Machine Learning From Scratch

Hieu Nguyen

hieutn@bu.edu

Machine Learning From Scratch

Hieu Nguyen

20th December 2024

For Elaina. Thank you for always believing in me.

Hieu

Contents

Part I

Supervised Learning

1

Linear Regression

"If you're the one teaching the machine, shouldn't be called machine teaching?"

–

Machine learning has the misconception that it is this novel field that currently hot and pave way for the future. Machine learning is an science founded on the backs of statistics. The reason it became so popular recently is due to the sheer computational power of modern computers. As such, I don't want to yap about something that has been known for maybe more than half a century. I want to revisit these ideas and build them from scratch, because I think it will be more useful for the reader and myself that way. While the mathematics is important and will be covered, the emphasis will be more on the implementation, as the implementation is the thing that's hot. What that means is that unlike my probability and QFT notes, these will not include many example problems, but will include a lot more coding.

1.0.1 Setting Up The Problem

The point of machine learning is to find structure from data. What information can we extract? If the data was perfect and there were no randomness involved, then this would be trivial and the field would have been solved maybe hundreds of years ago. But that is not true; data is noisy. This is due to a variety of reasons, and we will not dwell on it too much. What we will do, though, is find the best possible path through the noisy forest. Let us consider the following problem: suppose you go out and you collect data, and when you

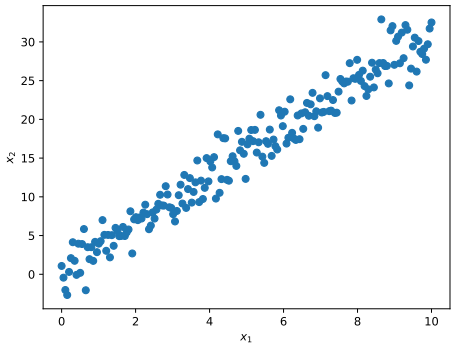

Figure 1.1: Example plot for datasets x_1 and x_2

plot the dataset on an XY plane, you see that there seems to be a correlation between x_1 and x_2 (examine figure 1.1). It seems as though if you increase x_1 then x_2 also increases; but by how much? What is the relation? That is what we are going to try to answer in this chapter.

1.1 Least Squares

The method that we are going to be using to solve this problem is called least squares, and relies on one major assumption: **the relationship is linear**. Suppose that there exists a line that passes through the average of all the data points, and I'm using the term average loosely here. That is, I want it so that about half the points are above the line and half to be below so that I can track the average trend. If we are going to make the assumption that the relationship is linear, then there must exists a function

$$x_2 = \theta_1 x_1 + \theta_0 \tag{1.1}$$

for some values of θ_1 and θ_0 that best encapsulates the relationship between x_1 and x_2. For now on, we are going to denote x_i as our input data, y_i as our output (in this case, y_1 corresponds to x_2), and $\hat{y}$ as our prediction.

1.1.1 The Cost Function

We are going to introduce a function that measures basically the average distance of every data point to the line (our prediction)

$$J(\theta_0, \theta_1) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{1.2}$$

$$= \sum_{i=1}^{N} (y_i - (\theta_1 x_i + \theta_0))^2$$

Each data point x_i has an output y_i, and we want to find the values for the slope and intercept (θ) that minimizes the squared distance. Notice that we are using the squared distance because otherwise, the cost function can be negative and will give us very bad results when we try to minimize it. Another way to think about it is you care more about the distance than the displacement. Now we need to find the values of θ_0 and θ_1 that minimizes the cost function, and to do that we take a derivative and set it equal to 0.

$$\frac{\partial}{\partial \theta_i} J(\theta_0, \theta_1) = 0$$

$$\implies \sum_{i=1}^{N} \frac{\partial}{\partial \theta_1} (y_i - (\theta_1 x_i + \theta_0))^2 = 0$$

$$-2 \sum_{i=1}^{N} x_i (y_i - \theta_1 x_i - \theta_0) = 0$$

Now, solving for θ_1, we see that

$$\sum_{i=1}^{N} x_i y_i - \theta_1 \sum_{i=1}^{N} x_i^2 - \theta_0 \sum_{i=1}^{N} x_i = 0$$

Isolating the θ_1 term, we get that

$$\theta_1 = \frac{\sum_{i=1}^{N} x_i y_i - \theta_0 \sum_{i=1}^{N} x_i}{\sum_{i=1}^{N} x_i^2}$$

But now we have an undetermined variable: the intercept θ_0 term. We can solve for this by taking the derivative of θ_0 and minimizing that too

$$\frac{\partial J}{\partial \theta_0} = -2 \sum_{i=1}^{N} (y_i - (\theta_1 x_i + \theta_0))$$

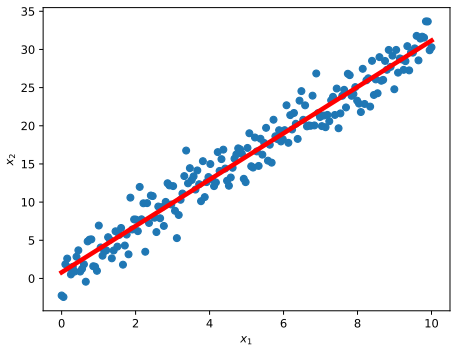

Figure 1.2: Fitting a line using the slope and intercept values from calculated from equation 1.3 and 1.4

Setting the derivative to 0

$$\sum_{i=1}^{N} (y_i - \theta_1 x_i - \theta_0) = 0$$

Then solving for θ_0

$$\theta_0 = \frac{\sum_{i=1}^{N} y_i - \theta_1 \sum_{i=1}^{N} x_i}{N}$$

This gives us 2 equations for 2 variables, we can solve the system of equations. I will use our magic helper tool called Mathematica to solve this system of equation, and in doing so, we find that

$$\theta_1 = \frac{N \sum_{i=1}^{N} x_i y_i - \left(\sum_{i=1}^{N} x_i\right)\left(\sum_{i=1}^{N} y_i\right)}{N \sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2} \tag{1.3}$$

$$\theta_0 = \frac{\sum_{i=1}^{N} x_i^2 \sum_{i=1}^{N} y_i - \sum_{i=1}^{N} x_i \sum_{i=1}^{N} x_i y_i}{N \sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2} \tag{1.4}$$

That is gnarly. But to check proof of concept, we will use these equations to find the slope and intercept and see how it fits to the line. We find that for our given data set. Peep figure 1.2.

$$\theta_1 = 3.036549977758679 \qquad \theta_0 = 0.7963857006463769$$

This is really great, we found a way to do this for 2 dimensional data sets! But in real life, datasets are almost never 2 dimensional like this. In fact, to get any meaningful predictions, these algorithms are trained on millions and millions of data points living in millions and millions of dimensional vector spaces. Computing the derivatives for the slopes and intercept by hand like this is near impossible, and more importantly, impractical. We must resort to

vectorization.

1.1.2 The Normal Equations

Instead of computing derivatives one at a time, we will do them all simultaneously for any arbitrary dimension. We note that when we do this, our fit is no longer a line but the hyperplane $N-1$ dimensional hyperplane. Let us attempt to vectorize our procedure by translating the cost function $J(\theta)$ into a vector function (J is still a scalar, but now x and y are matrices). If we think about what the sum of squares is really doing, it is taking a dot product with itself where the vector is $\mathbf{y}-\hat{\mathbf{y}}$ (you can convince yourself that this is true by expanding the quadratic and seeing that there are no free indices and all indices on every term are repeated). Thus, the cost function becomes

$$J(\theta) = (\mathbf{y}-\hat{\mathbf{y}})^T (\mathbf{y}-\hat{\mathbf{y}}) \tag{1.5}$$

Reintroducing our predictor gives

$$J(\theta) = (\mathbf{y}-\mathbf{X}\theta)^T (\mathbf{y}-\mathbf{X}\theta)$$

Where $\mathbf{y}$ is the vector of output, $\mathbf{X}$ is the matrix of inputs called **the feature matrix**. θ now is a vector. One thing we should note: the columns of $\mathbf{X}$ contain all the data points for its respective category AND one extra column of all 1's for the intercept term. Look back to equation 1.1, notice that there were no data points multiplying the intercept term. Once we have vectorized our cost function, we can minimize it by taking the derivative with respect to θ. To do so, first expand the quadratic

$$J(\boldsymbol{\theta}) = \mathbf{y}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\boldsymbol{\theta} - \boldsymbol{\theta}^T\mathbf{X}^T\mathbf{y} + \boldsymbol{\theta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}$$

We can simplify a bit more and notice that since$\mathbf{y}^T\mathbf{X}\boldsymbol{\theta}$ is a scalar, it is equal to its own transpose $\mathbf{y}^T\mathbf{X}\boldsymbol{\theta} = (\mathbf{y}^T\mathbf{X}\boldsymbol{\theta})^T = \boldsymbol{\theta}^T\mathbf{X}^T\mathbf{y}$ which simplifies the expansion to

$$J(\boldsymbol{\theta}) = \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\boldsymbol{\theta} + \boldsymbol{\theta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}$$

Now taking the derivative term by term

$$\frac{\partial J}{\partial \boldsymbol{\theta}} = -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}$$

And setting it equal to 0 gives us

$$2\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - 2\mathbf{X}^T\mathbf{y} = 0$$

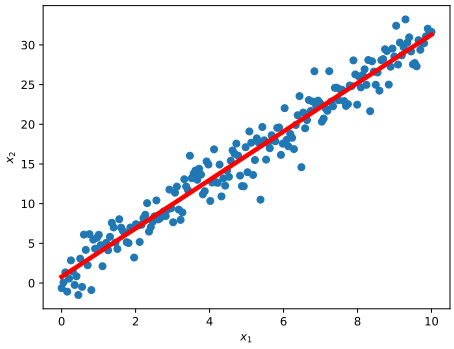

Figure 1.3: Least squares fitting with the θ values given by the normal equations

$$\implies \mathbf{X}^T\mathbf{X}\boldsymbol{\theta} = \mathbf{X}^T\mathbf{y}$$

Thus we can solve for θ by left multiplying the inverse of $\mathbf{X}^T\mathbf{X}$

$$\boldsymbol{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{1.6}$$

The above equation is called **the normal equation**. The reason for that is because the solution the solution is the normal projection of y onto the subspace spanned by the column space of $\mathbf{X}$ blah blah blah. At one point in my life I would have preferred the geometric derivation of this equation but now, I see that calculus gives us a systematic approach.

To check proof of concept, we will perform the least squares fitting on the 2D case. Code will be provided.

```
import numpy as np

# Example feature vector x and output vector y
x = np.linspace(0,10,200)  # Replace with your 1D array of features
y = 3 * x + 1 + np.array([np.random.normal(0, 2) for i in range(len(x))])

# Reshape x to be a 2D array (if it's not already)
X = x.reshape(-1, 1)

# Add a column of ones to X for the intercept term
X_with_intercept = np.hstack((np.ones((X.shape[0], 1)), X))

# Solve for theta using the normal equation
theta = np.linalg.inv(X_with_intercept.T @ X_with_intercept) @
X_with_intercept.T @ y

```

```
17 print("Theta:", theta)
```

et Voila! The power of statistics!

1.2 Gradient Descent

We mentioned before that the purpose of vectorization was to deal with the fact that the data points could live in extremely high dimensional vector spaces. As such, taking the inverse of the covariance matrix $\mathbf{X}^T\mathbf{X}$ may not be computationally effective (timewise). This is due to the fact that the best algorithms for inverting matrices are $O(n^3)$. We will attempt to by pass this by using gradient descent. If we think back to multivariable calculus and the purpose of the gradient, it was to find the direction of highest ascension. The idea here is that the cost function is some convex surface in hyperspace; if we take the gradient, it will point in the direction of highest ascension. The negative gradient will give us the direction of greatest descension. If we take steps following the negative gradient, it will eventually lead us to the global minimum. The algorithm goes as follows:

Gradient Descent

(1) **1.** Start at some random point for θ_i
(2) **2.** Find the gradient of J
(3) **3.** Subtract the gradient (multiplied by some scaling factor α to denote the step size) from the current value of θ_i
(4) **4.** Repeat.

Our job now is to find the gradient of J at each value of θ_i. Begin with the definition of J

$$J(\theta) = \sum_{i=1}^{N} \left(y_i - \sum_{j=i}^{M} x_j^i \theta^j + \theta_0 \right)^2$$

Where M is the number of different feature categories and N are the number of data points in each category. Since this is a scalar function, we expect that taking the gradient would give us a vector. Thus, computing the gradient gives us

$$\nabla_\theta J = -2 \sum_{i=1}^{N} \left(y_i - \sum_{j=i}^{M} x_j^i \theta^j + \theta_0 \right) x_j^i \delta_k^j$$

The Kronnecker delta arises from the fact that the gradient ∂_j of $c_i x_i$ is by 1 if the indices match and 0 else. Simplifying the gradient then gives is

$$= -2\sum_{i=1}^{N}\left(y_i - \sum_{j=i}^{M} x_j^i \theta^j + \theta_0\right) x_k^i$$

Notice that there is only 1 free index, and so this is indeed a vector. We can simplify the notation just a bit more and say that

$$\left(y_i - \sum_{j=i}^{M} x_j^i \theta^j + \theta_0\right) = \Delta_i$$

Now the gradient term can be computed as

$$\nabla_\theta J = -2\sum_{i=1}^{N} \Delta_i x_k^i \tag{1.7}$$

We have not yet computed the derivative of θ_0, and doing so gives

$$\frac{\partial J}{\partial \theta_0} = \sum_{i=1}^{N} \frac{\partial}{\partial \theta_0}\left(\Delta_i^2\right) = \sum_{i=1}^{N} 2\Delta_i\left(-1\right) = -2\sum_{i=1}^{N}\Delta_i$$

Explicitly, in matrix form, the gradient is

$$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \vdots \\ \frac{\partial J}{\partial \theta_M} \end{bmatrix} = \begin{bmatrix} -2\sum_{i=1}^{N}\Delta_i \\ -2\sum_{i=1}^{N} x_{i1}\Delta_i \\ \vdots \\ -2\sum_{i=1}^{N} x_{iM}\Delta_i \end{bmatrix} \tag{1.8}$$

Now that we have the gradient, we have to subtract the negative gradient from the θ vector

$$\begin{bmatrix} \theta_0^{\text{new}} \\ \theta_1^{\text{new}} \\ \theta_2^{\text{new}} \\ \vdots \\ \theta_M^{\text{new}} \end{bmatrix} = \begin{bmatrix} \theta_0^{\text{old}} \\ \theta_1^{\text{old}} \\ \theta_2^{\text{old}} \\ \vdots \\ \theta_M^{\text{old}} \end{bmatrix} - \alpha \begin{bmatrix} -2\sum_{i=1}^{N}\Delta_i \\ -2\sum_{i=1}^{N} x_{i1}\Delta_i \\ -2\sum_{i=1}^{N} x_{i2} \\ \vdots \\ -2\sum_{i=1}^{N} x_{iM}\Delta_i \end{bmatrix} \tag{1.9}$$

I do flip flop between lowering and upper indices and this is because of physics notation. All that matters is that if the indices are repeated, they are summed. We repeat equation 1.9 until we reach the global minimum. How do we know when to stop? There are a couple

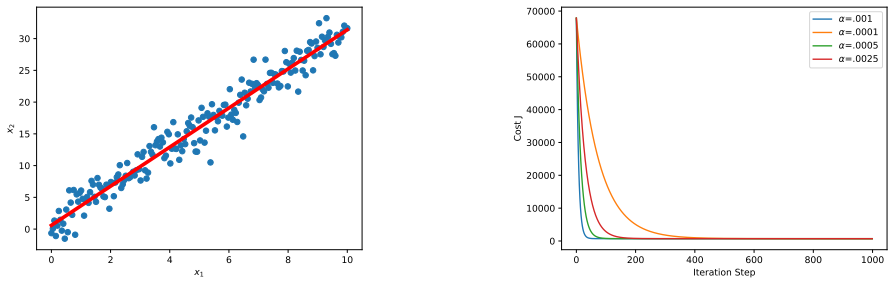

Figure 1.4: On the left, we have the plot of least squares using the gradient descent. On the right, we have the cost (and subsequently the accuracy) as a function of the iteration step plotted for different values of the learning rate α. Obviously for a fixed number of iterations that worse predictions as α gets smaller because for smaller steps it will take longer to reach the minimum

of things you can do, and one of them is to stop when the difference between subsequent θ values are sufficiently small (how small is a choice that we make). In practice, we need not worry about such things because packages will have gradient descent algorithms built in and they do all the heavy lifting for you. Trying to implement gradient descent from scratch is akin to reinventing the wheel. However, as a proof of concept, we will now implement the gradient descent algorithm on our original problem.

```
# Example input feature matrix X and output vector y
X = np.array([...])  # Replace with your feature matrix
y = np.array([...])  # Replace with your output vector

# Add a column of ones to X for the intercept term
X_with_intercept = np.hstack((np.ones((X.shape[0], 1)), X))

# Initialize theta to zeros or random values
theta = np.zeros(X_with_intercept.shape[1])

# Set the learning rate (step size)
alpha = 0.001

# Number of iterations for gradient descent
iterations = 1000

# Gradient descent algorithm
for _ in range(iterations):
    # Compute the predictions
    predictions = X_with_intercept @ theta

```

```
    # Compute the errors
    errors = predictions - y

    # Compute the gradient (note that we divide by len(y) to normalize vector)
    gradient = (2 / len(y)) * (X_with_intercept.T @ errors)

    # Update theta by subtracting the gradient multiplied by the learning rate
    theta -= alpha * gradient

print("Theta:", theta)
```

We test our result by computing the plot our results in figure 1.4. Gradient descent will be a power tool going forward because often times we will not be able to get exact solutions and must rely on it. We also note that we did not demonstrate linear regression with multiple categorical features. Fear not! You can play around with the code and see that it still works. We only used 2D examples for visualization purposes.

