y^{(1)}), \cdots (x^{(m)}, y^{(m)})\right\} \subset X \times \{0, 1\}.$$

Instead of finding a model which assigns a deterministic label $y(\vec{x}) \in \{0, 1\}$ to each datapoint $\vec{x} \in X$, we associate a Bernoulli trial[7] $B(\vec{x})$ to each $\vec{x}$, where

$$B(\vec{x}) = \begin{cases} 0, & y(\vec{x}) = 0 \\ 1, & y(\vec{x}) = 1. \end{cases}$$

The probability $B(\vec{x}) = 1$ is precisely the probability of $\vec{x}$ being assigned a label of 1. We want to model the function

$$p : X \to [0, 1] \qquad p(\vec{x}) = P\left(B(\vec{x}) = 1\right).$$

which assigns each $\vec{x}$ to the probability that it is labelled with 1. Due to its simplicity, linear regression is often the first line of attack. In particular, we could model $p(\vec{x})$ by the regression plane

$$p(\vec{x}) = w_0 + \vec{w} \cdot \vec{x},$$

[7] A Bernoulli trial is a random variable with precisely two outcomes. Think coin flips or passing/failing an exam.

where $\vec{w} = \langle w_1, \cdots, w_n \rangle$ is the vector of weights corresponding to the features $\vec{x} = \langle x_1, \cdots, x_n \rangle$. While this model is simple, it is problematic because $w_0 + \vec{w} \cdot \vec{x}$ is unbounded, but on the other hand, $p(\vec{x})$ is a probability, so it lies strictly on the interval $[0, 1]$. In order to fix this, we apply the so-called *logistic transformation*[8] and obtain the model

$$\log\left(\frac{p(\vec{x})}{1 - p(\vec{x})}\right) = w_0 + \vec{w} \cdot \vec{x}. \tag{7}$$

The right-hand side of (7) is called the *log-odds function*. Note that as $p(\vec{x})$ tends to zero, the log-odds function tends to $-\infty$, as $p(\vec{x})$ tends to one, the log-odds function tends to ∞, and the log-odds function is zero when $p(\vec{x}) = 0.5$. Solving (7) for $p(\vec{x})$, we have

$$p(\vec{x}) = \sigma(w_0 + \vec{w} \cdot \vec{x}), \tag{8}$$

where $\sigma(z)$ is the *sigmoid function*

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{9}$$

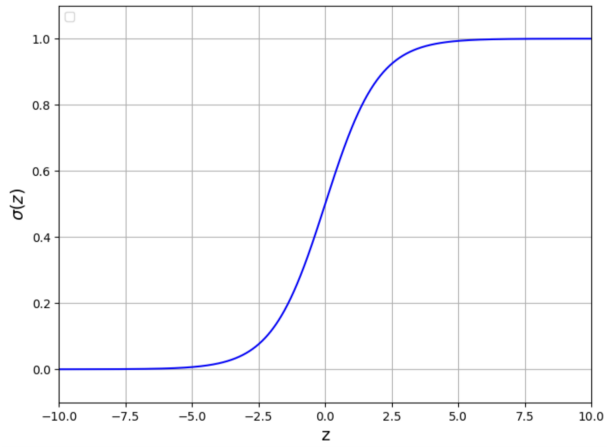


FIGURE 13. Graph of the sigmoid function.

Note that the S-shaped sigmoid function is a viable option for $p(\vec{x})$ since it is bounded on the interval $[0, 1]$, meaning that we can interpret its outputs as probabilities, and we can fit the the graph of the sigmoid function $\sigma(w_0 + \vec{w} \cdot \vec{x})$ to our data by adjusting the parameters $w_0,\ \vec{w}$. In particular, in the $n = 1$ case, we can use w_0 to shift the graph left-and-right, and adjust $w_1 \in (0, \infty)$ to change the steepness of the curve. You can play around with the parameters for yourself here.

In order to find the optimal parameters $w_0,\ \vec{w}$, we will use a statistical method called *maximum likelihood estimation*. Rather than choosing the parameters $w_0,\ \vec{w}$ which minimize a cost function as in Section 4, we choose parameters for which $p(\vec{x}) = p(\vec{x};\ w_0,\ \vec{w})$

[8]Here, $\log$ denotes the base-e natural logarithm.

