Introduction

What this Book Covers

This book covers the building blocks of the most common methods in machine learning. This set of methods is like a toolbox for machine learning engineers. Those entering the field of machine learning should feel comfortable with this toolbox so they have the right tool for a variety of tasks. Each chapter in this book corresponds to a single machine learning method or group of methods. In other words, each chapter focuses on a single tool within the ML toolbox.

In my experience, the best way to become comfortable with these methods is to see them derived from scratch, both in theory and in code. The purpose of this book is to provide those derivations. Each chapter is broken into three sections. The *concept* sections introduce the methods conceptually and derive their results mathematically. The *construction* sections show how to construct the methods from scratch using Python. The *implementation* sections demonstrate how to apply the methods using packages in Python like `scikit-learn`, `statsmodels`, and `tensorflow`.

Why this Book

There are many great books on machine learning written by more knowledgeable authors and covering a broader range of topics. In particular, I would suggest An Introduction to Statistical Learning, Elements of Statistical Learning, and Pattern Recognition and Machine Learning, all of which are available online for free.

While those books provide a conceptual overview of machine learning and the theory behind its methods, this book focuses on the bare bones of machine learning algorithms. Its main purpose is to provide readers with the ability to construct these algorithms independently. Continuing the toolbox analogy, this book is intended as a user guide: it is not designed to teach users broad practices of the field but rather how each tool works at a micro level.

Who this Book is for

This book is for readers looking to learn new machine learning algorithms or understand algorithms at a deeper level. Specifically, it is intended for readers interested in seeing machine learning algorithms derived from start to finish. Seeing these derivations might help a reader previously unfamiliar with common algorithms understand how they work intuitively. Or, seeing these derivations might help a reader experienced in modeling understand how different algorithms create the models they do and the advantages and disadvantages of each one.

This book will be most helpful for those with practice in basic modeling. It does not review best practices—such as feature engineering or balancing response variables—or discuss in depth when certain models are more appropriate than others. Instead, it focuses on the elements of those models.

What Readers Should Know

The *concept* sections of this book primarily require knowledge of calculus, though some require an understanding of probability (think maximum likelihood and Bayes' Rule) and basic linear algebra (think matrix operations and dot products). The appendix reviews the math and probabilityneeded to understand this book. The concept sections also reference a few common machine learning methods, which are introduced in the appendix as well. The concept sections do not require any knowledge of programming.

The *construction* and *code* sections of this book use some basic Python. The construction sections require understanding of the corresponding content sections and familiarity creating functions and classes in Python. The code sections require neither.

Where to Ask Questions or Give Feedback

You can raise an issue here or email me at dafrdman@gmail.com.

Table of Contents

Conventions and Notation

The following terminology will be used throughout the book.

- Variables can be split into two types: the variables we intend to model are referred to as **target** or **output** variables, while the variables we use to model the target variables are referred to as **predictors**, **features**, or **input** variables. These are also known as the *dependent* and *independent* variables, respectively.
- An **observation** is a single collection of predictors and target variables. Multiple observations with the same variables are combined to form a **dataset**.
- A **training** dataset is one used to build a machine learning model. A **validation** dataset is one used to compare multiple models built on the same training dataset with different parameters. A **testing** dataset is one used to evaluate a final model.
- Variables, whether predictors or targets, may be **quantitative** or **categorical**. Quantitative variables follow a continuous or near-contih234nuous scale (such as height in inches or income in dollars). Categorical variables fall in one of a discrete set of groups (such as nation of birth or species type). While the values of categorical variables may follow some natural order (such as shirt size), this is not assumed.
- Modeling tasks are referred to as **regression** if the target is quantitative and **classification** if the target is categorical. Note that regression does not necessarily refer to ordinary least squares (OLS) linear regression.

Unless indicated otherwise, the following conventions are used to represent data and datasets.