1.3 Regularization

One thing that we do have to take into consideration is that for larger and larger datasets, it is possible to run into data points that are highly correlated. What this will do is make the rows of the feature matrix the same (or very similar). This will reduce the rank of the correlation matrix from being full rank and then taking the inverse of this becomes untenable. What we do we will do now is perform Ridge regression, which just means we will introduce a squared bias term $\lambda \sum_{k=1}^{M} ||\theta_k||^2$ to punish θ values that are very very large. This would make sense since if the correlation matrix is lower ranked, the θ values might blow up when we try to take inverses (because when taking inverses, we divide by the determinant and if the matrix is almost singular, than eigenvalues become very very small making the determinant very small and therefore the inverse matrix blows up). To see how introducing the penalty term works, begin with the cost function

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} \left(y_i - \sum_{k=1}^{M} x_{ik}\theta_k - \theta_0 \right)^2 + \lambda \sum_{k=1}^{M} \theta_k^2$$

We can rewrite this in matrix form as

$$J(\boldsymbol{\theta}) = (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T (\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) + \lambda \boldsymbol{\theta}^T \boldsymbol{\theta}$$

Then expanding everything out

$$J(\boldsymbol{\theta}) = \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\boldsymbol{\theta} + \boldsymbol{\theta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} + \lambda\boldsymbol{\theta}^T\boldsymbol{\theta}$$

Taking the derivative and setting it to 0

$$\frac{\partial J}{\partial \boldsymbol{\theta}} = -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} + 2\lambda\boldsymbol{\theta}$$

$$\implies 2\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} + 2\lambda\boldsymbol{\theta} = 2\mathbf{X}^T\mathbf{y}$$

$$\implies (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\theta} = \mathbf{X}^T\mathbf{y}$$

Now solving for θ by taking the inverse

$$\boldsymbol{\theta} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \tag{1.10}$$

We see that the effect of the penalty term introduces a nonzero term on the diagonals of the correlation matrix. This will guarantee full rank of the matrix, making inversion stable.

1.3.1 Gradient Descent With Ridge

For the same reasons that we might want to do gradient descent with regular least squares, we will want to be interested in doing gradient descent with Ridge regression. The only thing that changes here is now we need to find the gradient for the Ridge term. Starting from the cost function

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N}\left(y_i - \sum_{k=1}^{M} x_{ik}\theta_k - \theta_0\right)^2 + \lambda\sum_{k=1}^{M}\theta_k^2$$

We see that the only thing that changes is just the Ridge penalty term, thus we need to find

$$\frac{\partial}{\partial\theta_k}\left(\lambda\sum_{k=1}^{M}\theta_k^2\right) = 2\lambda\theta_k$$

Everything else is the same because we have already computed the gradient term in ther previous section. Thus the gradient term is

$$\frac{\partial J}{\partial\theta_k} = -2\sum_{i=1}^{N} x_{ik}\Delta_i + 2\lambda\theta_k \tag{1.11}$$

and in matrix form

$$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \vdots \\ \frac{\partial J}{\partial \theta_M} \end{bmatrix} = \begin{bmatrix} -2\sum_{i=1}^{N} \Delta_i \\ -2\sum_{i=1}^{N} x_{i1}\Delta_i + 2\lambda\theta_1 \\ \vdots \\ -2\sum_{i=1}^{N} x_{iM}\Delta_i + 2\lambda\theta_M \end{bmatrix} \tag{1.12}$$

The modified code is also provided

```
def ridge_regression(X, y, lambda_reg):
    """
    Perform ridge regression on the provided dataset.

    Parameters:
    X (numpy.ndarray): The feature matrix (size: n_samples x n_features).
    y (numpy.ndarray): The target vector (size: n_samples).
    lambda_reg (float): The regularization parameter.

    Returns:
    numpy.ndarray: The estimated coefficients, including the intercept.
    """
    n_samples, n_features = X.shape

    # Add a column of ones to X for the intercept term
    X_intercept = np.c_[np.ones(n_samples), X]

    # Identity matrix for regularization (exclude intercept term)
    I = np.eye(n_features + 1)
    I[0, 0] = 0  # No regularization for the intercept

    # Compute the ridge regression coefficients
    XtX = X_intercept.T @ X_intercept
    Xty = X_intercept.T @ y
    coefficients = np.linalg.inv(XtX + lambda_reg * I) @ Xty

    return coefficients

# Example usage:
# X = np.array([[1, 2], [2, 3], [3, 4]])  # feature matrix
# y = np.array([1, 2, 3])                 # target vector
# lambda_reg = 1.0                        # regularization parameter
# coefficients = ridge_regression(X, y, lambda_reg)
# print("Coefficients:", coefficients)
```

2

Linear Classification

"Behold! A Man!"

– Diogenes, holding a plucked chicken

Linear regression was about using linear methods (namely linear algebra) to fit curves to (mainly) continuous data. Now, what if the data were strictly categorical? That is, does this number belong to the 0 class or the 1 class? Is this a dog or a cat? Is this a man or not a man (thanks Diogenes). Let us try to set a fake problem shown in Figure 2.1. We want to try to find an algorithm that can accurately classify whether or not an input data should be in the $x = 1$ class or the $x = 3$ class.

2.1 Why Not Linear Regression For Classification?

Well, we've developed the tools to do linear regression in the last chapter, can we not use those same tools to do classification? How do we begin with this? The premature answer is: yes, sort of. Let us begin with the least squares method, with trying to minimize $\sum_{i=1}^{N} (y_i - \hat{y}_i)^2$ Where now our outputs y_1 take on binary values such as 0 or 1 (left or right, black and white, etc.). As long as you are able to represent the output as some sort of binary numerical, this should be hokey dokey. Our predictors $\hat{y}$ will continue to be continuous, and this is the result of linear regression. We can force it to be binary (choosing 0/1 as the convention) by saying if the output is greater than .5, we make it 1 and if less, it's 0. Since

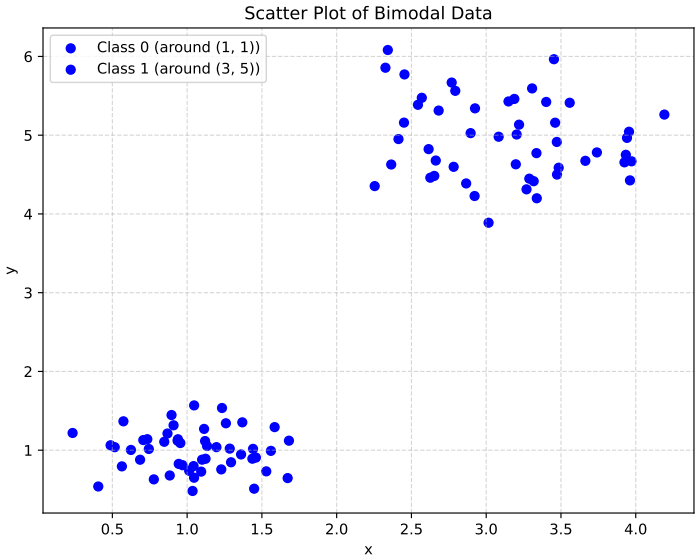

Figure 2.1: Scatter plot of points distributed from a bimodal distribution centered at $x = 1$ and $x = 3$. We want to find a way to classify points.

we are doing linear regression, we can use the normal equations

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

To solve for the θ values that best fits the data. Being the good data scientists that we are, we have to prepare the data before stabbing it with least squares regression. Namely, we first have to construct the feature matrix $\mathbf{X}$. The first column will be the intercept column of all 1s, the second column will the column of the x coordinate data points, and the third column will be the y coordinate data points

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \\ \vdots & \vdots & \vdots \\ 1 & x_{100} & y_{100} \end{bmatrix}$$

The third column is not to be confused with our output y_i. It actually might be better to think of the third column as x_2, where x_2 is the second input. The output vector will be

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

Where we have defined 0 to be $x = 1$ class and 1 to be $x = 3$ class. Now we have the feature matrix and output data. Before performing least squares classification, we have to split the data into training set and testing set. We will use the submatrix of X to calculate the θ values then use that model to predict the testing data. We compare the accuracy with the true value. Here is the code:

```
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Set the random seed for reproducibility
np.random.seed(0)

# Generate random points from a bimodal distribution
# First mode centered around (1, 1)
x1 = np.random.normal(loc=1, scale=0.2, size=50)
y1 = np.random.normal(loc=1, scale=0.2, size=50)

# Second mode centered around (3, 5)
x2 = np.random.normal(loc=3, scale=0.2, size=50)
y2 = np.random.normal(loc=5, scale=0.2, size=50)

# Combine the data
x = np.concatenate([x1, x2])
y = np.concatenate([y1, y2])

# Create labels: 0 for the first mode, 1 for the second
labels = np.array([0] * 50 + [1] * 50)

# Prepare the feature matrix
X = np.c_[np.ones(x.shape[0]), x, y]  # Add intercept term

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.3,
    random_state=0)

```

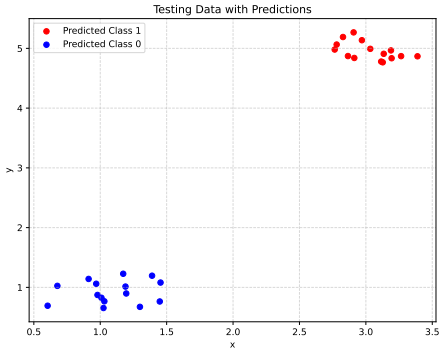

Figure 2.2: Running least squares classification on testing data. We classified the class using colors

```
# Solve the normal equation on the training set
theta = np.linalg.pinv(X_train.T @ X_train) @ X_train.T @ y_train

# Predict function
def predict(X, theta):
    predictions = X @ theta
    return np.where(predictions >= 0.5, 1, 0)

# Make predictions on the test data
y_pred = predict(X_test, theta)

# Calculate accuracy on the test set
accuracy = np.mean(y_pred == y_test)

accuracy, y_pred, y_test
```

The output is

```
(1.0,
 array([0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
        0, 1, 0, 0, 0, 1, 1, 1]),
 array([0, 1, 0, 1, 1, 1, 0, 1, 1, 1,1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
        0, 1, 0, 0, 0, 1, 1, 1]))
```

It seems like the accuracy is 100%! That's pretty good, right? Yes, however, the biggest problem with this method of classification is that once you have more than 2 classes, this model now sucks. This was pretty obvious from the get go, since a line can only separate the data into 2 regions. What do we do now?

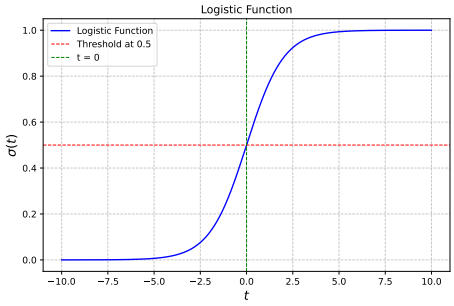

Figure 2.3: The logistic function

2.2 Logistic Regression

The problem with least squares classification is that the generated line is unbounded. We would like to bound the output so that the output has the interpretation of something like probability. A function that bounds inputs below by 0 and above by 1 is the logistic function

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

which we plot in Figure 2.3. In fact, we will use this exactly as the conditional probability of the data point x_i belonging to the class y_i where y_i can take on 2 different classes

$$\mathbb{P}(y = 1 \mid x) = \sigma(z) = \frac{1}{1+e^{-z}}$$

$$\mathbb{P}(y = 0 \mid x) = 1 - \sigma(z) = 1 - \frac{1}{1+e^{-z}}$$

Where $z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$ We can compact the above conditional probabilities into 1 by writing

$$\mathbb{P}(y \mid \boldsymbol{x}, \boldsymbol{w}) = \left(\sigma\left(\boldsymbol{w}^\top \boldsymbol{x}\right)\right)^y \left(1 - \sigma\left(\boldsymbol{w}^\top \boldsymbol{x}\right)\right)^{1-y} \tag{2.1}$$

We've introduced the dependencies of the weights $\boldsymbol{w}$ since these are parameters that we will have to solve for in order to find the best fit to the data. Notice that for the above equation, the probability that the point x is classified as $y = 0$ is $1 - \sigma(\boldsymbol{w}^T \boldsymbol{x}$ and $\sigma(\boldsymbol{w}^T \boldsymbol{x}$ for $y = 0$. If we make the assumption that each observed data point is independent of each other, then the probability of the whole data is

$$\mathbb{P}(\{y^{(i)}\}_{i=1}^m \mid \{\boldsymbol{x}^{(i)}\}_{i=1}^m, \boldsymbol{\theta}) = \prod_{i=1}^{m} \left(\sigma(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)})\right)^{y^{(i)}} \left(1 - \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)})\right)^{1-y^{(i)}} \tag{2.2}$$

This is called the **likelihood function of** w. However, working with productions of probabilities tends to be really annoying both analytically and computationally. We can simplify calculations by taking the log of the probabilities. When we do this, products of functions turn into logs

$$\log L(\boldsymbol{\theta}) = \sum_{i=1}^{m} \left[y^{(i)} \log(\sigma(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)})) \right]$$

The above result is called the **log likelihood** function. Simplifying some more gives

$$\log L(\boldsymbol{\theta}) = \sum_{i=1}^{m} \left[y^{(i)} \log \left(\frac{1}{1 + e^{-\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}}} \right) + (1 - y^{(i)}) \log \left(1 - \frac{1}{1 + e^{-\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}}} \right) \right]$$

$$\sum_{i=1}^{m} \left[y^{(i)} (-\log(1 + e^{-\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}})) + (1 - y^{(i)})(-\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)} - \log(1 + e^{-\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}})) \right]$$

$$= \sum_{i=1}^{m} \left[-\log(1 + e^{-\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}}) + (1 - y^{(i)}) \boldsymbol{\theta}^\top \boldsymbol{x}^{(i)} \right]$$

We can use the identity

$$-\log \left(1 + e^{-\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}} \right) = \boldsymbol{\theta}^\top \boldsymbol{x}^{(i)} - \log \left(1 + e^{\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}} \right)$$

and then sub it back into the likelihood function to get

$$\log L(\boldsymbol{\theta}) = \sum_{i=1}^{m} \left[y^{(i)} \boldsymbol{\theta}^\top \boldsymbol{x}^{(i)} - \log(1 + e^{\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}}) \right] \tag{2.3}$$

What exactly are we trying to solve here? We are trying to find the values for **w** that best fit the data. That is the same thing as finding the **maximum likelihood estimator** for **w**. One problem though; taking the derivative gives us a transcendental function. Therefore we cannot solve the above analytically for **w**. Our only hope is to perform gradient descent.

2.2.1 Gradient Descent

We can treat our log-likelihood function $l(\boldsymbol{\theta})$ as our cost function. Let us find the gradient. Going term by term we get

$$\nabla_\theta \sum_{i=1}^{m} y^{(i)} \boldsymbol{\theta}^\top \boldsymbol{x}^{(i)} = \sum_{i=1}^{m} y^{(i)} \boldsymbol{x}^{(i)}$$

Moving onto the second term

$$\nabla_\theta \sum_{i=1}^{m} -\log(1 + e^{\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}}) = \sum_{i=1}^{m} \frac{e^{\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}}} \boldsymbol{x}^{(i)} = \left(\sigma(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)})\right) \boldsymbol{x}^{(i)}$$

By chain rule and simplifying using the fact that $\frac{e^{\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}}}{1+e^{\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}}} = \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)})$. Combining both terms, we find that the gradient term is

$$\nabla_{\boldsymbol{\theta}}\, l(\boldsymbol{\theta}) = \sum_{i=1}^{m} \left[(y^{(i)} - \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}))\boldsymbol{x}^{(i)}\right] \tag{2.4}$$

Since we are trying to maximize the log-likelihood function, it's actually more accurate to call this the **gradient ascent** algorithm. The only difference is that we will now update our values of $\boldsymbol{\theta}$ like

$$\boldsymbol{\theta} := \boldsymbol{\theta} + \alpha \sum_{i=1}^{m} \left[(y^{(i)} - \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}))\boldsymbol{x}^{(i)}\right] \tag{2.5}$$

Where α is our usual learning rate. If we were very fancy, we would slap on the L_2 regularization term and the gradient becomes

$$\boldsymbol{\theta} := \boldsymbol{\theta} + \alpha \left(\sum_{i=1}^{m} \left[(y^{(i)} - \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)}))\boldsymbol{x}^{(i)}\right] - \lambda\boldsymbol{\theta} \right)$$

Once we find the optimal $\boldsymbol{\theta}$ values, we can classify the data points using (2.1). We say that if $\mathbb{P}(y = 1) \geq .5$, then choose the class 1 else 0.

2.2.2 Why Is This A Linear Classifier?

It's a bit weird to be discussing this under the linear classifier chapter, isn't it? Well, not at all! The reason why we call this a linear classifier is because the decision boundary is linear. We make classification depending on whether $\mathbb{P}(y = 1) \geq .5$ or else. But the probability is given by the sigmoid. The only time this happens is when $\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n \geq 0$ and the else case happens when $\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n < 0$. Thus, the decision boundary is at $\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n = 0$, which is a linear hyperplane.

2.2.3 An Example

We all love proofs of concepts, so let's do one. First, we construct our feature matrix as usual. Don't forget to add the bias column! Then we create our labels y. We use sklearn to split the data and output into training sets and testing sets. We implement gradient descent with L_2 regularization (Ridge).