"agrees" with the labelled data Y. For each labelled datapoint $x^{(i)}$, set

$$P\left(B(x^{(i)}) = y^{(i)}\right) = \begin{cases} p(x^{(i)}), & y^{(i)} = 1, \\ 1 - p(x^{(i)}), & y^{(i)} = 0. \end{cases} \tag{10}$$

Written differently, set

$$P\left(B(x^{(i)}) = y^{(i)}\right) = p(x^{(i)})^{y^{(i)}}(1 - p(x^{(i)})^{1-y^{(i)}}. \tag{11}$$

It is a good exercise to prove that for yourself (10) and (11) are equivalent. In order to find the optimal parameters for $p(\vec{x}) = \sigma(w_0 + \vec{w} \cdot \vec{x})$, we can maximize the *likelihood function*

$$L(w_0,\ \vec{w}) = \prod_{i=1}^{m} P\left(B(x^{(i)} = y^{(i)}\right) = \prod_{i=1}^{m} p\left(x^{(i)}\right)^{y^{(i)}} \left(1 - p\left(x^{(i)}\right)\right)^{1-y^{(i)}},$$

In order to transform the product to a sum to make things easier when differentiating, we can equivalently minimize the negative *log-likelihood function*, given by

$$\tilde{L}(w_0,\ \vec{w}) = -\sum_{i=1}^{m} y^{(i)} \log\left(p\left(x^{(i)}\right)\right) + (1 - y^{(i)}) \log\left(1 - p\left(x^{(i)}\right)\right). \tag{12}$$

Simplifying (12) and plugging in $p(\vec{x}) = \sigma(w_0 + \vec{w} \cdot \vec{x})$, we have

$$\begin{aligned} \tilde{L}(w_0,\ \vec{w}) &= -\sum_{i=1}^{m} y^{(i)} \log\left(\frac{p\left(x^{(i)}\right)}{1 - p\left(x^{(i)}\right)}\right) - \sum_{i=1}^{m} \log\left(1 - p\left(x^{(i)}\right)\right) \\ &= \sum_{i=1}^{m} \log\left(1 + e^{w_0 + \vec{w} \cdot x^{(i)}}\right) - \sum_{i=1}^{m} y^{(i)} \left(w_0 + \vec{w} \cdot x^{(i)}\right). \end{aligned}$$

Taking partial derivatives, for all $1 \leq j \leq n$, we have

$$\frac{\partial \tilde{L}}{\partial w_0} = \sum_{i=1}^{m} \left(p(x^{(i)}) - y^{(i)}\right) \qquad \frac{\partial \tilde{L}}{\partial w_j} = \sum_{i=1}^{m} \left(p(x^{(i)}) - y^{(i)}\right) x_j^{(i)}. \tag{13}$$

Note that $\nabla \tilde{L}$ has continuous partial derivatives, so we can use gradient descent to minimize $\tilde{L}$.[9] After obtaining the approximated minimum $\overline{w_0}, \overline{\vec{w}}$, the logistic regression function is given by

$$p(\vec{x}) = \sigma(\overline{w_0} + \overline{\vec{w}} \cdot \vec{x}) = \frac{1}{1 + e^{-\left(\overline{w_0} + \overline{\vec{w}} \cdot \vec{x}\right)}}.$$

Let's see logistic regression in practice with an example.

[9]Gradient descent is particularly useful here, since it is impossible to find minimum of $\tilde{L}$ algebraically. This is because the $\frac{\partial \tilde{L}}{\partial w_j}$ are all transcendental functions, so we can't set them equal to zero and solve exactly!

Example 6.1. Suppose we want to classify whether or not a person is at risk for skin cancer, based on the diameter or a melenoma lesion on their arm. We are given a dataset of twenty individuals who were previously screened for skin cancer with the size of their lesion, and whether or not they had skin cancer (represented by 0 and 1 respectively). The data is visualized below.

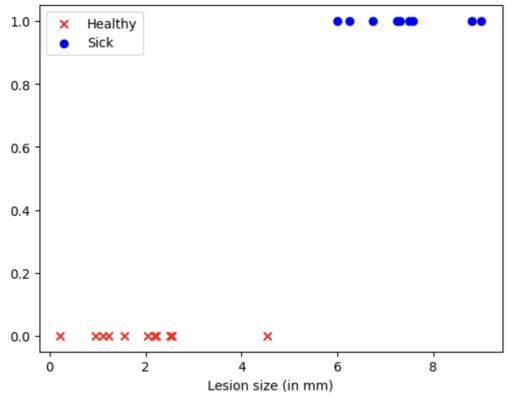


FIGURE 14. Plot of twenty individuals.