- Training datasets are assumed to have N observations and D predictors.
- The vector of features for the n^{th} observation is given by $\mathbf{x}_n$. Note that $\mathbf{x}_n$ might include functions of the original predictors through feature engineering. When the target variable is single-dimensional (i.e. there is only one target variable per observation), it is given by y_n; when there are multiple target variables per observation, the vector of targets is given by $\mathbf{y}_n$.
- The entire collection of input and output data is often represented with $\{\mathbf{x}_n, y_n\}_{n=1}^N$, which implies observation n has a multi-dimensional predictor vector $\mathbf{x}_n$ and a target variable y_n for $n = 1, 2, \dots, N$.
- Many models, such as ordinary linear regression, append an intercept term to the predictor vector. When this is the case, $\mathbf{x}_n$ will be defined as

$$\mathbf{x}_n = \begin{pmatrix} 1 & x_{n1} & x_{n2} & \dots & x_{nD} \end{pmatrix}.$$

- *Feature matrices* or *data frames* are created by concatenating feature vectors across observations. Within a matrix, feature vectors are row vectors, with $\mathbf{x}_n$ representing the matrix's n^{th} row. These matrices are then given by $\mathbf{X}$. If a leading 1 is appended to each $\mathbf{x}_n$, the first column of the corresponding feature matrix $\mathbf{X}$ will consist of only 1s.

Finally, the following mathematical and notational conventions are used.

- Scalar values will be non-boldface and lowercase, random variables will be non-boldface and uppercase, vectors will be bold and lowercase, and matrices will be bold and uppercase. E.g. b is a scalar, B a random variable, $\mathbf{b}$ a vector, and $\mathbf{B}$ a matrix.
- Unless indicated otherwise, all vectors are assumed to be column vectors. Since feature vectors (such as $\mathbf{x}_n$ and $\boldsymbol{\phi}_n$ above) are entered into data frames as rows, they will sometimes be treated as row vectors, even outside of data frames.
- Matrix or vector derivatives, covered in the math appendix, will use the numerator layout convention. Let $\mathbf{y} \in \mathbb{R}^a$ and $\mathbf{x} \in \mathbb{R}^b$; under this convention, the derivative $\partial\mathbf{y}/\partial\mathbf{x}$ is written as

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_b} \\ \frac{\partial y_2}{\partial x_1} & \dots & \frac{\partial y_2}{\partial x_b} \\ & \dots & \\ \frac{\partial y_a}{\partial x_1} & \dots & \frac{\partial y_a}{\partial x_b} \end{pmatrix}.$$

- The likelihood of a parameter θ given data $\{x_n\}_{n=1}^N$ is represented by $\mathcal{L}\left(\theta; \{x_n\}_{n=1}^N\right)$. If we are considering the data to be random (i.e. not yet observed), it will be written as $\{X_n\}_{n=1}^N$. If the data in consideration is obvious, we may write the likelihood as just $\mathcal{L}(\theta)$.

Concept

Model Structure

Linear regression is a relatively simple method that is extremely widely-used. It is also a great stepping stone for more sophisticated methods, making it a natural algorithm to study first.

In linear regression, the target variable y is assumed to follow a linear function of one or more predictor variables, $x_1, \dots, x_D$, plus some random error. Specifically, we assume the model for the n^{th} observation in our sample is of the form

$$y_n = \beta_0 + \beta_1 x_{n1} + \dots + \beta_D x_{nD} + \epsilon_n.$$

Here β_0 is the intercept term, β_1 through β_D are the coefficients on our feature variables, and ϵ is an error term that represents the difference between the true y value and the linear function of the predictors. Note that the terms with an n in the subscript differ between observations while the terms without (namely the βs) do not.

The math behind linear regression often becomes easier when we use vectors to represent our predictors and coefficients. Let's define $\mathbf{x}_n$ and $\boldsymbol{\beta}$ as follows:

$$\begin{aligned} \mathbf{x}_n &= \begin{pmatrix} 1 & x_{n1} & \dots & x_{nD} \end{pmatrix}^\top \\ \boldsymbol{\beta} &= \begin{pmatrix} \beta_0 & \beta_1 & \dots & \beta_D \end{pmatrix}^\top. \end{aligned}$$

Note that $\mathbf{x}_n$ includes a leading 1, corresponding to the intercept term β_0. Using these definitions, we can equivalently express y_n as

$$y_n = \boldsymbol{\beta}^\top \mathbf{x}_n + \epsilon_n.$$

Below is an example of a dataset designed for linear regression. The input variable is generated randomly and the target variable is generated as a linear combination of that input variable plus an error term.