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Set the random seed for reproducibility
np.random.seed(0)

# Generate random points from a bimodal distribution
# First mode centered around (1, 1)
x1 = np.random.normal(loc=1, scale=0.3, size=50)
y1 = np.random.normal(loc=1, scale=0.3, size=50)

# Second mode centered around (3, 5)
x2 = np.random.normal(loc=3, scale=.5, size=50)
y2 = np.random.normal(loc=5, scale=.5, size=50)

# Combine the data
x = np.concatenate([x1, x2])
y = np.concatenate([y1, y2])

# Create labels: 0 for the first mode, 1 for the second
labels = np.array([0] * 50 + [1] * 50)

# Stack the features together
X = np.vstack((x, y)).T

# Add intercept term to X
X = np.hstack((np.ones((X.shape[0], 1)), X))

# Split the dataset into training and testing sets (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.3,
    random_state=0)

# Define the sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Logistic regression with gradient descent and L2 regularization
def logistic_regression_l2(X, y, learning_rate=0.01, iterations=1000, lambda_=0.1)
    :
    m, n = X.shape
    theta = np.zeros(n)

    for _ in range(iterations):
        z = np.dot(X, theta)
        hypothesis = sigmoid(z)
```

Figure 2.4: Logistic regression using L_2 regularization. Same data set as least squares classifier. The line plotted in green is the decision boundary

```
        gradient = (np.dot(X.T, (hypothesis - y)) / m) + (lambda_ / m) * theta
        # Do not regularize the intercept term
        gradient[0] -= (lambda_ / m) * theta[0]
        theta -= learning_rate * gradient

    return theta

# Train the logistic regression model with L2 regularization on the training data
theta = logistic_regression_l2(X_train, y_train, learning_rate=0.1, iterations
    =1000, lambda_=0.1)

# Predict on the testing data
z_test = np.dot(X_test, theta)
predictions = sigmoid(z_test) >= 0.5

# Calculate accuracy
accuracy = accuracy_score(y_test, predictions)
print(f'Accuracy on the testing data: {accuracy * 100:.2f}%')
```

The output of our accuracy is

```
Accuracy on the testing data: 100.00%
```

We include the plot of the classified test data in Figure 2.4

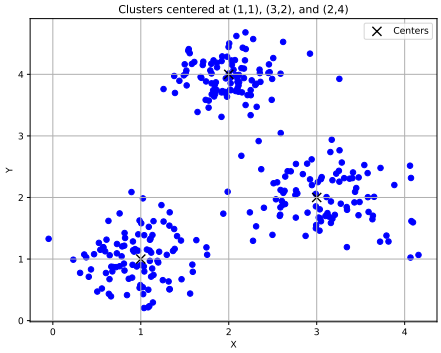

Figure 2.5: Randomly generated 3 clusters

2.3 Multiclass Classification

Once we have an understanding of logistic regression with 2 classes, doing multiclass classification is just a generalization of the 2 class case. More specifically, the the conditional probability now becomes

$$\mathbb{P}(\boldsymbol{y} \mid \boldsymbol{g(\theta)}) = \prod_{k=1}^{K} g(\theta)_k^{y_k}$$

. Where $g(\theta)_k$ is the probability of the of y_k belong to the class $K = k$. It plays the same role that $\sigma(z)$ did in logistic regression. $\theta_k^{y_k}$ can be solved explicitly as

$$\mathbb{P}(y = k \mid \mathbf{x}, \theta) = \frac{\exp(\theta_k \cdot \mathbf{x})}{\sum_{j=1}^{K} \exp(\theta_j \cdot \mathbf{x})} \tag{2.6}$$

Intuitively, this is just a generalization of the sigmoid function called the **softmax** function. Going through the motion, we find that the probability of the observed data point given $\boldsymbol{w}$ and $\boldsymbol{x}$ is

$$\mathbb{P}\left(\boldsymbol{y} \mid \boldsymbol{x}, \{\boldsymbol{\theta}_k\}_{k=1}^{K}\right) = \prod_{k=1}^{K} \left(\frac{\exp\left\{\boldsymbol{x}^\top \boldsymbol{\theta}_k\right\}}{\sum_{k'=1}^{K} \exp\left\{\boldsymbol{x}^\top \boldsymbol{\theta}_{k'}\right\}} \right)^{y_k} \tag{2.7}$$

The likelihood function can be computed as

$$L(\theta) = \prod_{i=1}^{N} \prod_{k=1}^{K} \left(\frac{\exp(\theta_k \cdot \mathbf{x}_i)}{\sum_{j=1}^{K} \exp(\theta_j \cdot \mathbf{x}_i)} \right)^{y_{ik}}$$

And then computing the log likelihood

$$\mathscr{L}(\theta) = \log L(\theta) = \sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log\left(\frac{\exp(\theta_k \cdot \mathbf{x}_i)}{\sum_{j=1}^{K}\exp(\theta_j \cdot \mathbf{x}_i)}\right)$$

Asking mathematica to simplify this expression for us gives

$$\mathscr{L}(\theta) = \sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik}\left(\theta_k \cdot \mathbf{x}_i - \log\left(\sum_{j=1}^{K}\exp(\theta_j \cdot \mathbf{x}_i)\right)\right)$$

I know that I flip flop between matrix notation and vector notation but in the end it shouldn't matter because it's just the dot product (a scalar). Now, similar to how we can solve logistic regression exactly, we cannot solve this exactly either. We will have to use gradient descent to maximize the likelihood. It should be pretty straightforward that

$$\nabla_\theta \mathscr{L} = \sum_{i=1}^{N}\left(y_{ik}\mathbf{x}_i - \frac{\exp(\theta_k \cdot \mathbf{x}_i)}{\sum_{j=1}^{K}\exp(\theta_j \cdot \mathbf{x}_i)}\mathbf{x}_i\right) = \sum_{i=1}^{N}\mathbf{x}_i\left(y_{ik} - g(\theta)\right) \tag{2.8}$$

After some hardy simplifications! Let us do a quick example. Suppose now we have 3 clusters to classify. We will run softmax regression using gradient descent using the following code

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Softmax regression using gradient descent
class SoftmaxRegressionGD:
    def __init__(self, learning_rate=0.01, n_iterations=1000, regularization=0.01)
    :
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.regularization = regularization
        self.theta = None

    def softmax(self, z):
        exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  # for numerical
    stability
        return exp_z / np.sum(exp_z, axis=1, keepdims=True)

    def fit(self, X, y):
```

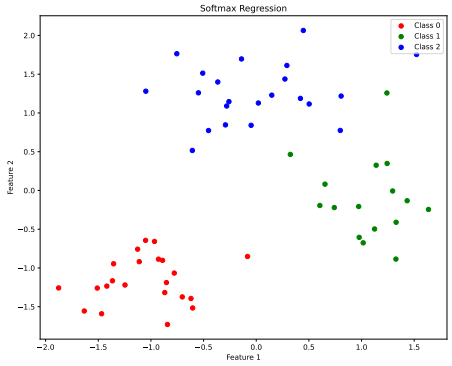

Figure 2.6: Softmax regression of data from figure 2.5

```
        m, n = X.shape
        k = len(np.unique(y))
        y_one_hot = np.eye(k)[y.astype(int)]
        self.theta = np.zeros((n, k))

        for _ in range(self.n_iterations):
            scores = X.dot(self.theta)
            probs = self.softmax(scores)
            gradient = -1/m * X.T.dot(y_one_hot - probs) + self.regularization * self.theta
            self.theta -= self.learning_rate * gradient

    def predict(self, X):
        scores = X.dot(self.theta)
        probs = self.softmax(scores)
        return np.argmax(probs, axis=1)

# Train the model
model = SoftmaxRegressionGD(learning_rate=0.1, n_iterations=1000, regularization=0.01)
model.fit(X_train_scaled, y_train)

# Predict the test set
y_pred = model.predict(X_test_scaled)

# Calculate the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

With an output accuracy of

```
Accuracy: 0.97
```

Peep Figure 2.6 for the results.

3

Basis Expansion And Regularization

"Basis (Domain) Expansion!"

– Gojo

Up until now, we've been working with data that could be fit by a linear curve or classified using a linear separation boundary. Sometimes, or maybe all the time, we might desire a nonlinear regression. How can we modify what we've built to be able to handle that?

3.1 Choosing A Basis

A thing to emphasize is that when we did linear regression, we CHOSE to use a line to fit the data. We could have chosen any function that we wanted to; it was just that linear functions have been well studied throughout history. In the same vein, choosing what linear function to fit is a choice that we have to make. Suppose that we have data that look like in Figure 3.1 When we can plot the data out like this, perhaps we can make the guess that perhaps the line that fits this will be quadratic (in the future when we talk about higher dimensional data, we might not be afforded this luxury). To tackle this problem, we will do an expansion in the basis of functions to fit:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + ... \tag{3.1}$$

If we believe that the ground truth function will be quadratic, then we will only need to worry until the x^2 term. When trying to learn the θ values, we can map this to multiple

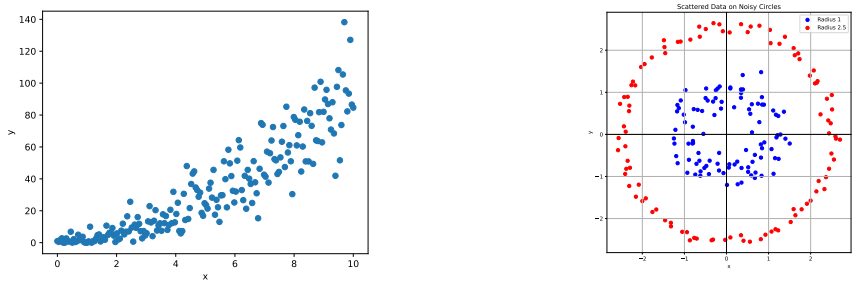

Figure 3.1: On the left we plot quadratic function to fit using least squares. On the right, we aim to classify using logistic regression

regression, where treat x_i^2 as just another feature. Written out in matrix form, this just means that our feature matrix will be

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ \vdots & \vdots & \vdots \\ 1 & x_{100} & x_{100}^2 \end{bmatrix}$$

So then we can learn the θ coefficients multiplying the basis functions x_i^n either using the normal equations or gradient descent. Either way, we plot the results in Figure 3.2. We also include the code:

```
x = [...] # Create your own data set
y = x**2 + ... # Create your own data set

# Create feature matrix with quadratic term
X = np.vstack((np.ones(len(x)), x, x**2)).T

# Solve for the theta coefficients using the normal equation
theta = np.linalg.inv(X.T @ X) @ X.T @ y

# Predict y values using the model
y_pred = X @ theta
```

So in the case of curve fitting, this is really straightforward. All that really changes is the feature matrix. What about logistic regression? If we had data shown in figure 3.1, we can

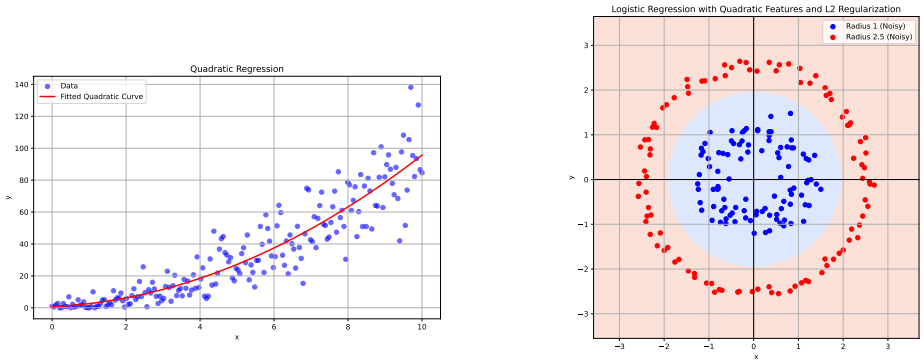

Figure 3.2: On the left we plot quadratic function to fit using least squares. On the right, we aim to classify using logistic regression

deduce that the decision boundary has to be a circle. The equation for a circle is given as

$$x_1^2 + x_2^2 = c$$

This the basis expansion would be

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2$$

We can do the exact same thing that we did before for least squares where we just treat the quadratic terms as extra features in the exponent of the logistic function (Figure 3.2). The code for this is given:

```
# Create feature matrix with quadratic terms
X_circle_1 = np.vstack((x_circle_1_noisy, y_circle_1_noisy)).T
X_circle_2 = np.vstack((x_circle_2_noisy, y_circle_2_noisy)).T
X = np.vstack((X_circle_1, X_circle_2))
X_quadratic = np.hstack((np.ones((X.shape[0], 1)), X, X**2))  # Add intercept and
    quadratic terms

# Create labels
y = np.hstack((np.zeros(n_points), np.ones(n_points)))

# Standardize features
scaler = StandardScaler()
X_quadratic_scaled = scaler.fit_transform(X_quadratic)

# Run logistic regression
```

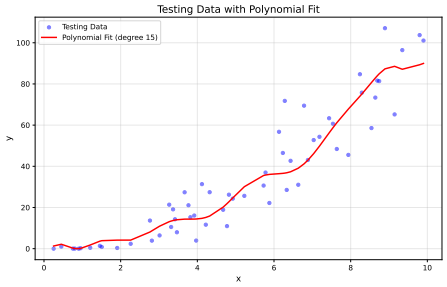

Figure 3.3: Fitting the quadratic data to degree n=15 polynomial.