We want to estimate the parameters of the function

$$p(x) = \sigma(w_0 + w_1 x) = P(\text{A patient with an } x \text{ mm lesion has skin cancer}).$$

Note that logistic regression will provide a good fit for the data, since the probability of an individual being sick is quite low if their lesion size is below a threshold of ≈ 5 mm, and increases rapidly if the lesion size is larger before levelling off. That is, the probability of an individual being sick as a function of their lesion size should follow an S-shaped sigmoid curve. Rather than go over the mathematical details again, let's see how logistic regression is implemented in Python.

```
import numpy as np
from sklearn.linear_model import LogisticRegression
Y = [(1.23, 0), (2.22, 0), (6.27, 1) ...] # List of labelled data.
X = np.array([x[0] for x in Y]).reshape(-1, 1)
y = np.array([x[1] for x in Y])
model = LogisticRegression() # Train the logistic regression model.
model.fit(X, y)
w_0 = model.intercept_[0] # Extract the parameters.
w_1 = model.coef_[0][0]
# Print the logistic regression function.
print(f'The logistic regression line is given by sigma({w_0:.2f} + {w_1:.2f}x)')
```

After running this code, we have $w_0 \approx -6.90,\ w_1 \approx 1.40$, hence

$$p(x) \approx \sigma(-6.90 + 1.40x) = \frac{1}{1 + e^{(6.90 - 1.40x)}}.$$

We can plot $p(x)$ with the data using the following code:

```
import matplotlib.pyplot as plt
plt.scatter(X, y, color='blue', label='Data points')
x_values = np.linspace(min(X), max(X), 100)
y_values = 1 / (1 + np.exp(-(w_0 + w_1 * x_values)))
plt.plot(x_values, y_values, color='red', label='Logistic regression line')
plt.xlabel('Lesion size (in mm)')
plt.ylabel('Probability')
plt.legend()
plt.show()
```

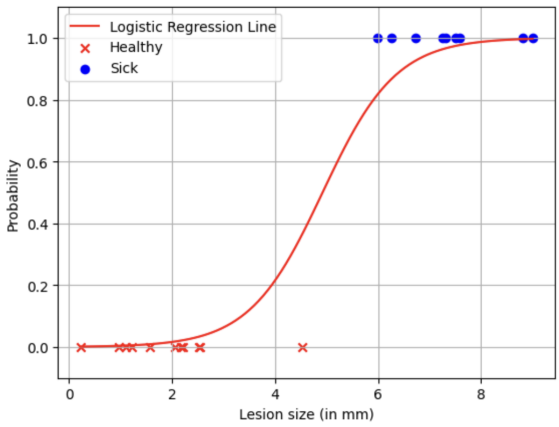


FIGURE 15. Plot of the data, with the logistic regression function $p(x)$.

It appears that the logistic regression line is a good fit for the data. If we wish to be more precise, we can test the fit precisely using five-fold cross-validation.

Let's use the model determine if a lesion size is indicative of skin cancer. First, since $p(3) \approx 0.063$, someone with a 3mm lesion has only an estimated 6.3% chance of having skin cancer. On the other hand, $p(9) = 0.99$, so someone with a 9mm lesion is almost certainly sick. Finally, the threshold lesion size k at which someone has a 50/50 shot of being sick is where $p(k) = 0.5$, or where $6.90 - 1.40k = 0$. Solving, we have $k \approx 4.93\text{mm}$, so those with a lesion size of 5mm or above have some serious cause for concern.

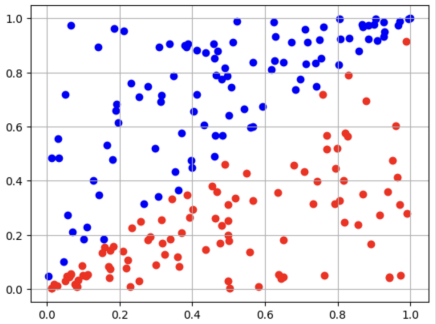
In the next section, we will discuss a set of algorithms called *support vector machines*, which, aside from logistic regression, represent the most popular approach to binary classification problems.

7. Support Vector Machines

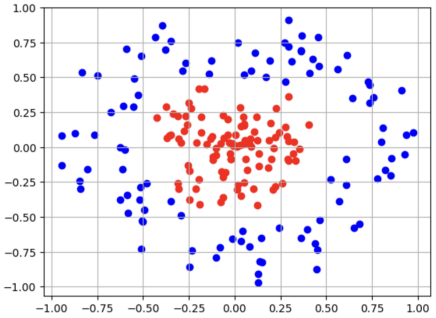
FIGURE 16. Highway separating palm tree species. Source: Pikwizard.