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# generate data
np.random.seed(123)
N = 20
beta0 = -4
beta1 = 2
x = np.random.randn(N)
e = np.random.randn(N)
y = beta0 + beta1*x + e
true_x = np.linspace(min(x), max(x), 100)
true_y = beta0 + beta1*true_x

# plot
fig, ax = plt.subplots()
sns.scatterplot(x, y, s = 40, label = 'Data')
sns.lineplot(true_x, true_y, color = 'red', label = 'True Model')
ax.set_xlabel('x', fontsize = 14)
ax.set_title(fr"$y = {beta0} + ${beta1}$x + \epsilon$", fontsize = 16)
ax.set_ylabel('y', fontsize=14, rotation=0, labelpad=10)
ax.legend(loc = 4)
sns.despine()
```

../../_images/concept_2_0.png

Parameter Estimation

The previous section covers the entire structure we assume our data follows in linear regression. The machine learning task is then to estimate the parameters in $\boldsymbol{\beta}$. These estimates are represented by $\hat{\beta}_0, \dots, \hat{\beta}_D$ or $\hat{\boldsymbol{\beta}}$. The estimates give us *fitted values* for our target variable, represented by $\hat{y}_n$.

This task can be accomplished in two ways which, though slightly different conceptually, are identical mathematically. The first approach is through the lens of *minimizing loss*. A common practice in machine learning is to choose a loss function that defines how well a model with a given set of parameter estimates the observed data. The most common loss function for linear regression is squared error loss. This says the *loss* of our model is proportional to the sum of squared differences between the true y_n values and the fitted values, $\hat{y}_n$. We then *fit* the model by finding the estimates $\hat{\boldsymbol{\beta}}$ that minimize this loss function. This approach is covered in the subsection Approach 1: Minimizing Loss.

The second approach is through the lens of *maximizing likelihood*. Another common practice in machine learning is to model the target as a random variable whose distribution depends on one or more parameters, and then find the parameters that maximize its likelihood. Under this approach, we will represent the target with Y_n since we are treating it as a random variable. The most common model for Y_n in linear regression is a Normal random variable with mean $E(Y_n) = \boldsymbol{\beta}^\top \mathbf{x}_n$. That is, we assume

$$Y_n | \mathbf{x}_n \sim \mathcal{N}(\boldsymbol{\beta}^\top \mathbf{x}_n, \sigma^2),$$

and we find the values of $\hat{\boldsymbol{\beta}}$ to maximize the likelihood. This approach is covered in subsection Approach 2: Maximizing Likelihood.

Once we've estimated $\boldsymbol{\beta}$, our model is *fit* and we can make predictions. The below graph is the same as the one above but includes our estimated line-of-best-fit, obtained by calculating $\hat{\beta}_0$ and $\hat{\beta}_1$.

```python
# generate data
np.random.seed(123)
N = 20
beta0 = -4
beta1 = 2
x = np.random.randn(N)
e = np.random.randn(N)
y = beta0 + beta1*x + e
true_x = np.linspace(min(x), max(x), 100)
true_y = beta0 + beta1*true_x

# estimate model
beta1_hat = sum((x - np.mean(x))*(y - np.mean(y)))/sum((x - np.mean(x))**2)
beta0_hat = np.mean(y) - beta1_hat*np.mean(x)
fit_y = beta0_hat + beta1_hat*true_x