```
15 theta = logistic_regression_l2(X_quadratic_scaled, y, learning_rate=0.01,
       iterations=10000, lambda_=0.1)
```

3.2 Overfitting: The Burden Of Choice

Once we open the floodgates to all functions, we go from infinite possibilities to even bigger infinite possibilities. We mentioned before that when the data is 2D, we have the privilege of being able to plot it and see what function to choose. In high dimension, it's impossible to visualize and so the question becomes: how do we know what function to choose? How do we know that we are fitting properly? There are a couple answers but none of them give us the exact answer. To see why this is a difficult question to ask; attempt to fit the quadratic function with a degree 15 polynomial. We see that the curve has more degrees of freedom to fit closer to training data, but could blow up or be wildly incorrect for unseen testing data. For example, if we train on only the first half of the data then test the second half, we could see that the curve does a hoirrble job predicting (Figure 3.4). So what do we see? We see that if we increase the model complexity, the accuracy on the training data will increase but then the accuracy on the testing data might suck. This an example of the **bias variance trade-off**. In the case where the training data has very high accuracy but predicts horribly, then we say that the data is **overfitted**. We say that it is **underfitted** if the opposite happens. We tend to care more about overfitting because our models are more susceptible to overfitting.

3.2.1 Bias Variance Tradeoff

In statistics, if we might want to ask "how for off are our estimators for the data?" Since our estimators are functions of the data points (which are random variables), then we should expect that our estimators are also random variables. In that case, it might be tempting

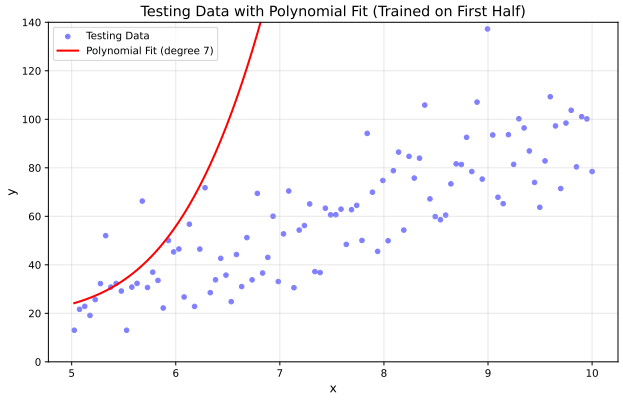

Figure 3.4: testing curve on unseen data with degree n=7. We could not include n=15 because it blew up to infinity way too fast

to say that a measure of how "wrong" we are is just given by the expectation of the true population parameter minus the estimator. But it's quite easy to show that you can have an unbiased estimator and always be wrong; so this is not the most ideal. Similar to the least squares issue, you might have data points above and below the average. We will circumvent this by squaring the difference and taking the expectation

$$\text{MSE} = \mathbb{E}[(f(x) - \hat{f}(x))^2] \tag{3.2}$$

where the MSE is called the **mean squared error**. We start with

$$f(x) - \hat{f}(x) = (f(x) - \mathbb{E}[\hat{f}(x)]) + (\mathbb{E}[\hat{f}(x)] - \hat{f}(x))$$

Which, all that we did was insert a 0 term $\mathbb{E}[\hat{f}(x)]) + (\mathbb{E}[\hat{f}(x)]$ Then squaring the above gives

$$(f(x) - \hat{f}(x))^2 = [(f(x) - \mathbb{E}[\hat{f}(x)]) + (\mathbb{E}[\hat{f}(x)] - \hat{f}(x))]^2$$

Expanding the square

$$= (f(x) - \mathbb{E}[\hat{f}(x)])^2 + 2(f(x) - \mathbb{E}[\hat{f}(x)])(\mathbb{E}[\hat{f}(x)] - \hat{f}(x)) + (\mathbb{E}[\hat{f}(x)] - \hat{f}(x))^2$$

Taking the expectation on both sides. We note that the cross term $\mathbb{E}[2(f(x)-\mathbb{E}[\hat{f}(x)])(\mathbb{E}[\hat{f}(x)]-\hat{f}(x))]$ is 0 because $f(x) - \mathbb{E}[\hat{f}(x)]$ is a constant with respect to the randomness in $\hat{f}(x)$. Then, this can be simplified to

$$\mathbb{E}[(f(x)-\hat{f}(x))^2] = \mathbb{E}[(f(x)-\mathbb{E}[\hat{f}(x)])^2]+\mathbb{E}[2(f(x)-\mathbb{E}[\hat{f}(x)])(\mathbb{E}[\hat{f}(x)]-\hat{f}(x))]+\mathbb{E}[(\mathbb{E}[\hat{f}(x)]-\hat{f}(x))^2]$$

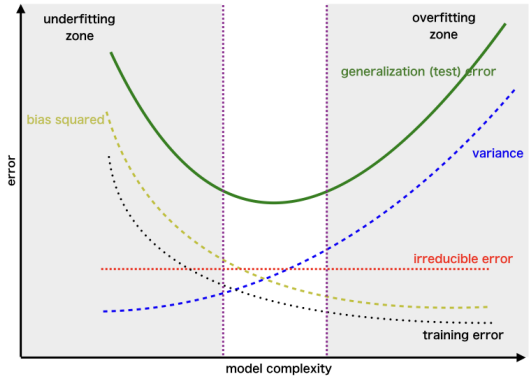

Figure 3.5: Demonstration of bias variance tradeoff.

$$= \mathbb{E}[(f(x) - \hat{f}(x))^2] = (f(x) - \mathbb{E}[\hat{f}(x)])^2 + \mathbb{E}[(\mathbb{E}[\hat{f}(x)] - \hat{f}(x))^2]$$

Defining $(f(x) - \mathbb{E}[\hat{f}(x)])^2$ to be bias squared and $\mathbb{E}[(\mathbb{E}[\hat{f}(x)] - \hat{f}(x))^2]$ to be the variance, we can see that the MSE can be decomposed into

$$\text{MSE} = \text{bias}^2(\hat{f}(x)) + \text{Var}[\hat{f}] \tag{3.3}$$

What does this equation mean? It means that error in our estimator can always be decomposed into the bias and variance of the estimator. Since the error will always exist (we can't have exact fit), if you try to decrease the bias (accuracy, basically) the variance will have to increase (the stability of your model). If you try to decrease your variance, then the bias has to increase. Unfortunately, we cannot eat for free in this world. But looking at Figure 3.5, we see that although the bias and variance have too trade off, there actually exists (maybe) some sort of optimal answer, where we minimize the squared error for some model complexity. How do we know what optimizes the curve? It's very difficult, and this is the human part of machine learning. We have to be the ones to decide when to stop. We can use the bias variance tradeoff to guide us, but ultimately the decision is ours. I should note however, that this is result is mostly theoretical. In work, it's often expensive to try to run multiple models and simulations to see what works. In that case, the discretion is even more on us.

3.3 Regularization

I won't dwell too much on here, because we already touched on it a bit in chapter 1. One way to tame the volatility of our model is to regularize it. What do we mean by regularization exactly? What we mean is we have introduce a bias term, usually in the form of $\lambda||\theta||^2$ as

we saw in Ridge regression. Before, we saw that introducing this term in the least squares cost function also introduces λ to the diagonals of the normal equation. What this is really doing is introducing bias into our model. To see this suppose we add a regularization term to the cost

$$\text{Cost} = \text{MSE} + \lambda \sum \theta_i^2$$

To minimize the cost function with the constraint of the Ridge term, θ values that were once very very large have to shrink. This means that the model might not fit to the data as well (increasing the bias) but now the model will not be as susceptible to blowing up (decreasing variance). There are other regularization methods to consider than just Ridge. For example, the LASSO regularization

$$\text{Cost} = \text{MSE} + \lambda \sum ||\theta_i|| \tag{3.4}$$

We won't go into it here but LASSO performs similarly to Ridge but instead of shrinking the large θ coefficients, it sends them to 0 entirely. If we wanted to implement gradient descent on this, then we would need to take the gradient of the absolute value function, which we know is just the $\text{sgn}(x)$ function. Are there any other regularizations to consider? In general, we could use the p norm

$$L_p = ||\theta^p||^{1/p}$$

Basically to summarize: to make sure that our models are stable, it's good practice to regularize. What type of regularization we choose is up to us and depends on what we are trying to do.

4

Neural Networks

"Oh yeah, it's big brain time"

– Megamind

In a reductionist statement, machine learning is about function estimation. In the sections prior, our goal was to estimate the function in which the data is generated/separated by. This typically involved a single function with multiple parameters θ. The point of a neural network is to have multiple functions contribute to the estimation simultaneously. This idea was inspired by the architecture of the brain, where neurons are interconnected and information can be sent via electrical currents. We will walk through how a neural network estimates functions (feed forward) and how neural networks learn the parameters to do the estimation (back propagation).

4.1 Neural Network Architecture

A neural network has 3 basic components: a node, a layer, and an activation function. Each circle is called a node, and it stores some information about the function. A column of nodes is called a layer. Each node in 1 layer is connected to another node in a different layer through an activation function. The layers in between the input layer and the output layer are called the hidden layers. Peep Figure 4.1 for a representation. The input layer takes in a value (or a set of values depending on the architecture) and sends them to the hidden layers via an activation function. We are the ones to choose the activation function, but ultimately the activation function just transforms the input in some way and spits out an

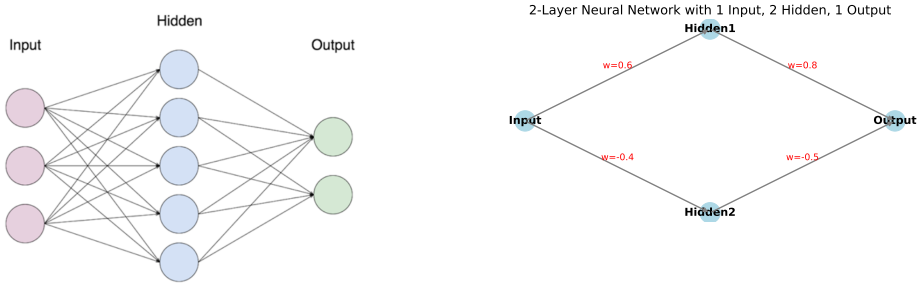

Figure 4.1: On the left is a general schematic of a neural network. On the right is a specific neural network with weights.

output. This could be done many times. We will treat the hidden layer for now as a black box, but you can effectively treat it as 1 giant series of transformations. The final layer outputs or result through one final activation function. Much like how current is propagated through a network of neurons, you can imagine information being propagated through the layers in much the same way.

4.2 Feed Forward Algorithm

Suppose that we have the neural network shown in Figure 4.1; essentially constant weights. Suppose we have an input $x_0 = 1$. Then how does this input get transformed? On the connection, we multiply by 0.6, so now hidden 1 takes on a value of 0.6. Similarly, hidden 2 takes on a value of -0.4. Once we have the values for hidden 1 and hidden 2, we act on it with the activation function of the hidden layer. Let's suppose that we choose the activation function to be ReLU, which is the rectified linear function (x for $x > 0$ and 0 else). In that case, hidden 1 just stays as it is while hidden 2 becomes 0. We multiply hidden 1 by 0.8 and add to it $-.5$*hidden 2 (which is just 0). We add up the results and we get a final answer of 0.48. This is for a single input data. We can imagine that if we passed in a function $f(x)$ that the neural network would give something different. This is just a demonstration of how the architecture works. In reality, the weights w may not necessarily be a scalar multiplication, they can be multiplication with an additional bias (Figure 4.2)

4.2.1 Vectorized Feed Forward

Like with everything that we are going to do, we will want to vectorize this so that the computation will not be as costly. Let's start with the input layer. Since the input only has 1 node, it is a scalar x. The weights w_{11} and w_{21} can then be written as a 2x1 column

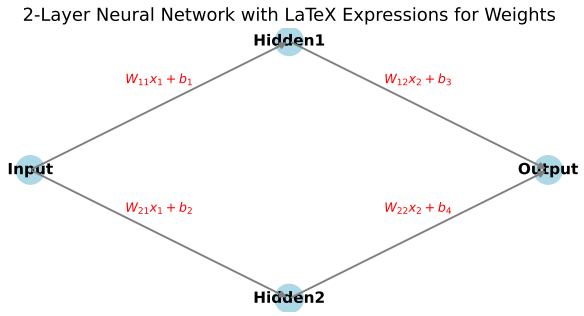

Figure 4.2: The neural network in 4.1(b) but with general weights and biases

matrix $\boldsymbol{W_1}$ (just by examination of the indices). The bias vector $\boldsymbol{b}$ is also a 2x1 column vector. Thus the first computation is

$$x \implies \boldsymbol{W_1}x + \boldsymbol{b_1}$$

Once we've hit the first hidden layer, the transformed x is hit with the activation function. There are many activation functions to choose from, but the most popular ones are the sigmoid function $\sigma(z)$ as seen previously in logistic regression and the ReLU function $r(x) = \max\{0, x\}$. Let us stick to ReLU. How we hit the transformed input with the ReLU function to get

$$\boldsymbol{W}_1 x + \boldsymbol{b}_1 \implies \max\{0, \boldsymbol{W_1}x + \boldsymbol{b_1}\} = \boldsymbol{u}$$

Remember that the output is still a vector. Let us call that output vector $\boldsymbol{u}$. Passing through from the hidden layer to the output, we multiply by the second set of weights

$$\boldsymbol{u} \implies \boldsymbol{W}_2\boldsymbol{u} + \boldsymbol{b}_2 = v$$

Since the output is a scalar sum of the elements of $\boldsymbol{v}$, we can just take the dot product of $\boldsymbol{v}$ with the identity vector (vector of all 1). Combining our result, we see that the vectorized algorithm is

$$x \implies \boldsymbol{W_1}x + \boldsymbol{b_1} \implies \max\{0, \boldsymbol{W_1}x + \boldsymbol{b_1}\} = \boldsymbol{u} \implies \boldsymbol{W}_2\boldsymbol{u} + \boldsymbol{b}_2 = \boldsymbol{v} \implies v \tag{4.1}$$

Let us write code to compute the generic forward pass

```
import numpy as np

# ReLU activation function
def relu(x):
```

```
    return np.maximum(0, x)

# Initialize random weights and biases
np.random.seed(0)  # For reproducibility
W_input_hidden = np.random.rand(2, 1)  # 2 hidden nodes, 1 input node
b_hidden = np.random.rand(2, 1)        # Biases for hidden layer

W_hidden_output = np.random.rand(1, 2) # 1 output node, 2 hidden nodes
b_output = np.random.rand(1, 1)        # Bias for output layer

# Input vector (1 input node)
x_input = np.array([[1]])

# Forward pass
# Compute hidden layer activations
z_hidden = np.dot(W_input_hidden, x_input) + b_hidden
a_hidden = relu(z_hidden)

# Compute output layer activations
z_output = np.dot(W_hidden_output, a_hidden) + b_output
a_output = z_output  # Assuming linear activation for output

print("Hidden layer activations:", a_hidden)
print("Output layer activations:", a_output)
```

It's a good exercise to examine the code and analyze the dimensions of the weight matrices. They are not all the same dimension.

4.3 Back Propagation

Now that we have a basic understanding of how neural networks compute, we need to discuss how neural networks learn. Surprisingly, the mechanism for learning is similar to the back propagating action potential in actual neurons, so we use that name to call the algorithm the **back propagation algorithm**. What is the goal? The goal per usual is to minimize the cost function, but what IS the cost function here? We will still use least squares!

$$J = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{4.2}$$

We should be careful: this is not the exact same as linear regression. This is because our predictors $\hat{y}_i$ are not just simple functions of the weights $\boldsymbol{w}$; they are compositions of functions of $\boldsymbol{w}$. Thus, there are many different weights to optimize simultaneously, and they may be functions of each other. So when we take a derivative, we have to be mindful of

the chain rule. Typically, when we optimize the weights, we have to find the derivative of weight and do gradient descent on the weights. We do this going backwards function by function, hence the name back propagation. Let us do a calculation. Starting with the cost function in matrix notation

$$J = (y - v)$$

$$\implies = (y - \boldsymbol{W}_2\boldsymbol{u} + \boldsymbol{b}_2)^2$$

What parameters do we want to optimize? We want to optimize $\boldsymbol{W}_2$ and $\boldsymbol{b}_2$. Starting with $\boldsymbol{b}_2$, we find that the derivative is

$$\frac{\partial J}{\partial \boldsymbol{b}_2} = 2\left(y - \boldsymbol{W}_2\boldsymbol{u} + \boldsymbol{b}_2\right)$$

Since the term is linear, the derivative of the inside with respect to $\boldsymbol{b}_2$ is 1. We see that we are going to see the term $2\left(y - \boldsymbol{W}_2\boldsymbol{u} + \boldsymbol{b}_2\right)$ a lot so we are going to define it to be $\boldsymbol{\delta}_2$. Doing so, we find that the gradient for $\boldsymbol{b}_2$ is $\boldsymbol{\delta}_2$. What about the weight matrix $\boldsymbol{W}_2$? We still get the $\boldsymbol{\delta}_2$ but now the derivative of the inside is no longer just 1, we need to find

$$\frac{d}{d\boldsymbol{W}_2}\boldsymbol{W}_2\boldsymbol{u} = \boldsymbol{u}^T$$

which is just $\boldsymbol{u}^T$ by definition. Thus the gradient for $\boldsymbol{W}_2$ is $\boldsymbol{\delta}_2\boldsymbol{u}^T$. Here is where the chain rule really shines! We want to compute the gradient term for $\boldsymbol{W}_1$ and $\boldsymbol{b_1}$. To do so, we need to compute

$$\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{W}_1} \quad \& \quad \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{b}_1}$$

Remember that we passed $\boldsymbol{W}_1 x + \boldsymbol{b}_1$ through the ReLU function. To reiterate what we are trying to do, by the chain rule we know

$$\frac{\partial J}{\partial \boldsymbol{W_1}} = \frac{\partial J}{\partial \text{RSS}} \frac{\partial \text{RSS}}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{W}_1}$$

Where RSS is the sum of squared residual term (basically the least squares term). The derivative of the RSS with respect to $\boldsymbol{u}$ is just $\boldsymbol{W}_2^T$ by definition. Then, by chain rule, if we remember that

$$\boldsymbol{u} = \text{ReLU}\left(\boldsymbol{W}_1 x + \boldsymbol{b_1}\right)$$

Taking the derivative of $\boldsymbol{u}$ with respect to $\boldsymbol{W}_1$ requires that knowing the derivative of the ReLU function. But fear not! We know what the derivative of the ReLU(x) function is: it's

just the Heaviside theta function $\Theta(x)$! Thus,

$$\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{W}_2} = \Theta(\boldsymbol{W}_1 x + \boldsymbol{b_1}) \frac{d}{d\boldsymbol{W}_1} \boldsymbol{W}_1 x = \Theta(\boldsymbol{W}_1 x + \boldsymbol{b_1}) x$$

If we can clean up the terms a bit by defining $\Theta(\boldsymbol{W}_1 x + \boldsymbol{b_1}) = \boldsymbol{\delta}_1$, then we can simultaneously solve for the derivatives of $\boldsymbol{u}$ with respect to $\boldsymbol{W}_1$ and $\boldsymbol{b}_1$

$$\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{W}_1} = \boldsymbol{\delta}_1 x \quad \& \quad \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{b}_1} = \boldsymbol{\delta}_1$$

Then, the full gradient is

$$\frac{\partial J}{\partial \boldsymbol{W}_1} = \boldsymbol{\delta}_2 \boldsymbol{W_2}^T \boldsymbol{\delta}_1 x \quad \& \quad \frac{\partial J}{\partial \boldsymbol{b}_1} = \boldsymbol{\delta}_2 \boldsymbol{W_2}^T \boldsymbol{\delta}_1 \tag{4.3}$$

And the gradient of the other weights are

$$\frac{\partial J}{\partial \boldsymbol{W}_2} = \boldsymbol{\delta_2} x \quad \& \quad \frac{\partial J}{\partial \boldsymbol{b}_2} = \boldsymbol{\delta}_2 \tag{4.4}$$

The gradient descent algorithm is pretty straightforward, you just subtract the current values of all the weights by the learning rate α times their respective gradients.

4.4 An Example

Neural networks are extremely powerful and are extremely versatile (part of what makes them so powerful). They can learn on a wide range of problems, including both regression and classification. We do perform by doing a classification problem. We generate 200 random numbers. This will be our input data. Our output data will be the the Heaviside Θ function on the input. We will train the neural network with 1 input node, 1 hidden layer of 5 nodes, and 1 output node. The activation function will be ReLU but you are free to choose whatever activation function you want as long as you remember to take the derivative carefully! We then split our data 80/20, train on the training data and then test our accuracy on the testing data. Here is the code

```
# ReLU activation function and its derivative
def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return (x > 0).astype(float)

# Generate example data
np.random.seed(1)
```

```
X = np.random.randn(200, 1)  # 200 samples, 1 feature
y = (X > 0).astype(float)    # Binary target

# Split the data into training and testing sets (80/20 split)
train_size = int(0.8 * X.shape[0])
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Initialize weights and biases
input_size = 1
hidden_size = 5  # More nodes in the hidden layer
output_size = 1

W_input_hidden = np.random.randn(hidden_size, input_size)
b_hidden = np.random.randn(hidden_size, 1)

W_hidden_output = np.random.randn(output_size, hidden_size)
b_output = np.random.randn(output_size, 1)

# Learning rate
learning_rate = 0.01

# Training loop
epochs = 1000
for epoch in range(epochs):
    # Forward pass for training data
    z_hidden = np.dot(W_input_hidden, X_train.T) + b_hidden
    a_hidden = relu(z_hidden)

    z_output = np.dot(W_hidden_output, a_hidden) + b_output
    a_output = z_output  # Linear activation for output

    # Cost (Mean Squared Error)
    cost = np.mean((a_output - y_train.T) ** 2)

    # Backward pass
    dz_output = a_output - y_train.T
    dW_hidden_output = np.dot(dz_output, a_hidden.T) / X_train.shape[0]
    db_output = np.sum(dz_output, axis=1, keepdims=True) / X_train.shape[0]

    dz_hidden = np.dot(W_hidden_output.T, dz_output) * relu_derivative(z_hidden)
    dW_input_hidden = np.dot(dz_hidden, X_train) / X_train.shape[0]
    db_hidden = np.sum(dz_hidden, axis=1, keepdims=True) / X_train.shape[0]

    # Update weights and biases
    W_hidden_output -= learning_rate * dW_hidden_output
    b_output -= learning_rate * db_output
```

```
    W_input_hidden -= learning_rate * dW_input_hidden
    b_hidden -= learning_rate * db_hidden

    # Print cost every 100 epochs
    if epoch % 100 == 0:
        print(f'Epoch {epoch}, Cost: {cost:.4f}')

# Test the network
z_hidden_test = np.dot(W_input_hidden, X_test.T) + b_hidden
a_hidden_test = relu(z_hidden_test)

z_output_test = np.dot(W_hidden_output, a_hidden_test) + b_output
a_output_test = z_output_test

# Convert continuous output to binary predictions
predictions = (a_output_test > 0.5).astype(float)

# Calculate accuracy
accuracy = np.mean(predictions.T == y_test) * 100
print(f'Accuracy on testing data: {accuracy:.2f}%')
```

doing this, we get an accuracy value of