Support Vector Machines (SVMs) are arguably the most effective set of algorithms for solving binary classification problems. Unlike logistic regression, an SVM does not assign datapoints $\mathbf{x} \in \mathbb{R}^n$ to a probability of being in class 0 or class 1; instead, it attempt to find a *decision boundary*, that is, a hyperplane $H \subset \mathbb{R}^n$ which separates the labelled points either class. Based on which side of H that an unlabelled datapoint $\mathbf{x} \in \mathbb{R}^n$ lies on, we assign it to class 0 or class 1. Such algorithms are called *linear classification algorithms*. In this section, we will go over one of the more common SVM algorithms, called *hard-margin SVM*.

The key assumption for the use of SVM is that our two classes of data are *linearly seperable*, meaning that the labelled datapoints in either class should be separable by a line or hyperplane.



(a) Linearly separable data.



(b) Not linearly separable data.

There are two common approaches for determining whether two classes of labelled data are linearly separable. The first approach is checking if the *convex hulls*[10] about either sets of points are disjoint. If so, the data is linearly separable. The second approach is checking if the perceptron algorithm converges – if so, the data is linearly separable.

Let's sketch the derivation of hard-margin SVM. To start, suppose we have a dataset $X \subset \mathbb{R}^n$ with a subset

$$Y = \left\{(x^{(1)}, y^{(1)}), \cdots (x^{(m)}, y^{(m)})\right\} \subset X \times \{-1, 1\}$$

of binary-labelled data. In order to simplify things later on, here we take the label space to be $\{-1, 1\}$ rather than $\{0, 1\}$. We want to find an optimal decision boundary

$$h : \mathbb{R}^n \to \mathbb{R}, \qquad h(\vec{x}) = w_0 + \vec{w} \cdot \vec{x} = 0$$

which separates the two classes of data, also called a *linear classifier*. As usual, $\vec{w} = \langle w_1, \cdots, w_n \rangle$ is the vector of weights corresponding to the features $\vec{x} = \langle x_1, \cdots, x_n \rangle$. In order to ensure that the points $x^{(i)}$ with the label $y^{(i)} = -1$ lie "below" the graph of h, we need to impose the condition

$$w_0 + \vec{w} \cdot x^{(i)} \leq -1, \tag{14}$$

and conversely, for the points $x^{(i)}$ with the label $y^{(i)} = 1$ to lie "above" the graph of h, we must have

$$w_0 + \vec{w} \cdot x^{(i)} \geq 1, \tag{15}$$

Combining these two conditions, for all $1 \leq i \leq m$, $w_0 + \vec{w} \cdot x^{(i)}$ must satisfy

$$y^{(i)} \left(w_0 + \vec{w} \cdot x^{(i)}\right) \geq 1. \tag{16}$$

Take a moment to convince yourself that (14) and (15) are necessary constraints on the hyperplane, and together, they are equivalent to (16).

The condition (16) alone is enough to guarantee that the hyperplane $h(\vec{x}) = 0$ linearly classifies the points. In order to make sure that h is the optimal classifier, we can maximize the distance from the hyperplane to the points on either size. In order to do this, we can simply consider the datapoints which are "closest" to the hyperplane, called *support vectors*. In particular, support vectors are defined as labelled datapoints $x^{(i)}$ for which (16) becomes an equality

$$y^{(i)} \left(w_0 + \vec{w} \cdot x^{(i)}\right) = 1.$$

Letting $v^{(1)}, \cdots, v^{(k)} \in X$ denote the set of support vectors, note that the distance from each $v^{(i)}$ to $h(\vec{x}) = 0$ is given by

$$\frac{\left|w_0 + \vec{w} \cdot v^{(i)}\right|}{\|\vec{w}\|} = \frac{1}{\|\vec{w}\|}. \tag{17}$$

[10]The convex hull about a set of points is the "smallest polytope" containing them.

Therefore, in order to maximize the distance of $h(\vec{x}) = 0$ from the support vectors $v^{(i)}$, we need to maximize $\|\vec{w}\|$. Combining (16) and (17), we need to find the parameters $w_0,\ \vec{w}$ which solve the following optimization problem:

$$\text{minimize} \quad ||\vec{w}|| \quad \text{subject to} \quad y^{(i)}\left(w_0 + \vec{w}\cdot x^{(i)}\right) \geq 1, \quad 1 \leq i \leq m. \tag{18}$$

Using the method of Lagrange multipliers, (18) can transformed into a quadratic programming problem, which can be solved in $O(m^2)$ time using the sequential minimal optimization (SMO) algorithm. Denoting the solution of (18) by $\overline{w_0}, \overline{w}$, the desired hyperplane is given by

$$h(\vec{x}) = \overline{w_0} + \overline{w}\cdot\vec{x} = 0.$$

In order to grasp the intuition behind many machine learning algorithms, it is often beneficial to first understand how thet are implemented for data in $\mathbb{R}^2$. This is particularly true for SVM, which, despite being one of the more mathematically complex algorithms in supervised learning, has a simple intuitive idea behind it. First, think of binary-classified data with two features as points in $\mathbb{R}^2$. The decision boundary obtained by SVM can be understood as the straight road which runs between the two classes of points with the widest sidewalk,around it. An example of this is pictured below.

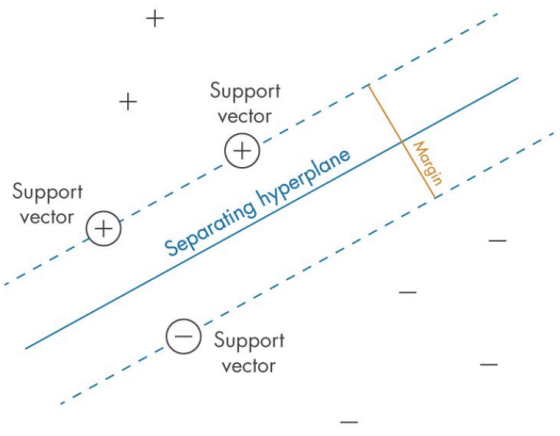


FIGURE 17. Visualization of SVM, courtesy of MathWorks.

The most popular generalization of hard-margin SVM is *soft-margin SVM*, which allows for a small amount of misclassified points, or "noise". Moreover, SVM can even be applied to binary-classified data with a totally nonlinear decision boundary, such as the points inside and outside of the unit circle, using a slick technique called the *kernel trick*. These two methods will be discussed further in the next section.

8. SOFT-MARGIN SVM AND THE KERNEL TRICK

In this section, we will discuss soft-margin SVM and the kernel trick, the two most common adaptations of hard-margin SVM to data which is not linearly separable.[11] In real-life applications of SVM, both methods are often used in tandem.

8.1. **Soft-margin SVM.** As mentioned in Section 7, the soft-margin SVM algorithm generalizes hard-margin SVM by allowing for a few misclassified datapoints. Since most data is not linearly separable, this algorithm is immensely useful in practice. In particular, it is most helpful for data which is "almost" linearly separable, meaning that the data is linearly separable apart from a few outliers, and for data which is linearly separable, but still contains outliners. In the latter case, hard-margin SVM can be applied, but it often causes overfitting, as shown in the below figure.

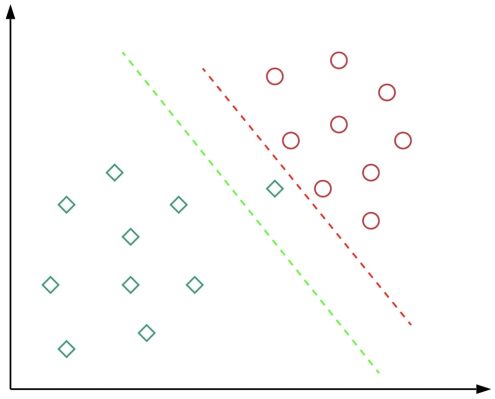


FIGURE 18. Linearly separable data with two decision boundaries.

While the two classes of data (diamonds and circles) can technically be separated using the red line, it is clear that the green line is a superior linear classifier, since it "ignores" the singular diamond which is annoyingly close to the circles.

Recall that, for hard-margin SVM, the ideal linear classifier $w_0 + \vec{w} \cdot \vec{x} = 0$ is the solution to the minimization problem

$$\text{minimize} \quad ||\vec{w}|| \quad \text{subject to} \quad y^{(i)} \left(w_0 + \vec{w} \cdot x^{(i)}\right) \geq 1, \quad 1 \leq i \leq m, \tag{19}$$

where $Y = \{(x^{(1)}, y^{(1)}), \cdots, (x^{(m)}, y^{(m)})\} \subset \mathbb{R}^n$ is our set of labelled data. In order to soften this conditon, we introduce a so-called slack variable ξ_i for each datapoint $x^{(i)}$, such that $\xi_i = 0$ if $y^{(i)}$ is on the correct side of the linear classifier, and if not, ξ_i is the

[11] Much of the contents of this section was inspired by this article by Rishabh Misra.

distance from the hyperplane bounding the points in its class. To see what this means, look carefully at the below visual.