```
Accuracy on testing data: 97.50%
```

Part II

Unsupervised Learning

5

K-Means and Gaussian Mixture Models

Previously, we have been doing supervised machine learning, which just means that the machine learns by finding correlations in the input data with the output data. the term supervised here means that we tell the model what the correct answer is a priori because there exists some ground truth. But now, what if there isn't a ground truth to cling onto? In this case, we cannot supervise the machine anymore and it must learn patterns in the underlying structure of the data. This method is called **unsupervised learning**. The problem that we will be trying to solve is the clustering problem, which we presented earlier in classification. We will explore 2 main topics: K-means which is a hard label clustering method and GMMs which are probabilistic models.

5.1 Problem Set Up

We are going to first set up the problem that we want to solve before we talk about the methods. Consider the plot in Figure 2.5, replotted in 5.1. The difference now is that we don't know what labels the points should be. That is, there is no output data to the corresponding input data. We want to cluster the data. That is, to classify what category each data point belongs to. This problem emphasizes the big difference between unsupervised learning and supervised learning: we don't know what the answer is for unsupervised learning. Furthermore, as we will see in a bit, the computer cannot see the data and group the points by eye as we might. When we try to minimize a cost function, there might be many many

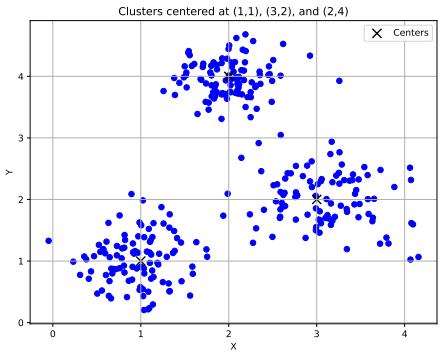

Figure 5.1: Replotted plot of clusters from Figure 2.5

local minimums that the gradient could approach to. We note that the data may not even need to be continuous in the example shown in Figure 5.1. You might have a heat map that you want to cluster the points to. The only thing that is needed in the intersection of all these problems is the existence of a notion of distance between points. In the case of Figure 5.1, because the data is continuous on $\mathbb{R}^2$, we can use the Euclidean distance. In fact, our choice of a distance doesn't really matter too much as long as we can define a distance.

5.2 The K-Means Algorithm

Let's first break down some notation and terminology. What is the k in k means? K is the number of clusters that we want to use to group the data. It is the human telling the machine how many groups there must exist. In our working example, we see by eye that k must be 3. In practice, we will not be afforded this luxury of visualizability. Before we can discuss what the means part means (ba dum tss), we need to define a centroid. A centroid is just the a point that is the center of mass of a group of points. That is, the weighted midpoint of a group of points. In our example, the centroids are given by the x markers at the center of each cluster. We can dissect the name k-means: the k is the number of centroids, and we find the centroids by finding the average distance of each data point to their nearest centroid. We cluster based on the point's proximity to its nearest centroid, with the idea of "nearest" being given by the defined distance (Euclidean, in our case). With just the anatomy of the name, we can easily write out the algorithm for K-means

K-Means Algorithm

(1) First choose a k. This is typically the hardest and easiest step, because how do we know what k will optimize our clustering?
(2) Randomly choose points to be the centroids.
(3) Find the distance (using whatever definition of distance we defined) of every point to those centroids. Label the points to the nearest centroid.
(4) Find the new center of mass by calculating the average position of every data point in the cluster
(5) Repeat steps 3-5 until no new clusters are formed or until told to stop

The k-means algorithm is a surprisingly simple algorithm, yet also surprisingly powerful. Let us see what it is capable of. We will create the code to run k-means on the data generated

```
# Number of clusters
k = 3

# Function to calculate the Euclidean distance
def euclidean_distance(a, b):
    return np.linalg.norm(a - b)

# Initialize centroids randomly from the data points
initial_centroids = X[np.random.choice(X.shape[0], k, replace=False)]

def k_means(X, k, initial_centroids, max_iters=100, tol=1e-4):
    centroids = initial_centroids
    for iteration in range(max_iters):
        # Step 2: Assign clusters
        clusters = [np.argmin([euclidean_distance(x, centroid) for centroid in
    centroids]) for x in X]

        # Step 3: Update centroids
        new_centroids = np.array([X[np.array(clusters) == j].mean(axis=0) for j in
     range(k)])

        # Check for convergence
        if np.all(np.linalg.norm(new_centroids - centroids, axis=1) < tol):
            break

        centroids = new_centroids

    return centroids, clusters

# Run k-means algorithm
final_centroids, cluster_assignments = k_means(X, k, initial_centroids)
```

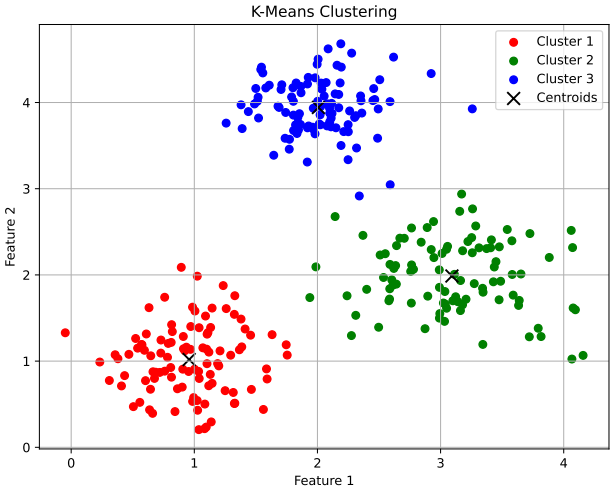

Figure 5.2: Clustering results using k-means with k = 3

We plot our results in Figure 5.1.

5.2.1 Convergence And Optimization

We mentioned before that there exists many local minimums that these centroids can converge to. The question now is: how do we know if the result is the optimal result if we can't plot it? Ther are a couple of ways, but not all of them are perfect (as the saying goes, if things were perfect, we wouldn't be doing machine learning). The first method is called **the elbow method**. In this case, we run k-means n times with n different values of k (e.g. k=1 to k=11). We compute the sum of squared distance of each point to their respective centroids. For some value of k, there might be a significant drop in the RSS and then subsequent k values marginally decrease the RSS (this point is called the **elbow** becuase it looks like an elbow). That is the value of k that we use for the number of clusters (peep Figure 5.3). What if the eblow method doesn't work? That is, what if the elbow method has no elbow and the decrease in RSS has no singificant jump? Or what if it has multiple significant jumps? There are still other methods to explore from. The other major method is the **silhouette score** method. This method is simple (but also quite costly). Let us pick a point x. We calculate the average distance of x to all other data points in its cluster. This is called **cohesion** and is given by $a(x)$. Next, we find the average distance of x to every point in the nearest cluster NOT INCLUDING its own. This is called the

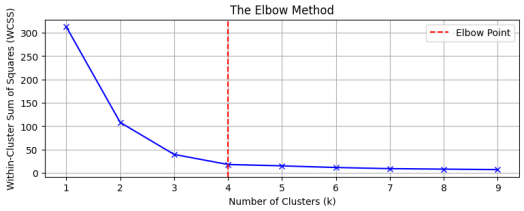

Figure 5.3: (For visualization purposes only, this is NOT from our data) notice that there is marginal decrease in the RSS (or inertia) after k=4, so we choose k = 4

separation $b(x)$. The silhouette score is given by

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \tag{5.1}$$

and measures how likely a point is to belonging to its cluster or its neighboring cluster. The score is given from -1 to 1, with -1 being the highest likelihood of misclassification while 1 is the highest likelihood of being in the correct cluster. A score of 0 means that the point is perfectly on the decision boundary. Some other methods we could use could be cross validation (if applicable) and stability tests. For example, we run k means with the same k multiple times with the same k value and see what is the most likely location of centroid and what is the most likely choice of clusters. We should note that for testing k-means, it is often pretty expensive because you have to run the algorithm multiple times. If you have massive amounts of data points (which oftentimes you might), then this will be very costly. To learn more about this, consider looking into information theory.

5.3 Gaussian Mixture Models

In this section, we discuss a softer form of clustering called **Gaussian mixture models**. The premise of this is pretty straightforward, we assume that each cluster is normally distributed with μ_k and σ_k, where k is the k-th cluster. To show how a computation is done, let us begin a mixture of 2 Gaussians, $\mathcal{N}(1,1)$ and $\mathcal{N}(3,2)$. With probability $\pi_1 = .3$, choose cluster 1 which has a normal $\mathcal{N}(1,1)$ distribution and with probability .7 choose cluster 2 which has normal $\mathcal{N}(3,2)$. The parameter π_k is probability of a specific component

$$\mathbb{P}(k) = \pi_k$$

It is a vector of probabilities called the **mixing coefficient** which is a **latent variable** of the model because it is not something that we can observe. Alternatively, the data points coming from the different normal distributions are called **observables**, because they are the

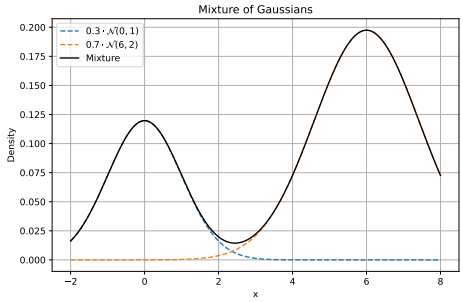

Figure 5.4: Mixture of Gaussians with different means and standard deviations but mixing coefficients .3 and .7.

things that we do observe. The distribution for π_k will always be multi-Bernoulli (sometimes misnamed multinomial). Using the law of total probability, we can find the distribution of the whole data set

$$f_X(\mathbf{x}) = \sum_z \mathbb{P}(z) f_X(\mathbf{x} \mid z)$$

In our example, the distribution of the data would be

$$f_X(\mathbf{x}) = .3 * \text{Normal}(0, 1) + .7 * \text{Normal}(6, 2)$$

An exaggerated plot of a mixture of Gaussian is plotted in Figure 5.4. Once we have all the mixing coefficients and the mean and standard deviations known, what is the probability of that a data point x is classified into cluster k? That is, what is $\mathbb{P}(k \mid \mathbf{x})$? We solved this rather pretty straightforward by using Baye's theorem

$$\mathbb{P}(k \mid \mathbf{x}) \propto \mathbb{P}(k)\mathbb{P}(\mathbf{x} \mid k)$$

Remember that there exists a $\propto$ since the densities might not sum to 1 but that's ok we can just renormalize them once we have them. Do we have all the parts that we need to compute the probability of a point belonging to a cluster? Yes! $\mathbb{P}(k) = \pi_k$ and $\mathbb{P}(\mathbf{x} \mid z)$ is the Gaussian of that specific cluster. For example, suppose we observe $x = 2$. What is the probability that it belongs to cluster k=1? Or k=2? Given that $\mathbb{P}(k = 1) = .3$ and $\mathbb{P}(k = 2) = .7$,

$$\begin{aligned}
\mathbb{P}(k = 1 \mid x) &= \frac{\mathbb{P}(k = 1) f_X(x \mid k = 1)}{\mathbb{P}(k = 1) f_X(x \mid k = 1) + \mathbb{P}(k = 2) f_X(x \mid z = 2)} \\
&= \frac{0.7 \cdot 0.054}{0.7 \cdot 0.054 + 0.3 \cdot 0.027} \approx 0.824
\end{aligned}$$

To find the probability of getting the second cluster, it would just be the complement of the above, which is approximately .176. What this telling us is that there is an 82% probability that the observation $x = 2$ belongs to $k = 1$. The reason why Gaussian mixture models are called softer clustering is that they don't necessarily tell us exactly what cluster each observation belongs to; they just give us a probability of the point belonging to that cluster. If we wanted to turn GMMs into something like k-means, we would choose the classification with higher probability and force that point into the cluster.

5.3.1 How Do Gaussian Mixture Models Learn?

There are 2 ways that Gaussian mixture models can learn: either gradient based or through expectation maximization. Either way, we need to first find the derivative of the loglikelihood

$$\frac{\mathrm{d}}{\mathrm{d}\theta} \log \mathbb{P}(\mathbf{x}) = \frac{\mathrm{d}}{\mathrm{d}\theta} \log \sum_z \mathbb{P}(z, \mathbf{x})$$

Then carrying out the derivative

$$= \frac{\frac{\mathrm{d}}{\mathrm{d}\theta} \sum_z \mathbb{P}(z, \mathbf{x})}{\sum_{z'} \mathbb{P}\left(z', \mathbf{x}\right)}$$

$$= \frac{\sum_z \frac{\mathrm{d}}{\mathrm{d}\theta} \mathbb{P}(z, \mathbf{x})}{\sum_{z'} \mathbb{P}\left(z', \mathbf{x}\right)}$$

Now we are going to do a trick. We are going to introduce a log into the derivative of the numerator

$$= \frac{\sum_z \mathbb{P}(z, \mathbf{x}) \frac{\mathrm{d}}{\mathrm{d}\theta} \log \mathbb{P}(z, \mathbf{x})}{\sum_{z'} \mathbb{P}\left(z', \mathbf{x}\right)}$$

This is mathematically equivalent to multiplying and dividing by 1. The reason we do this is so that we can extract an expectation like so

$$= \sum_z \left(\frac{\mathbb{P}(z, \mathbf{x})}{\sum_{z'} \mathbb{P}\left(z', \mathbf{x}\right)} \right) \frac{\mathrm{d}}{\mathrm{d}\theta} \log \mathbb{P}(z, \mathbf{x})$$

using the law of total expectation, we can write the term in the parenthesis as

$$\sum_z \mathbb{P}(z|\mathbf{x}) \frac{\mathrm{d}}{\mathrm{d}\theta} \log \mathbb{P}(z, \mathbf{x})$$

Which, by definition, is the conditional expectation

$$= \mathbb{E}_{p(z|\mathbf{x})} \left[\frac{\mathrm{d}}{\mathrm{d}\theta} \log p(z, \mathbf{x}) \right] \tag{5.2}$$

What this tells us is that the derivative of the log likelihood boils down to computing the expectation of the log derivative. For example, suppose we have the mixture model previously established, we can find the gradient of the log likelihood with respect to the first μ_1 as

$$\frac{\partial}{\partial \mu_1} \log \mathbb{P}(x) = \mathbb{E}_{\mathbb{P}(z|x)} \left[\frac{\partial}{\partial \mu_1} \log \mathbb{P}(z, x) \right]$$

Using Bayes' theorem, we can expand the log joint probability as

$$= \mathbb{E}_{\mathbb{P}(z|x)} \left[\frac{\partial}{\partial \mu_1} \log \mathbb{P}(z) + \frac{\partial}{\partial \mu_1} \log \mathbb{P}(x|z) \right]$$

Now, since the mixing coefficient $\mathbb{P}(z)$ does not depend on μ, this derivative gets killed. We are now left with the second term. However, only the case for $z = 1$ survives, because for the case of $z = 2$, we have μ_2 and μ_2 is independent of μ_1 so it gets killed by the derivative

$$= \mathbb{P}(z = 1 \mid x) \left[\frac{\partial}{\partial \mu_1} \log \mathbb{P}(x \mid z = 1) \right]$$

And since we've already computed $\mathbb{P}(z = 1 \mid x)$ previously, and using the fact that our mixture models are Gaussian, we can write the log derivative and compute the expectation to be

$$0.824 \cdot \frac{x - \mu_1}{\sigma_1^2} \tag{5.3}$$

For multiple training cases, this can easily be generalized to

$$\frac{\partial \ell}{\partial \mu_1} = \sum_{i=1}^{N} \mathbb{P}\left(z^{(i)} = 1 \mid x^{(i)} \right) \cdot \frac{x^{(i)} - \mu_1}{\sigma_1^2} \tag{5.4}$$

A skeleton code might give us

```
# Number of clusters
k = 3
n_samples, n_features = X.shape

# Initialize means, covariances, and mixing coefficients
means = np.random.rand(k, n_features) * 5
covariances = [np.eye(n_features) for _ in range(k)]
mixing_coefficients = np.ones(k) / k

# Learning rate for gradient descent
learning_rate = 0.01

# Gaussian PDF function
def gaussian_pdf(x, mean, cov):
```

```
    diff = x - mean
    return np.exp(-0.5 * diff.T @ np.linalg.inv(cov) @ diff) / (np.sqrt((2 * np.pi
    ) ** n_features * np.linalg.det(cov)))

# Gradient-based update
max_iter = 100
for iteration in range(max_iter):
    # Compute responsibilities
    responsibilities = np.zeros((n_samples, k))
    for i, x in enumerate(X):
        for j in range(k):
            responsibilities[i, j] = mixing_coefficients[j] * gaussian_pdf(x,
    means[j], covariances[j])
        responsibilities[i, :] /= np.sum(responsibilities[i, :])

    # Update means using the gradient
    for j in range(k):
        weighted_sum = np.sum(responsibilities[:, j, None] * X, axis=0)
        means[j] += learning_rate * (weighted_sum - means[j] * np.sum(
    responsibilities[:, j])) / np.sum(responsibilities[:, j])

    # Optionally update covariances and mixing coefficients (simplified)
    N_k = np.sum(responsibilities, axis=0)
    for j in range(k):
        diffs = X - means[j]
        covariances[j] = np.sum(responsibilities[:, j, None, None] * np.einsum('ni
    ,nj->nij', diffs, diffs), axis=0) / N_k[j]
        mixing_coefficients[j] = N_k[j] / n_samples