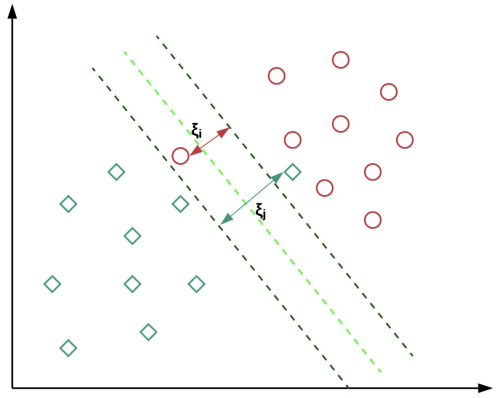


FIGURE 19. Slack variables for two misclassified datapoints.

We can then modify the condition required for our linear classifier to be as follows. For some $C \in [0, \infty)$, we choose the $w_0,\ \vec{w}$ which solve the optimization problem

$$\text{(20)} \qquad \text{minimize} \quad ||\vec{w}|| + C\sum_{i=1}^{m} \xi_i \quad \text{subject to} \quad y^{(i)}\left(w_0 + \vec{w}\cdot x^{(i)}\right) \geq 1 - \xi_i, \quad 1 \leq i \leq m.$$

Note that there are two modifications from the case of hard-margin SVM. Firstly, rather than just minimizing $||\vec{w}||$, we want to minimize $||\vec{w}|| + C\sum_{i=1}^{m} \xi_i$, where $C\sum_{i=1}^{m} \xi_i$ is the weighted sum of the slack variables. The extra parameter C can be adjusted depending on how much we care about classifying points correctly, vs how much we care about maximizing the margin about the linear classifier. If C is large, more emphasis is placed on accurate classification, whereas if C is small, more emphasis is placed on maximizing the margin.

Note that we also relax the lower bound on $y^{(i)}\left(w_0 + \vec{w}\cdot x^{(i)}\right)$. If $x^{(i)}$ is classified correctly, then the lower bound is $1 - \xi_i = 1 - 0 = 1$, but if $x^{(i)}$ is classified incorrectly, the hyperplane is given a bit of extra wiggle room.

8.2. **The kernel trick.** In many cases, the two classes of labelled data $Y \subset \mathbb{R}^n$ cannot be separated by a $(n-1)$-dimensional hyperplane. For example, the points inside and outside of the unit circle in $\mathbb{R}^2$ cannot be separated by a straight line. In order to apply SVM in such cases, we can apply a technique known as the *kernel trick*. By applying a nonlinear transformation $\phi : \mathbb{R}^n \to \mathbb{R}^m$ for some m, we can linearly separate the labelled points in $(n+1)$-dimensional space by an n-dimensional hyperplane. For example, in order to separate the points inside and outside the unit circle, we could define the map

$$\phi : \mathbb{R}^2 \to \mathbb{R}, \qquad \phi(\vec{x}) = ||\vec{x}||.$$

If $\vec{x}$ lies inside the unit circle, we have $||\vec{x}|| < 1$, and if not, $||\vec{x}|| > 1$. After associating each $\vec{x} \in \mathbb{R}^2$ with the point $(\vec{x}, \phi(\vec{x})) \in \mathbb{R}^3$, we can separate the points inside and outside of the unit circle with the plane $z = 1$ in $\mathbb{R}^3$.

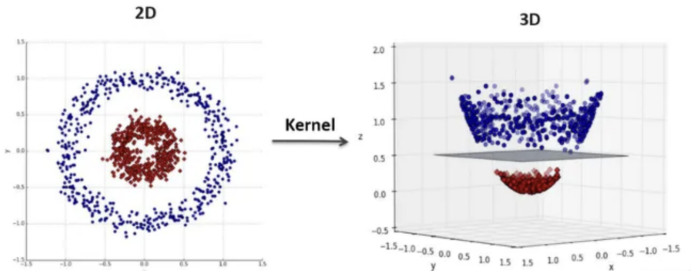


FIGURE 20. The kernel trick in $n = 2$ dimensions. Courtesy of Suraj Yadav.

Note that using the transformation function ϕ, we can define a new distance between datapoints. Ordinarily, the linear distance between vectors $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$ can be measured using the dot product $\mathbf{x}_1 \cdot \mathbf{x}_2 = ||\mathbf{x}_1||\,||\mathbf{x}_2|| \cos(\theta)$, where θ is the angle between the two vectors. The dot product is a type of a *kernel function* called a *linear kernel*. More generally, for some transformation function $\phi : \mathbb{R}^n \to \mathbb{R}^m$, a kernel function $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is defined as

$$K(\mathbf{x}_1, \mathbf{x}_2) = \phi(\mathbf{x}_1) \cdot \phi(\mathbf{x}_2).$$

For an appropriate transformation ϕ, K can gives us a linear measure of the distance between datapoints which may have a nonlinear relationships. Thus, while the linear separability requirement makes SVM impossible to apply in most cases, it can often be overcome by using transforming the data appropriately.

One common nonlinear kernel function is the *degree-d polynomial kernel* K_d, defined as[12]

$$K_d(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1 \cdot \mathbf{x}_2 + c)^d.$$

While the kernel trick is an essential step in the data preparation process in many applications of SVMs, the mathematical and numerical methods required to find appropriate kernel functions in higher dimensions lie outside the scope of this document.

[12]While K_d isn't explicitly written in the form $K_d(\mathbf{x}_1, \mathbf{x}_2) = \phi(\mathbf{x}_1) \cdot \phi(\mathbf{x}_2)$, Mercer's theorem guarantees that a map ϕ representing K_d exists.

9. Appendix A: Assumptions for linear regression

In this section, we will go over the assumptions needed for the appropriate usage of linear regression, as well as some practical methods of checking them.

Given a set $Y = \{(\vec{x}^{(1)}, y^{(1)}), \cdots, (\vec{x}^{(m)}, y^{(m)})\} \subset \mathbb{R}^{n+1}$ of m labelled datapoints, let $f(x_1, \cdots, x_n) = w_0 + w_1 x_1 + \cdots + w_n x^n$ be the regression plane to the points in Y.

Definition 9.1. From a set of labelled data Y with size m, and for all $1 \leq i \leq m$, the ith residual is defined as

$$e_i = f(\vec{x}^{(1)}) - y^{(1)}.$$

In order for f to be a good model for the labelled data, the residuals must satisfy the following four assumptions.

(1) **Independence.** None of the features $x_1, \cdots, x_n$ should be highly correlated. This can be tested statistically using the *Durbin-Watson test*. Given a list `residuals` of the residuals, the Python code for the test is as follows.

```
from statsmodels.stats.stattools import durbin_watson
durbin_watson_stat = durbin_watson(residuals)
print(f'Durbin-Watson statistic: {durbin_watson_stat}')
```

The Durbin-Watson statistic is always a real number between zero and four. A statistic closer to two suggests independence, whereas a statistic closer to zero or four suggests correlation.

(2) **Linearity.** The datapoints $\vec{x}^{(1)}, \cdots, \vec{x}^{(m)}$ should have an approximately linear relationship to the labels $y^{(1)}, \cdots, y^{(m)}$. This assumption is often checked visually.

(3) **Normality of residuals.** The residuals $e_1, \cdots, e_m$ should be normally distributed with mean 0 and variance 1. The two most popular methods of checking for normality are the *Q-Q plot*, and the *Shapiro-Wilk Test*. The Q-Q plot can be visualized in Python using the following code.

```
import scipy.stats as stats
import matplotlib.pyplot as plt
stats.probplot(residuals, dist="norm", plot=plt)
plt.show()
```

If the points fall approximately on the line $y = x$, the residuals are normal. On the other hand, the Shapiro-Wilk Test can be implemented in Python with the following code.

```
from scipy.stats import shapiro
shapiro_test = shapiro(residuals)
print(f'Shapiro-Wilk test p-value: {shapiro_test.pvalue}')
```

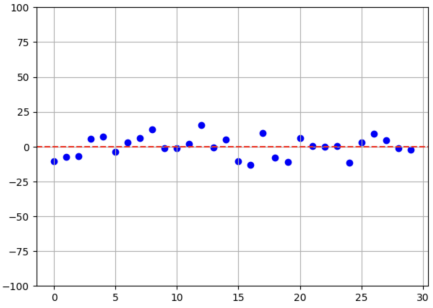
If the outputted p-value exceeds the significance level of 0.05, we can assume the residuals are normally distributed.

(4) **Homoscedasticity (Same variance).** The size of the errors e_i should all be approximately equal. One test for this assumption is the *White Test*, which is implemented in Python below.

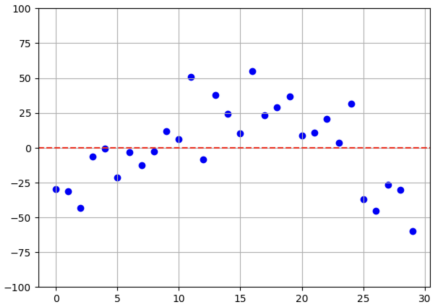
```
from statsmodels.stats.diagnostic import het_white
lm_stat, lm_p_value, f_stat, f_p_value = het_white(residuals, X)
print(f'White test p-value: {lm_p_value}')
```

If the p-value exceeds 0.05, we can assume homoscedasticity, or a lack of *heteroscedasticity*.