# Print final means
print("Final means:", means)
```

On your own, you can write code to find the variance. Now, in order to get something meaningful, we would have to run this algorithm many many many times for it to converge to something accurate. Is there an alternative? Yes!

5.3.2 Expectation Maximization Algorithm

The expectation maximization algorithm is a statistical procedure that makes use of the closed form gradient term in 5.4. Let us take equation 5.4 and set the derivative equal to 0. For simplicity, we will call $\mathbb{P}\left(z^{(i)} = 1 \mid x^{(i)}\right)$ to be the **responsibilities** r_k. Doing so gives

$$\frac{\partial \ell}{\partial \mu_1} = \sum_{i=1}^{N} r_k^{(i)} \cdot \frac{x^{(i)} - \mu_1}{\sigma_1^2} \tag{5.5}$$

Setting the derivative equal to 0 and solving for μ_1 gives us

$$\mu_1 = \frac{\sum_{i=1}^{N} r_1^{(i)} x^{(i)}}{\sum_{i=1}^{N} r_1^{(i)}} \tag{5.6}$$

Which looks a bit like a center of mass term. We might call it a day here and say that we have found the closed form solution for the mean μ_1 but here's the issue: the responsibilities r_k *depend on* μ_1. Therefore, we would have sort of a feedback loop kind of error because in calculating the μ_1, we change what the responsibilities are. But that's ok because that is exactly what we are going to do with the EM algorithm! We lay out the EM algorithm here

Repeat until converged:
E-step. Compute the expectations, or responsibilities, of the latent variables:

$$r_k^{(i)} \leftarrow \Pr\left(z^{(i)} = k \mid x^{(i)}\right)$$

M-step. Compute the maximum likelihood parameters given these responsibilities:

$$\boldsymbol{\theta} \leftarrow \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \sum_{k=1}^{K} r_k^{(i)} \left[\log \Pr\left(z^{(i)} = k\right) + \log p\left(\mathbf{x}^{(i)} \mid z^{(i)} = k\right)\right]$$

Typically, only one of the two terms within the bracket will depend on any given parameter, which simplifies the computations. Basically, we repeat 5.5 and 5.6 over and over again until we have convergence. Let us see an example for the Gaussian mixture model. We first take the E step, which gives

$$r_k^{(i)} \leftarrow \mathbb{P}\left(z^{(i)} = k \mid x^{(i)}\right)$$

$$= \mathbb{P}\left(z^{(i)} = k\right) p\left(x^{(i)} \mid z^{(i)} = k\right) \sum_{k'} \mathbb{P}\left(z^{(i)} = k'\right) \mathbb{P}\left(x^{(i)} \mid z^{(i)} = k'\right)$$

$$= \frac{\pi_k \cdot \mathcal{N}\left(x^{(i)}; \mu_k, \sigma_k\right)}{\sum_{k'} \pi_{k'} \cdot \mathcal{N}\left(x^{(i)}; \mu_{k'}, \sigma_{k'}\right)}$$

To compute the M step (calculating the mixing proportions), we have to use Lagrange multipliers. However, there is a trick that we can do and that is to rely on what we know about maximum likelihood estimators. A good estimator for the data (especially for proportions) is the number of times an event of interest occurs over the total number of events N_k/N. We can replace N_k for $\sum r_k$ in this case. As an exercise for the reader, we can check that this is indeed the MLE. The reason I don't want to go into the derivation is because it is not very instructive, and we will not really need that technique again (although

you absolutely should remember Lagrange multipliers). What this gives us is

$$\pi_k = \frac{\sum_{i=1}^{N} r_k^{(i)}}{N}$$

Looking back to the M step of our EM algorithm, we again see that maximizing the first term kills it because it does not depend on our parameters of interest μ and σ. This leaves us with only the second term. We can again make simplifications and say that the μ_k and σ_k only depend on the k cluster (or the k assignment) so the sum over k is only 1 term

$$\sum_{i=1}^{N} r_k^{(i)} \log \mathbb{P}\left(x^{(i)} \mid z^{(i)} = k\right) = \sum_{i=1}^{N} r_k^{(i)} \log \mathcal{N}\left(x^{(i)}; \mu_k, \sigma_k\right)$$

But we've taken maximizations of log likelihoods for Gaussians before. We can give the maximization updates as follows

$$\mu_k \leftarrow \frac{1}{\sum_{i=1}^{N} r_k^{(i)}} \sum_{i=1}^{N} r_k^{(i)} x^{(i)} \tag{5.7}$$

$$\sigma_k \leftarrow \frac{1}{\sum_{i=1}^{N} r_k^{(i)}} \sum_{i=1}^{N} r_k^{(i)} \left(x^{(i)} - \mu_k\right)^2 \tag{5.8}$$

```
# Number of clusters
k = 3
n_samples, n_features = X.shape

# Initialize means, covariances, and mixing coefficients
means = np.random.rand(k, n_features) * 5
covariances = [np.eye(n_features) for _ in range(k)]
mixing_coefficients = np.ones(k) / k

# EM Algorithm
max_iter = 100
for iteration in range(max_iter):
    # E-step
    responsibilities = np.zeros((n_samples, k))
    for i, x in enumerate(X):
        for j in range(k):
            responsibilities[i, j] = mixing_coefficients[j] * multivariate_normal.
    pdf(x, means[j], covariances[j])
        responsibilities[i, :] /= np.sum(responsibilities[i, :])

    # M-step
    N_k = np.sum(responsibilities, axis=0)
```

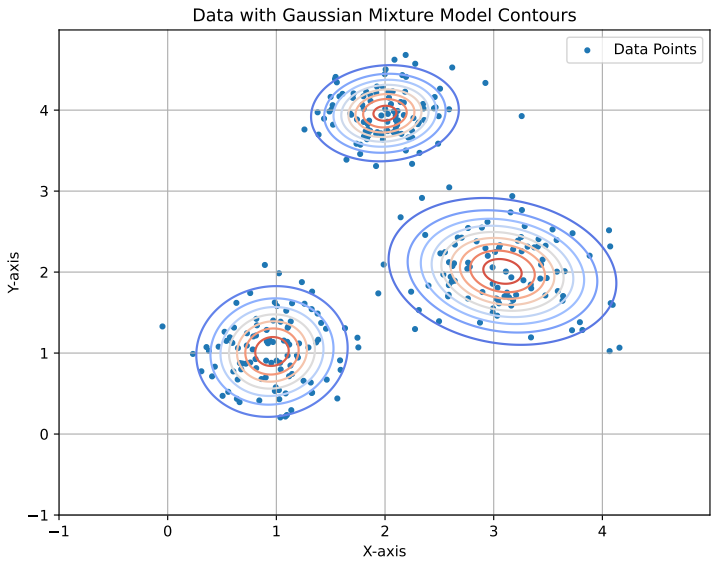

Figure 5.5: Contour plots of the fitted Gaussians using the EM algorithm.

```
    for j in range(k):
        # Update means
        means[j] = np.sum(responsibilities[:, j, None] * X, axis=0) / N_k[j]

        # Update covariances
        diffs = X - means[j]
        covariances[j] = np.sum(responsibilities[:, j, None, None] * np.einsum('ni
    ,nj->nij', diffs, diffs), axis=0) / N_k[j]

        # Update mixing coefficients
        mixing_coefficients[j] = N_k[j] / n_samples

# Print final means and covariances
print("Final means:", means)
print("Final covariances:", covariances)
```

We also plot our results in Figure 5.5.

5.4 Data Reduction and Principle Component Analysis

This next section is going to seem a bit out of nowhere, and maybe it is. In the last 2 sections, we were dealing with data points that lived in 2-dimensional vector spaces. The

	Kale	**Taco Bell**	**Sashimi**	**Pop Tarts**
Alice	10	1	2	7
Bob	7	2	1	10
Carolyn	2	9	7	3
Dave	3	6	10	2

Table 5.1: Preferences of individuals for different foods

issue is that in real life, as we have mentioned many times over, our data points live in insanely high- dimensional vector spaces. Is there a way to reduce the dimension of the problem but also keep the information? It seems like we are asking for too much but that is exactly principle component analysis does! Consider the following problem. Suppose we have a data set that looks something like table 5.1. Obviously your friend Bob is a psychopath who likes kale and pop tarts. If we tried to plot all the categories, we would struggle because the dimension of the categories are 4 dimensional. The goal of PCA is approximate the data in a lower dimensional space so that we can visualize what is going on. An example of how to do this from the above table is

$$\overline{\mathbf{x}} = a_1\mathbf{v}_1 + a_2\mathbf{v}_2$$

where $\overline{\mathbf{x}}$ is the sample average and

$$\mathbf{v}_1 = (3, -3, -3, 3),$$
$$\mathbf{v}_2 = (1, -1, 1, -1),$$

have the interpretation of correlation between rows in our table. For example, we see a that there is a negative correlation between liking kale and liking sashimi, probably because kale is vegetable and sashimi is a meat. The a coefficients are the coefficients belonging to a specific person. For example, (a_1, a_2) is (1,1) for Alice, (1,-1) for Bob, (-1,-1) for Carolyn, and (-1,1) for Dave. As we can see, a good approximation for Alice is $\mathbf{x} + \mathbf{v}_1 + \mathbf{v}_2 = (9.5, 0.5, 3, 7.5)$. How did we reduce the dimension? Well, we took a 4 dimensional space of categories and compressed down to just the 2 dimension a coefficients which contain most of the relevant data. Now let's see how we do this systematically for large data sets.

5.4.1 PCA and SVD

We haven't yet formalized what we mean by principle component. Let us do it here. Our goal is to write the the data as linear combination of orthogonal basis vectors

$$\boldsymbol{x}_n = \overline{\boldsymbol{x}} + \sum_{d=1}^{D} \alpha_d^{(n)} \boldsymbol{u}_d \tag{5.9}$$

The orthogonality condition is a constraint that we impose. The $\overline{\boldsymbol{x}}$ term is the sample mean of the data. Since the $\boldsymbol{u}_d$ vectors are an orthogonal basis, we can easily find the α_d coefficients to be

$$\alpha_d^{(n)} = \left(\boldsymbol{x}_n - \overline{\boldsymbol{x}}\right)^\top \boldsymbol{u}_d$$

You can think of the coefficients α_d as how much of $\boldsymbol{x}_n$ is explained from the $\boldsymbol{u}_d$ direction. Obviously, if we had all D of the $\boldsymbol{u}_d$ vectors, we would be able to exactly describe $\boldsymbol{x}_n$. What we want to know is: if we remove some of the $\boldsymbol{u}_d$ vectors, how much of the original space can be approximate with just a subspace of the original set? That is, how accurate is an approximation like

$$\hat{\boldsymbol{x}}_n \equiv \overline{\boldsymbol{x}} + \sum_{d=1}^{K} \alpha_d^{(n)} \boldsymbol{u}_d$$

with $K < D$. We find the squared distance between the approximation $\hat{\boldsymbol{x}}_n$ and $\boldsymbol{x}_n$ as follows

$$J_n\left(\{\boldsymbol{u}_d\}_{d=1}^K\right) = \left(\left(\overline{\boldsymbol{x}} + \sum_{d=1}^{D} \alpha_d^{(n)} \boldsymbol{u}_d\right) - \left(\overline{\boldsymbol{x}} + \sum_{d=1}^{K} \alpha_d^{(n)} \boldsymbol{u}_d\right)\right)^2$$

Since the sum $\sum_{d=1}^{K} \alpha_d^{(n)} \boldsymbol{u}_d$ is a subset of the sum $\sum_{d=1}^{D} \alpha_d^{(n)} \boldsymbol{u}_d$, their difference is just going to give

$$= \left(\sum_{d=K+1}^{D} \alpha_d^{(n)} \boldsymbol{u}_d\right)^2$$

since the sample averages $\overline{\boldsymbol{x}}$ cancel out. We can exploit the orthogonality of the $\boldsymbol{u}_d$ vectors and find that the above is equal to

$$J_n\left(\{\boldsymbol{u}_d\}_{d=1}^K\right) = \left(\sum_{d=K+1}^{D} \alpha_d^{(n)} \boldsymbol{u}_d\right)^2 = \sum_{d=K+1}^{D} \left(\alpha_d^{(n)}\right)^2$$

But we know what the α_d coefficients are! We can substitute it right back

$$L\left(\{\boldsymbol{u}_d\}_{d=1}^K\right) = \sum_{n=1}^{N} \sum_{d=K+1}^{D} \left(\left(\boldsymbol{x}_n - \overline{\boldsymbol{x}}\right)^\top \boldsymbol{u}_d\right)^2$$

We can simplify it further by expanding the quadratic

$$L\left(\{\boldsymbol{u}_d\}_{d=1}^K\right) = \sum_{n=1}^N \sum_{d=K+1}^D \left(\left(\boldsymbol{x}_n - \overline{\boldsymbol{x}}\right)^\top \boldsymbol{u}_d\right)\left(\left(\boldsymbol{x}_n - \overline{\boldsymbol{x}}\right)^\top \boldsymbol{u}_d\right)$$

And this reduces down to

$$= \sum_{n=1}^N \sum_{d=K+1}^D \boldsymbol{u}_d^\top \left(\left(\boldsymbol{x}_n - \overline{\boldsymbol{x}}\right)\left(\boldsymbol{x}_n - \overline{\boldsymbol{x}}\right)^\top\right)\boldsymbol{u}_d$$

Ah but we know what the term in parenthesis is! It is the **covariance matrix**

$$\boldsymbol{\Sigma} \equiv \frac{1}{N}\sum_{n=1}^N \left(\boldsymbol{x}_n - \overline{\boldsymbol{x}}\right)\left(\boldsymbol{x}_n - \overline{\boldsymbol{x}}\right)^\top \tag{5.10}$$

Our objective now can be simplified down to minimizing

$$\min\left\{\sum_{d=K+1}^D \boldsymbol{u}_d^\top \boldsymbol{\Sigma}\boldsymbol{u}_d\right\} \quad \text{s.t.} \boldsymbol{u}_d^\top \boldsymbol{u}_d = 1$$

We can do the constrained optimization problem using **Lagrange multipliers**

$$N\sum_{d=K+1}^D \boldsymbol{u}_d^\top \boldsymbol{\Sigma}\boldsymbol{u}_d + \lambda_d\left(1 - \boldsymbol{u}_d^\top \boldsymbol{u}_d\right)$$

Now if we differentiate this with respect to the $\boldsymbol{u}_d$ vectors and set it equal to 0, we get that

$$\boldsymbol{\Sigma}\boldsymbol{u}_d = \lambda_d \boldsymbol{u}_d \tag{5.11}$$

which imply that the Lagrange multipliers are the eigenvalues of the covariance matrix. If we substitute this back, we see that the optimization is solved if

$$\min \sum_{d=K+1}^D \boldsymbol{u}_d^\top \boldsymbol{\Sigma}\boldsymbol{u}_d = \min \sum_{d=K+1}^D \boldsymbol{u}_d^\top \left(\lambda_d \boldsymbol{u}_d\right) = \min \sum_{d=K+1}^D \lambda_d$$

And in order to minimize the sum of the eigenvalues, we have to take the smallest eigenvalues from $K+1$ to the last. That is, most of the information that is contained in the data comes from the largest eigenvalues of the covariance matrix. All of this is reliant on knowing the covariance matrix. However, as it turns out, due to singular value decomposition, we actually don't need to compute the covariance matrix, we can extract the same information using **singular value decomposition**. Again, either method gives us the same result

(because the singular values are just the square root of the eigenvalues. Let us now see how we can reconstruct the data by projecting it onto a smaller subpace.

```
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Original data from the table
data = np.array([
    [10, 1, 2, 7],
    [7, 2, 1, 10],
    [2, 9, 7, 3],
    [3, 6, 10, 2]
])

# Standardize the data
scaler = StandardScaler()
standardized_data = scaler.fit_transform(data)

# Perform PCA
pca = PCA(n_components=2)
pca.fit(standardized_data)

# Project Alice's data (first row) onto the first two principal components
alice_standardized = standardized_data[0].reshape(1, -1)
alice_projected = pca.transform(alice_standardized)

# Reconstruct Alice's data from the projection
alice_reconstructed = pca.inverse_transform(alice_projected)

# Reverse standardization to get the approximate original values
alice_approximation = scaler.inverse_transform(alice_reconstructed)

print("Approximated Data for Alice:")
print(alice_approximation)
```

The output is

```
Approximated Data for Alice:
[[9.54625869 0.49353959 2.53930347 7.42342464]]
```

Which to a good degree is the approximation that we made at the beginning of this section.

6

Support Vector Machines

Support vector machines have massive hype around them as being one of the best supervised learning method. Maybe this is true, for if it wasn't, we wouldn't be here talking about it. the problem that we are going to try to solve is one that we have encountered already, which is binary classification. Suppose we have data sets that are labeled as positive or negative (circles and squares, up and down, etc etc). The fundamental assumption that we are going to make is that the data is linearly separable. In that case, with logistic regression, we see that there exists some plane hyperplane that will split the data. Support vector machines takes a slightly different approach. Instead of maximizing the likelihood of the data, what we are going to do is we are going to consider the separation between the closest datum of each respective class and find the hyperplane that best separates the data. Figure 6.1 gives us a visualization of the problem. Suppose that we have blue point that is closest to the closest red point to the decision boundary. The hyperplane that separates the data will be at the midpoint of the closest 2 data points. The question that we are going to try to answer is: what is the equation for the margin that maximizes the width of the margin?

6.1 Finding The Margin

Let us begin by supposing that we have a data point that we will label positive (+1). If we had a decision boundary, then we can define the vector w as the vector that is perpendicular to the decision boundary. With respect to Figure 6.1, we can say that the vector $\boldsymbol{w}$ is parallel to the "margin" arrow. We can define the orthogonal distance from the datum x_i to

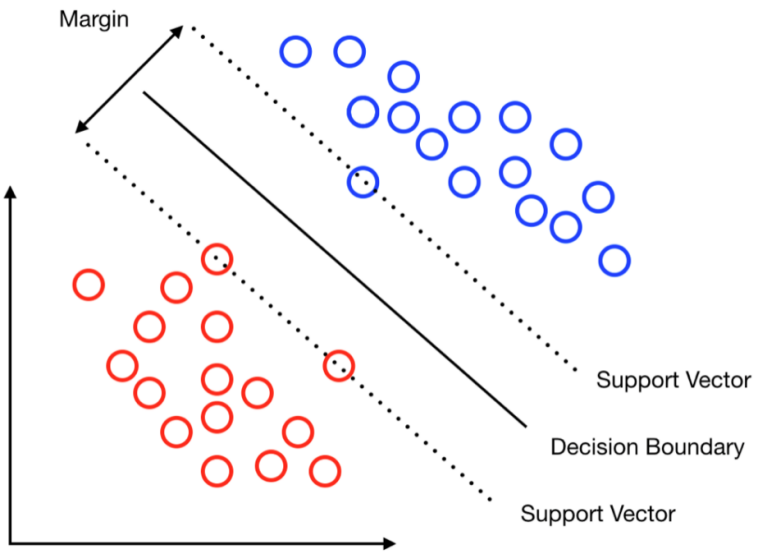

Figure 6.1: A visualization of the goal of support vector machines. We want to find the hyperplane that maximizes the width of the margin

the decision boundary

$$\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b \geq 0 \tag{6.1}$$

Why is this true? Taking the dot product of $\boldsymbol{w}$ and $\boldsymbol{x}$ will project $\boldsymbol{x}$ onto $\boldsymbol{w}$. The constant b is the bias term; you can think of it as the intercept of the decision boundary. The reason for the inequality $\geq$ is because at 0 we are on the decision boundary (think back to logistic regression). Thus, if $\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b \geq 0$, then we label $\boldsymbol{x}$ as a positive label. The thing is: we don't know what $\boldsymbol{w}$ is and we don't know what b is. What do we do from here? Without loss of generality, we can say that the hardest to learn case (the blue point closest to the decision boundary) happens when $\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b = 1$ and $\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b = -1$ for red. Obviously, if 6.1 is significantly large, then we know that we are deep into positive category and we are confident that we are correct. Likewise, if 6.1 is significantly negative, then we are confident that it is negative. We are going to introduce a label y_i that takes on values of $\{-1, 1\}$, with positivity when we classify as positive. Then 6.1 compacts to

$$y_i \left(w \cdot x_i + b\right) \geq 1 \tag{6.2}$$

for a single sample x_i with equality happening when we are in the "gutter" (at the **support vector** as seen in Figure 6.1).

$$y_i \left(w \cdot x_i + b \right) - 1 = 0 \tag{6.3}$$

Let's label the x_+ the blue point on the support vector and x_- the red point on the support vector. Geometrically, we can write the width of the margin as

$$(\boldsymbol{x}_+ - \boldsymbol{x}_-) \cdot \frac{\boldsymbol{w}}{||\boldsymbol{w}||} \tag{6.4}$$

Why is this true? $(\boldsymbol{x}_+ - \boldsymbol{x}_-)$ is a vector in between the 2 support vectors. If we take a dot product with the unit normal $\boldsymbol{w}/||\boldsymbol{w}||$, then we would get the length perpendicular to the decision boundary between the 2 support vectors. With equation 6.3, we can write the $\boldsymbol{x}_+$ and $\boldsymbol{x}_-$ in terms of b and $\boldsymbol{w}$ as

$$x_\pm = \frac{-b \pm 1}{||\boldsymbol{w}||}$$

subbing this back into 6.4 gives us the width as

$$\text{width} = \frac{2}{||\boldsymbol{w}||^2}$$

For the sake of convenience, maximizing the width is the same as minimizing $||\boldsymbol{w}||^2$ (with the factor of 2 gone because derivatives don't care about constants. The next couple of steps that we are going to take are also out of convenience. Since we are minimizing the $||\boldsymbol{w}||$, that's the same as minimizing $\frac{1}{2}\boldsymbol{w}^2$ ($|x|$ and $\frac{1}{2}x^2$ share the minimum point and minimum value). In true physics fashion, we are going to define the **Lagrangian**

$$\mathscr{L} = \frac{1}{2}\boldsymbol{w}^2 \tag{6.5}$$

and we are going to try to **minimize the action** with the constraint 6.3. We can modify the Lagrangian with a **Lagrange multiplier**

$$\mathscr{L} = \frac{1}{2}\boldsymbol{w}^2 - \sum_i^N \lambda_i \left[y_i \left(w \cdot x_i + b \right) - 1 \right] \tag{6.6}$$

Now we can take the derivative and the constraint will take care of itself in the Lagrange multiplier term

$$\frac{\partial \mathscr{L}}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_i^N \lambda_i y_i x_i$$

setting this equal to 0 gives us

$$\boldsymbol{w} = \sum_{i}^{N} \lambda_i y_i \boldsymbol{x}_i \tag{6.7}$$

Similarly, the derivative of the Lagrangian with respect to the bias b and setting it equal to 0 gives us

$$0 = \sum_{i}^{N} \lambda_i y_i \tag{6.8}$$

Now that we have the location of the minimum, we can substitute this back into the Lagrangian

$$\mathscr{L} = \frac{1}{2}\left(\sum_{i}^{N} \lambda_i y_i x_i\right)\left(\sum_{j}^{N} \lambda_j y_j x_j\right) - \left(\sum_{i}^{N} \lambda_i y_i x_i\right)\left(\sum_{j}^{N} \lambda_j y_j x_j\right) - b\sum_{i}^{N} \lambda_i y_i + \sum_{i}^{N} \lambda_i \tag{6.9}$$

This does indeed look ugly, here are some things we can do. Firstly, the last term $\sum_i^N \lambda_i y_i$ is 0 due to 6.8. Secondly, the terms in parenthesis are the same term so we can just simplify it down to 1 term with a double sum

$$\mathscr{L} = \sum_{i}^{N} \lambda_i - \frac{1}{2}\sum_{i}^{N}\sum_{j}^{N} \lambda_i \lambda_j y_i y_i x_i x_j \tag{6.10}$$

This is what is known as a **quadratic program**. We need to find the optimal values of λ that solves the above problem. Once we have them, we can plug them back into 6.7 to find the $||\boldsymbol{w}||$ and b values that will solve our margin problem.

6.2 Kernels

The progress that we have made so far only really works with data that are linearly separable. What do we do when our data is not linearly separable, as in the case of Figure 6.2 (left)? We use the tricks that we have learned in chapter 3: we do a basis function expansion. instead of linear functions of x_i, we will have $\boldsymbol{\Phi}(x)$. Then our quadratic program in 6.10 becomes

$$\boldsymbol{\lambda}^\star = \arg\max_{\boldsymbol{\lambda}} \left\{ \sum_{i=1}^{N} \lambda_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} y_i y_j \lambda_i \lambda_j \Phi\left(\boldsymbol{x}_i\right)^\top \Phi\left(\boldsymbol{x}_j\right) \right\}$$

$$\text{s.t.} \quad \sum_{i=1}^{N} y_i \lambda_i = 0 \quad \text{and} \quad \lambda_i \geq 0 \;\; \forall n \in 1, \ldots, N$$

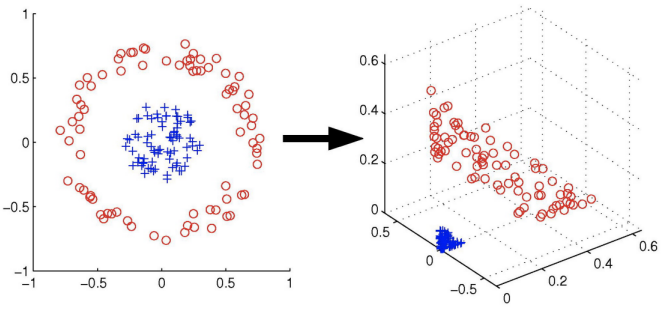

Figure 6.2: Linearly inseparable data. We have to expand it into 3 dimensions and then separate it using a hyperplane.

Here is one thing to notice: in the above, the quadratic function ultimately only depends on the inner product of $\Phi\left(\boldsymbol{x}_i\right)^\top \Phi\left(\boldsymbol{x}_{i'}\right)$. We call functions that depend on the dot product of $\boldsymbol{\Phi}(x)$ to be called **kernels.** There are multiple choices of kernels but the most common ones are the **polynomial kernels** and **radial basis functions kernels.** More precisely, a kernel function $K : \mathfrak{X} \times \mathfrak{X} \to \mathbb{R}$ is a function such that there exists some vector function $\Phi : \mathfrak{X} \to \mathbb{R}^J$ such that $K\left(\boldsymbol{x}, \boldsymbol{x}'\right) = \Phi(\boldsymbol{x})^\top \Phi\left(\boldsymbol{x}'\right)$. We can rewrite the quadratic program as

$$\boldsymbol{\lambda}^\star = \arg\max_{\boldsymbol{\lambda}} \left\{ \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \lambda_i \lambda_j K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) \right\} \tag{6.11}$$

$$\text{s.t.} \quad \sum_{i=1}^{N} y_i \lambda_i = 0 \quad \text{and} \quad \lambda_i \geq 0 \;\; \forall n \in 1, \ldots, N$$

6.3 An Illustrative Example

Let us look back to the clustering problem in chapter 2 (Figure 6.3). Let before writing our code, we should rewrite equation 6.10 in a way that is more easily understandable by a computer

$$\boldsymbol{w}^\star, b^\star = \arg\min_{\boldsymbol{w}, b} \left\{ \frac{1}{2} \|\boldsymbol{w}\|^2 \right\} \quad \text{s.t.} \quad y_n \left(\boldsymbol{w}^\top \boldsymbol{x}_n + b \right) \geq 1 \quad \forall n \in 1, \ldots, N \tag{6.12}$$

Notice that this was exactly what we were trying to do in 6.5 with the constraint of 6.3. Now, our code

```
import numpy as np
from cvxopt import matrix, solvers
```

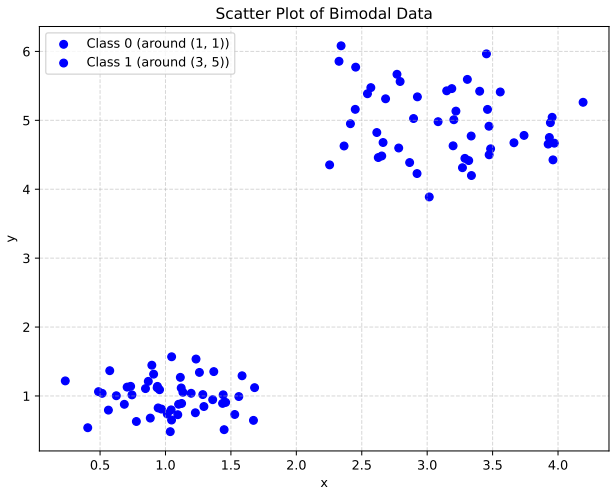

Figure 6.3: Clusters that we are trying to separate.

```

# Data preparation
np.random.seed(0)

# First mode centered around (1, 1)
x1 = np.random.normal(loc=1, scale=0.3, size=50)
y1 = np.random.normal(loc=1, scale=0.3, size=50)

# Second mode centered around (3, 5)
x2 = np.random.normal(loc=3, scale=0.5, size=50)
y2 = np.random.normal(loc=5, scale=0.5, size=50)

# Combine the data
x = np.concatenate([x1, x2])
y = np.concatenate([y1, y2])

# Create labels: 0 for the first mode, 1 for the second
labels = np.array([0] * 50 + [1] * 50)
y_labels = 2 * labels - 1  # Transform labels to -1 and 1

# Create the kernel matrix
N = len(x)
K = np.zeros((N, N))
for i in range(N):
    for j in range(N):
        K[i, j] = y_labels[i] * y_labels[j] * (x[i] * x[j] + y[i] * y[j])
```

```
29
30 # Convert to cvxopt format
31 P = matrix(K)
32 q = matrix(-np.ones(N))
33 G = matrix(-np.eye(N))
34 h = matrix(np.zeros(N))
35 A = matrix(y_labels, (1, N), 'd')
36 b = matrix(0.0)
37
38 # Solve the quadratic program
39 solution = solvers.qp(P, q, G, h, A, b)
40 lambdas = np.ravel(solution['x'])
41
42 # Support vectors have non zero lagrange multipliers
43 sv = lambdas > 1e-5
44 index = np.arange(len(lambdas))[sv]
45 lambdas_sv = lambdas[sv]
46 support_vectors = np.column_stack((x, y))[sv]
47 support_vector_labels = y_labels[sv]
48
49 # Calculate weights
50 w = np.zeros(2)
51 for i in range(len(lambdas_sv)):
52     w += lambdas_sv[i] * support_vector_labels[i] * support_vectors[i]
53
54 # Calculate bias
55 b = 0
56 for i in range(len(lambdas_sv)):
57     b += support_vector_labels[i] - np.dot(w, support_vectors[i])
58 b /= len(lambdas_sv)
59
60 print("Lagrange multipliers: ", lambdas)
61 print("Weights: ", w)
62 print("Bias: ", b)
```

the output is given as

```
     pcost       dcost       gap    pres   dres
 0: -2.9099e+00 -4.4764e+00  2e+02  1e+01  1e+00
 1: -8.1310e-01 -7.0368e-01  1e+01  9e-01  9e-02
 2:  1.1775e-02 -5.6253e-01  6e-01  1e-16  5e-15
 3: -1.4401e-01 -2.4341e-01  1e-01  1e-16  1e-15
 4: -1.9737e-01 -2.5320e-01  6e-02  3e-17  1e-15
 5: -2.2775e-01 -2.3253e-01  5e-03  6e-17  1e-15
 6: -2.3176e-01 -2.3189e-01  1e-04  2e-16  1e-15
 7: -2.3187e-01 -2.3187e-01  1e-06  6e-17  9e-16
 8: -2.3187e-01 -2.3187e-01  1e-08  4e-17  1e-15
Optimal solution found.
```

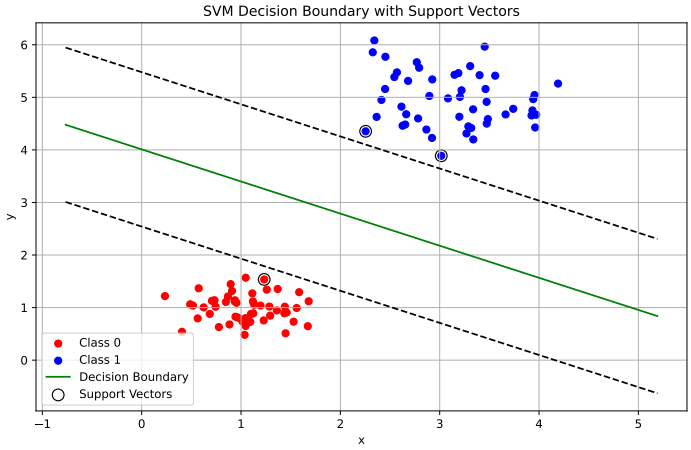

Figure 6.4: Plotting the decision and the support vectors once we've ran the code to calculate the weight and biases.

```
Lagrange multipliers:  [2.95596045e-10 3.38849055e-10 2.75339292e-10 3.00974903e
    -10
 5.43609598e-10 2.31307912e-10 3.59076645e-10 2.74259398e-10
 2.03259800e-10 2.49190366e-10 2.11238822e-10 3.48939330e-10
 2.31409310e-10 1.66822382e-10 3.13936887e-10 2.40878151e-10
 2.09590155e-10 2.86392033e-10 2.04274723e-10 2.12297927e-10
 1.83211812e-10 3.30603046e-10 8.51819397e-10 1.62696996e-10
 2.14660883e-09 1.62952156e-10 1.96240445e-10 2.03062371e-10
 3.70428528e-10 4.55252942e-10 1.86229637e-10 4.08749125e-10
 2.39005083e-10 1.41302945e-10 2.70576818e-10 6.37403891e-09
 3.87042410e-09 3.48378566e-10 1.75089595e-10 3.28408323e-10
 1.82156367e-10 2.57435786e-10 1.87902558e-10 2.20258190e-08
 2.53517597e-10 2.90765625e-10 1.94114857e-10 2.31871017e-01
 1.87287338e-10 2.79504489e-10 1.28374358e-10 1.31121427e-10
 6.95267770e-10 1.84650424e-10 2.59385263e-10 1.39987968e-10
 1.30722684e-10 7.17492427e-10 1.66806700e-10 1.40822341e-10
 1.36182994e-10 1.29436473e-10 1.37076589e-10 1.26949831e-10
 5.60143905e-10 1.27678072e-10 2.08495677e-10 8.56640222e-10
 1.27895366e-10 1.30630947e-10 1.28669941e-10 1.80514094e-10
 1.29279311e-10 1.54553870e-10 6.23836193e-10 1.34506073e-10
 3.58395701e-10 1.42816295e-10 1.64971650e-10 1.45440896e-10
 2.67987668e-10 3.64114602e-10 3.48286646e-10 1.55299289e-01
 1.43264479e-10 4.34266767e-10 2.68644158e-10 1.61986351e-10
 2.14585805e-10 1.31014507e-10 7.65717644e-02 1.36539473e-10
 1.54856838e-10 2.76683041e-10 1.25014634e-10 1.41596337e-10
 1.43256157e-10 1.27037032e-10 1.31204970e-10 1.29792807e-10]
Weights:  [0.35503642 0.58111213]
```