We can also perform a quick check of all four assumptions by looking at the *residual plot*, which consists of the points $(1, e_1), \cdots (m, e_m)$. If regression plane is a good fit, the residual plot should look like a random cloud of points centered at x-axis. To get a clearer sense of what that means, let's take a look at the residual plots from the linear regressions performed in Examples 2.2 and 4.1.



(a) Residual plot from Example 2.2.



(b) Residual plot from Example 4.1.

We will first analyze the residual plot from Example 4.1. Heteroscedasticity is usually detected through a lack of a "cone-like" pattern in the residual plot, where the residuals do not get farther from or closer to the x-axis from left to right. Since this does not occur in Example 4.1, the homoscedasticity assumption may be satisfied. However, the residuals do follow an arc, so the data may be nonlinear and correlated. The residuals also attain extreme values like 50, so we cannot assume the data is normally distributed with mean 0 and variance 1. Since three assumptions seem to be violated, there is sufficient evidence to reject linear regression as a viable model for the data in Example 4.1.

On the other hand, in Example 2.2, the residuals do seem to be randomly distributed about the x-axis with a relatively uniform and small variance. Therefore, we can conclude from the residual plot that linear regression is a good model for our data.

10. Appendix B: Gradient descent

Here, we will show that, if $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable, then at any point $\mathbf{p} \in \mathbb{R}^n$, $-\nabla f(\mathbf{p})$ is the "direction of fastest decrease" at $f(\mathbf{p})$. To start, let's define what it means for f to increase or decrease in a particular direction.

Definition 10.1. Let $f : \mathbb{R}^n \to \mathbb{R}$ be differentiable, and let $\mathbf{v} \in \mathbb{R}^n$ be a vector such that $||\mathbf{v}|| = 1$. The *directional derivative* of f at a point $\mathbf{p} \in \mathbb{R}^n$ is defined as

$$D_{\mathbf{v}} f(\mathbf{p}) = \nabla f(\mathbf{p}) \cdot \mathbf{v}.$$

Intuitively, $D_{\mathbf{v}} f(\mathbf{p})$ measures the rate of change of f at $\mathbf{p}$ in the direction of a given unit vector $\mathbf{v}$. If $D_{\mathbf{v}} f(\mathbf{p})$ is positive, f is increasing at $\mathbf{p}$ in the direction of $\mathbf{v}$. On the other hand, if $D_{\mathbf{v}} f(\mathbf{p})$ is negative, f is decreasing at $\mathbf{p}$ in the direction of $\mathbf{v}$. Moreover, the larger $|D_{\mathbf{v}} f(\mathbf{p})|$ is, the faster the rate of change of f is at $\mathbf{p}$ in the direction of $\mathbf{v}$.

We will show that, for any point $\mathbf{p} \in \mathbb{R}^n$, the directional derivative $D_{\mathbf{v}} f(\mathbf{p})$ is minimized when $\mathbf{v} = -\nabla f(\mathbf{p})$. First, by the definition of the dot product, we have

$$D_{\mathbf{v}} f(\mathbf{p}) = \nabla f(\mathbf{p}) \cdot \mathbf{v} = ||\nabla f(\mathbf{p})||\, ||\mathbf{v}|| \cos(\theta), \tag{21}$$

where $\theta \in [0, 2\pi)$ denotes the angle between $\nabla f(\mathbf{p})$ and $\mathbf{v}$. Since $||\nabla f(\mathbf{p})||\, ||\mathbf{v}|| = ||\nabla f(\mathbf{p})||$ is a positive scalar, $D_{\mathbf{v}} f(\mathbf{p})$ is minimized when $\cos(\theta)$ is minimized. But

$$\min_{\theta \in [0, 2\pi)} \cos(\theta) = \cos(\pi) = -1,$$

implying that $D_{\mathbf{v}} f(\mathbf{p})$ attains a minimum when the angle θ between $\nabla f(\mathbf{p})$ and $\mathbf{v}$ is π, or 180 degrees. Therefore, the directional derivative $D_{\mathbf{v}} f(\mathbf{p})$ is minimized when $\mathbf{v}$ is pointing opposite direction of $\nabla f(\mathbf{p})$, or when

$$\mathbf{v} = -\frac{\nabla f(\mathbf{p})}{||\nabla f(\mathbf{p})||}.$$

Moreover,

$$D_{\mathbf{v}} f(\mathbf{p}) = -||\nabla f(\mathbf{p})|| \leq 0$$

so f is decreasing at $\mathbf{p}$ in the direction of $-\nabla f(\mathbf{p})$ when $\nabla f(\mathbf{p}) \neq \mathbf{0}$. □