```
38 Bias:  -2.3302971166824618
```

We notice that with the exception of the first few, the Lagrange multipliers are basically 0. Once we have the weights and the bias, we can plot the decision boundary and the support vectors in Figure 6.4.

7

Bayesian Methods

I don't think there's any doubt in my mind or in anyone's mind that Bayesian statistics is one of the most important development in statistics. It is a completely different philosophy on statistics than its frequentist counterpart. In this chapter, we will look at the development of Bayesian statistics and how it is applied in learning methods like linear discriminant analysis.

7.1 Bayesian Statistics

Let's begin by comparing and contrasting Bayesian statistics with the classical frequentist interpretation of probability. In classical probability and statistics, the interpretation of a probability is the limiting frequency of events. For example, if I flip a coin 1000 times and 500 of those landed heads, then I might suspect that the probability of getting heads is 1/2. Because of this, unknown parameters like the p for the independent Bernoulli trials are fixed; there are no probabilities associated with them. There is also another property concerning confidence intervals but that's not really too important for machine learning. How is this contrasted with the Bayesian philosophy? The fundamental difference in Bayesian statistics is that the interpretation of probability is our own subjective belief that an event will occur. For example, I believe that you will drink coffee tomorrow with probability .5. There is no limiting case in this example because I cannot rerun tomorrow multiple times to determine how often you will drink coffee. Often what we will do is we start with some sort of guess called a **prior** and based on the observed data, we will update our probabilities. This update is called the **posterior**. This implies that we can make probability statements about

parameters like p even though they are fixed.

7.1.1 The Posterior Distribution

We said that we begin with the prior distribution on the parameter $\mathbb{P}(\Theta)$. What we want to know is the posterior $\mathbb{P}(\Theta|x_1, ..., x_n)$. That is; what is the probability of the parameter given our observed data. Well, we can use Bayes' theorem to find the posterior

$$\mathbb{P}(\Theta|x_1, ..., x_n) = \frac{\mathbb{P}(x_1, ..., x_n|\Theta)\mathbb{P}(\Theta)}{\mathbb{P}(x_1, ..., x_n)} \tag{7.1}$$

Then we can use the law of total probability to turn this into

$$\mathbb{P}(\Theta|x_1, ..., x_n) = \frac{\mathbb{P}(x_1, ..., x_n \mid \Theta = \theta)\mathbb{P}(\Theta = \theta)}{\sum_\theta \mathbb{P}(x_1, ..., x_n \mid \Theta = \theta)\mathbb{P}(\Theta = \theta)} \tag{7.2}$$

In the case of continuous random variables, this becomes

$$f(\theta \mid x) = \frac{f(x \mid \theta)f(\theta)}{\int f(x \mid \theta)f(\theta)d\theta} \tag{7.3}$$

In most applications of equation 7.3, we won't care about the normalization constant in the denominator. It often is more concise to just say that the posterior distribution is proportional to the likelihood times the prior, where $f(x \mid \theta)$ is the **likelihood function** of the data. Let us see an example. Consider a set of i.i.d. random variables $X_1, X_2, ..., X_n \sim \text{Bernoulli}(p)$ and we want to estimate what the p is. Our subjective prior for what p is will just be a uniform distribution on [0,1]. This is just an indicator of our ignorance. Plugging this into 7.3, we get that

$$f(\theta \mid x) \propto p^s(1-p)^{n-s} = p^{s+1-1}(1-p)^{n-s+1-1}$$

The reason that we included the +1 -1 is because we know that the beta distribution takes the form

$$f(x) == \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}p^{\alpha-1}(1-p)^{\beta-1} \tag{7.4}$$

thus if we take $s+1$ to be α and $n-s+1$ to be β, then we get the normalization constant for free and we see that our posterior distribution is

$$f\left(p \mid x^n\right) = \frac{\Gamma(n+2)}{\Gamma(s+1)\Gamma(n-s+1)}p^{(s+1)-1}(1-p)^{(n-s+1)-1} \tag{7.5}$$

We can find the average value of p by taking the average of the posterior distribution and in doing so we find that

$$\bar{p} = \frac{s+1}{n+2}$$

7.2 Linear Discriminant Analysis

Having had our discussion on Bayesian statistics, we turn our heads to an example of a Bayesian classifier **linear discriminant analysis.** The problem that we are going to be attempting to solve will be the same problem as linear classification. However, whereas in linear classification by logistic regression, we asked "what is the probability of getting the datum x_i given that this is in class k" ($\mathbb{P}(x_i|\Theta) =?$) Now, we are going to ask "given this datum, what is the probability of being in class k?" ($\mathbb{P}(\Theta|x_i) =?$) It seems like a different question to ask but these are actually very similar to each other, and are related by Bayes' theorem. We are going to change our notation just a bit to blend in with the machine learning crowd. Suppose we have a prior on what the class should be denoted by π_k. Then using Bayes' theorem, we get

$$\mathbb{P}(G = k \mid X = x) = \frac{f_k(x)\pi_k}{\sum_{\ell=1}^{K} f_\ell(x)\pi_\ell}$$

This is the fundamental basis for linear discriminant analysis. For most practical applications, we will make 2 more assumptions about the data: the probability of x given a class k $f_k(x)$ is a multivariate Gaussian AND each Gaussian distribution have the same covariance. That is,

$$f_k(x) = \frac{1}{(2\pi)^{p/2}\left|\mathbf{\Sigma}_k\right|^{1/2}} e^{-\frac{1}{2}(x-\mu_k)^T \mathbf{\Sigma}_k^{-1}(x-\mu_k)}$$

and we will study the case when $\mathbf{\Sigma}_k$ is the same for all classes. We have both the prior and the likelihood, we can find the posterior. However, what would be more instructive is if we found the decision boundary. When does this happen? This happens when $\mathbb{P}(G = k \mid X = x) = \mathbb{P}(G = l \mid X = x)$. Here is what we can do. Since we are interested in the location, we find the log of their ratios (will just be 0 because $\log(1) = 0$)

$$\log \frac{\mathbb{P}(G = k \mid X = x)}{\mathbb{P}(G = \ell \mid X = x)} = \log \frac{f_k(x)}{f_\ell(x)} + \log \frac{\pi_k}{\pi_\ell}$$

Subbing in for the distribution of x and simplifying using the fact that the covariant matrices are all the same gives us

$$0 = \log \frac{\pi_k}{\pi_\ell} - \frac{1}{2}\left(\mu_k + \mu_\ell\right)^T \mathbf{\Sigma}^{-1}\left(\mu_k - \mu_\ell\right) + x^T \mathbf{\Sigma}^{-1}\left(\mu_k - \mu_\ell\right)$$

The first 2 terms are just constants; the only thing that depends on the data is the last term. Thus, we can bring all the constants to one side and take

$$\frac{1}{2}\left(\mu_k + \mu_\ell\right)^T \mathbf{\Sigma}^{-1}\left(\mu_k - \mu_\ell\right) - \log \frac{\pi_k}{\pi_\ell} = x^T \mathbf{\Sigma}^{-1}\left(\mu_k - \mu_\ell\right)$$

Again, since the left hand side is just a number, we can compress down to

$$x^T \mathbf{\Sigma}^{-1} (\mu_k - \mu_\ell) = c \tag{7.6}$$

How does this give us a decision boundary? Well, let's break down equation 7.6. x^T is our data vector. We are taking a dot product with the centered $\mathbf{\Sigma}^{-1} (\mu_k - \mu_\ell)$. This should ring some bells from when we did support vector machines; this vector is the vector that is **perpendicular to the decision boundary!**. So the dot product is projecting our vector to the vector perpendicular to the decision boundary. If the the projection is greater than c, we are in class k (look back to log ratio) and if it is less than c, then we are in class l. It may seem like we are doing binary classification, but the labels l and k work generically.

7.2.1 Estimating The Parameters

This is great if we knew exactly what the true mean and true population covariance. However, if we did know, then we wouldn't need to do machine learning. So what exactly do we need to learn? We need to learn μ and Σ from the data. But wait! The data is Gaussian like we assumed; we know how to get estimators from Gaussian distributions. We know what the maximum likelihood estimators are for μ and Σ as well as the prior (the prior is just a proportion, so we can use the MLE for population proportions):

$$\hat{\pi}_k = N_k/N \quad \text{where } N_k \text{ is the number of class k observations}$$

$$\hat{\mu}_k = \sum_{g_i=k} x_i/N_k$$

$$\hat{\mathbf{\Sigma}} = \sum_{k=1}^{K} \sum_{g_i=k} (x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T /(N-K) \tag{7.7}$$

Let us do an example. Let us examine Figure 7.1. Here is the given code

```
# Set random seed for reproducibility
np.random.seed(42)

# Cluster centers
centers = np.array([[1, 1], [3, 2], [2, 4]])

# Number of points in each cluster
n_points = 100

# Generate random points around each center
cluster_1 = centers[0] + 0.4 * np.random.randn(n_points, 2)
cluster_2 = centers[1] + 0.5 * np.random.randn(n_points, 2)
cluster_3 = centers[2] + 0.3 * np.random.randn(n_points, 2)
```

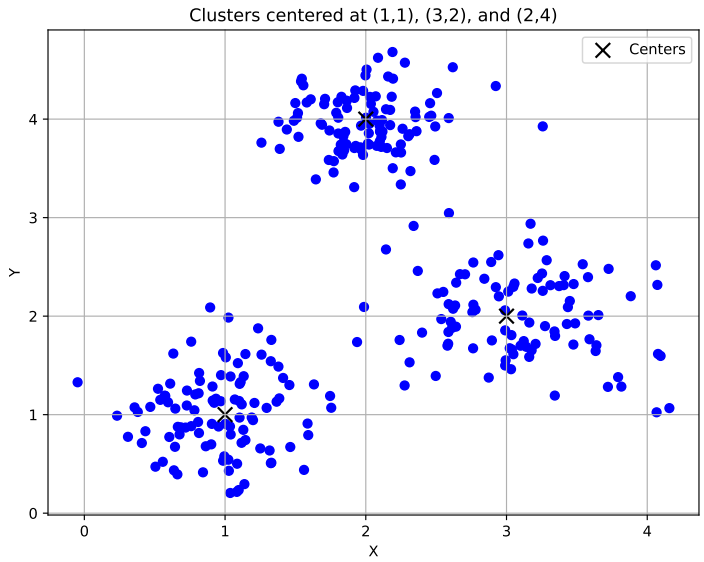

Figure 7.1: 3 clusters to perform LDA on

```
14
15 # Combine the clusters to form the dataset
16 X = np.vstack((cluster_1, cluster_2, cluster_3))
17
18 # Labels for each cluster
19 labels = np.array([0] * n_points + [1] * n_points + [2] * n_points)
20
21 # Number of clusters
22 k = 3
23
24 # Calculate class priors
25 pi_k_values = [n_points / len(X)] * k
26
27 # Calculate class means
28 mu_k = [np.mean(cluster_1, axis=0),
29         np.mean(cluster_2, axis=0),
30         np.mean(cluster_3, axis=0)]
31
32 # Calculate shared covariance matrix
33 cov_matrices = [np.cov(cluster_1, rowvar=False),
34                 np.cov(cluster_2, rowvar=False),
35                 np.cov(cluster_3, rowvar=False)]
36
37 # Shared covariance matrix
```

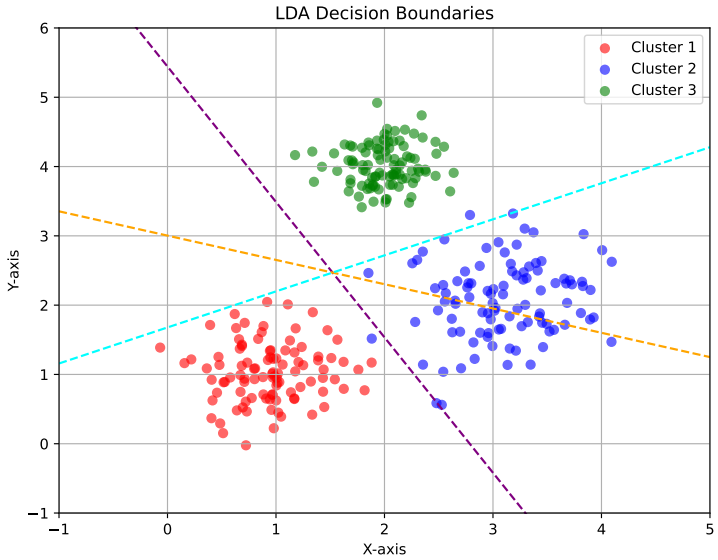

Figure 7.2: Result of the output code

```
Sigma = sum(pi_k_values[i] * cov_matrices[i] for i in range(k))

pi_k_values, mu_k, Sigma
```

The output is

```
([0.3333333333333333, 0.3333333333333333, 0.3333333333333333],
 [array([0.9537743 , 1.01360893]),
  array([3.06412436, 2.02174382]),
  array([1.98648878, 3.9621182 ])],
 array([[ 0.16068571, -0.00272904],
        [-0.00272904,  0.15400079]]))
```

The results are given in figure 7.2.

7.3 Quadratic Discriminant Analysis

The difference between quadratic discriminant analysis and linear discriminant analysis is that we relax the constraint that the covariances of all the distributions are the same. When we do this, the decision boundary is no longer linear but quadratic in the data. When we do analysis of taking the log ratios, the the covariances don't cancel out entirely. To see

this, let us define the discriminant function, which is just the log of the posterior

$$\delta_k(x) = -\frac{1}{2}\log|\mathbf{\Sigma}_k| - \frac{1}{2}(x-\mu_k)^T \mathbf{\Sigma}_k^{-1}(x-\mu_k) + \log \pi_k$$

Clearly, if the covariances are not the same, when we set the decision boundary for 2 classes equal to each other, the determinants do not cancel out and the the second term does not cancel out. Instead, we would get a quadratic decision boundary since the second term in the above is a **quadratic form.** We don't dwell too much on QDA because it is rarely done in practice; but we can see how the results that we have made in section 7.2 generalize.

8

Reinforcement Learning

Reinforcement learning is new paradigm of machine learning that is different than supervised and unsupervised learning. The inspiration comes from psychology, in we train animals to behave a certain why by rewarding good actions and punishing bad actions. The field of reinforcement learning is massive and it probably deserves an entire set of notes for itself. Similar to deep learning, what we are going to do is we are going to give the basics of reinforcement learning. We will not really be implementing any code simply because this is not something that we can write in a few lines of code.

8.1 Markov Decision Processes

Before we talk about any kind of algorithm, we will discuss a little bit about terminology and the set up of the system. Suppose we have an **agent**; a player, a driver, a robot that exists in an **environment**. Sometimes it's very difficult to explain to the agent exactly how to perform its task because the details of the mechanisms are either too complicated or we ourselves do not know how to solve such a problem. For example, imagine trying to explain to a robot how to walk. There are a lot of biophysical mechanisms that go into walking, like how much pressure to use, what angle should we rotate our joints, how wide each step should be, etc etc. There are a lot things that we do that we take for granted. Instead of trying to explain, what we do is we let the agent attempt to solve the problem itself. If it does something good, we reward it. If it does something bad, we punish it by maybe lowering the probability that it took the bad action. The Markov Decision Process (MDP) is based on stochastic processes and gives us a guide on how to perform such tasks. The

MDP framework consists of the following elements $(\mathcal{S}, \mathcal{A}, R, P)$:

- A set $\mathcal{S} = \{1, \ldots, N\}$ of possible states

- A set $\mathcal{A} = \{1, \ldots, M\}$ of possible actions

- A reward model $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, assigning reward to state tuples of state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$

-A transition model $P(s' \mid s, a)$, where $P(s' \mid s, a) \geq 0$ and $\sum_{s' \in \mathcal{S}} P(s' \mid s, a) = 1$ for all s, a, and defining the probability of reaching state s' in the next period given action a in state s. This is a probability mass function over the next state, given the current state and action

We suppose that the agent starts in some state s_0. There are a set of actions that it can take in the set $\mathcal{A}$. Each action that it takes has some reward to it. The reason that this is called a Markov process is because in stochastic processes, a Markov process is a process where future events are only affected by the current state and not past states. That is, what happens tomorrow only depends on today and not yesterday. One other ingredient that we are going to need is the **policy**, which is just a function

$$\pi_t : \mathcal{S} \to \mathcal{A}$$

There are a couple of methods that we can use to maximize the **utility** of the objective. The first is **finite horizon**; which just means that the agent cannot see past some time point. The reason why this might be useful is consider the following example: would you rather I give you 100 dollars today or 100 dollars in 10 years? If we are to be greedy, then we might say that the 100 dollars right now is of more utility to us. Thus we can write

$$\text{Utility} = \sum_{t=0}^{T-1} R(s_t, a_t)$$

But maybe we are not super greedy. In that case, the limit in the above exponential goes to infinity so that we can maximize as much utility as possible. This is called **infinite horizon**.

$$\text{Utility} = \sum_{t=0}^{\infty} R(s_t, a_t)$$

There is a middle ground to this. Suppose we want to see into the infinite future, but the longer we look out, the less important it is to us. For example, we would love to make 100 dollars in 10 years, but it matters more to us that we make 100 dollars now. This is called

the **total discounted reward**

$$\text{Utility} = \sum_{t=0}^{\infty} \gamma^t R\left(s_t, a_t\right)$$

where γ is some dampening factor. This will ensure that the utility converges to some finite number.

8.2 Q Function and Policy Iteration

One basic algorithm for reinforcement learning is called **value learning**. We need to define the **Q-function** which is just

$$Q\left(s_t, a_t\right) = \mathbb{E}\left[R_t \mid s_t, a_t\right] \tag{8.1}$$

Which is just the expected total future reward given the state that we are in and the action that we are to take. The objective of Q-learning is to choose the policy that maximizes our expected reward

$$\pi^*(s) = \underset{a}{\text{argmax}} Q(s,a)$$

The question is, how do we know which policy to take to maximize the Q function? Well, what we can do is create a cost function of sorts

$$\mathscr{L} = \mathbb{E}[\|(\overbrace{\left(r + \gamma \max_a\left(s', a'\right)\right.}^{\text{target}} - \overbrace{Q(s,a)}^{\text{predicted}} \|^2]$$

Our goal is to find the set of policies that minimizes the above cost. A simple agorithm to do this is

```
Initialize V(s) arbitrarily for all states s
Repeat until convergence:
    Δ = 0
    For each state s:
        v = V(s)
        V(s) = max_a Σ [P(s'|s,a) * (R(s,a,s') +   * V(s'))]
        Δ = maxΔ(, |v - V(s)|)
    If Δ <   (a small threshold), break

Extract the policy pi from V:
    For each state s:
         (s) = argmax_a Σ [P(s'|s,a) * (R(s,a,s') +   * V(s'))]
```

In order to do this, we would need to build a deep neural network to run all our computation.

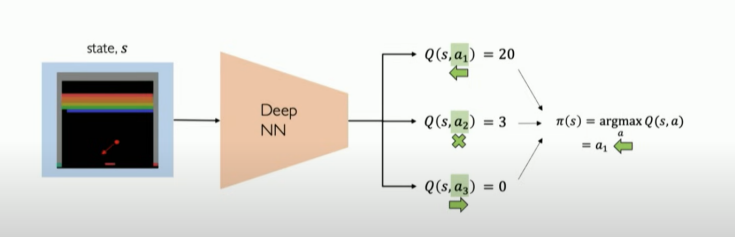

Figure 8.1: General shematic of how reinformcement works with deep neural netowrks

A simple schematic is shown in Figure 8.1. I'm being extremely hand-wavy and that's because I don't want to dwell too much on it. The big point that I am trying to make is that reinforcement learning takes on a different approach to machine learning that is different than what we have done so far. It still does come down to minimizing some cost function, but the objective is slightly different. Previously, we have data to that we want to extract information from. Now, we are throwing the computer into some environment with little training and hoping that it learns with reward and punishments. The types of problems that we hope to solve are very different than supervised and unsupervised learning as you can see.