# plot
fig, ax = plt.subplots()
sns.scatterplot(x, y, s = 40, label = 'Data')
sns.lineplot(true_x, true_y, color = 'red', label = 'True Model')
sns.lineplot(true_x, fit_y, color = 'purple', label = 'Estimated Model')
ax.set_xlabel('x', fontsize = 14)
ax.set_title(fr"Linear Regression for $y = {beta0} + ${beta1}$x + \epsilon$", fontsize
= 16)
ax.set_ylabel('y', fontsize=14, rotation=0, labelpad=10)
ax.legend(loc = 4)
sns.despine()
```

../../_images/concept_4_0.png

Extensions of Ordinary Linear Regression

There are many important extensions to linear regression which make the model more flexible. Those include Regularized Regression—which balances the bias-variance tradeoff for high-dimensional regression models—Bayesian Regression—which allows for prior distributions on the coefficients—and GLMs—which introduce non-linearity to regression models. These extensions are discussed in the next chapter.

Approach 1: Minimizing Loss

1. Simple Linear Regression

Model Structure

Simple linear regression models the target variable, y, as a linear function of just one predictor variable, x, plus an error term, ϵ. We can write the entire model for the n^{th} observation as

$$y_n = \beta_0 + \beta_1 x_n + \epsilon_n.$$

Fitting the model then consists of estimating two parameters: β_0 and β_1. We call our estimates of these parameters $\hat{\beta}_0$ and $\hat{\beta}_1$, respectively. Once we've made these estimates, we can form our prediction for any given x_n with

$$\hat{y}_n = \hat{\beta}_0 + \hat{\beta}_1 x_n.$$

One way to find these estimates is by minimizing a loss function. Typically, this loss function is the *residual sum of squares* (RSS). The RSS is calculated with

$$\mathcal{L}(\hat{\beta}_0, \hat{\beta}_1) = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \hat{y}_n\right)^2.$$

We divide the sum of squared errors by 2 in order to simplify the math, as shown below. Note that doing this does not affect our estimates because it does not affect which $\hat{\beta}_0$ and $\hat{\beta}_1$ minimize the RSS.

Parameter Estimation

Having chosen a loss function, we are ready to derive our estimates. First, let's rewrite the RSS in terms of the estimates:

$$\mathcal{L}(\hat{\beta}_0, \hat{\beta}_1) = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \left(\hat{\beta}_0 + \hat{\beta}_1 x_n\right)\right)^2.$$

To find the intercept estimate, start by taking the derivative of the RSS with respect to $\hat{\beta}_0$:

$$\begin{aligned}\frac{\partial \mathcal{L}(\hat{\beta}_0, \hat{\beta}_1)}{\partial \hat{\beta}_0} &= -\sum_{n=1}^{N} \left(y_n - \hat{\beta}_0 - \hat{\beta}_1 x_n \right) \\ &= -N(\bar{y} - \hat{\beta}_0 - \hat{\beta}_1 \bar{x}),\end{aligned}$$

where $\bar{y}$ and $\bar{x}$ are the sample means. Then set that derivative equal to 0 and solve for $\hat{\beta}_0$:

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}.$$

This gives our intercept estimate, $\hat{\beta}_0$, in terms of the slope estimate, $\hat{\beta}_1$. To find the slope estimate, again start by taking the derivative of the RSS:

$$\frac{\partial \mathcal{L}(\hat{\beta}_0, \hat{\beta}_1)}{\partial \hat{\beta}_1} = -\sum_{n=1}^{N} \left(y_n - \hat{\beta}_0 - \hat{\beta}_1 x_n \right) x_n.$$

Setting this equal to 0 and substituting for $\hat{\beta}_0$, we get

$$\begin{aligned}\sum_{n=1}^{N} \left(y_n - (\bar{y} - \hat{\beta}_1 \bar{x}) - \hat{\beta}_1 x_n \right) x_n &= 0 \\ \hat{\beta}_1 \sum_{n=1}^{N} (x_n - \bar{x}) x_n &= \sum_{n=1}^{N} (y_n - \bar{y})\, x_n \\ \hat{\beta}_1 &= \frac{\sum_{n=1}^{N} x_n (y_n - \bar{y})}{\sum_{n=1}^{N} x_n (x_n - \bar{x})}.\end{aligned}$$

To put this in a more standard form, we use a slight algebra trick. Note that

$$\sum_{n=1}^{N} c(z_n - \bar{z}) = 0$$

for any constant c and any collection $z_1, \dots, z_N$ with sample mean $\bar{z}$ (this can easily be verified by expanding the sum). Since $\bar{x}$ is a constant, we can then subtract $\sum_{n=1}^{N} \bar{x}(y_n - \bar{y})$ from the numerator and $\sum_{n=1}^{N} \bar{x}(x_n - \bar{x})$ from the denominator without affecting our slope estimate. Finally, we get

$$\hat{\beta}_1 = \frac{\sum_{n=1}^{N} (x_n - \bar{x})(y_n - \bar{y})}{\sum_{n=1}^{N} (x_n - \bar{x})^2}.$$

2. Multiple Regression

Model Structure

In multiple regression, we assume our target variable to be a linear combination of *multiple* predictor variables. Letting x_{nj} be the j^{th} predictor for observation n, we can write the model as

$$y_n = \beta_0 + \beta_1 x_{n1} + \dots + \beta_D x_{nD} + \epsilon_n.$$

Using the vectors $\mathbf{x}_n$ and $\boldsymbol{\beta}$ defined in the previous section, this can be written more compactly as

$$y_n = \boldsymbol{\beta}^\top \mathbf{x}_n + \epsilon_n.$$

Then define $\hat{\boldsymbol{\beta}}$ the same way as $\boldsymbol{\beta}$ except replace the parameters with their estimates. We again want to find the vector $\hat{\boldsymbol{\beta}}$ that minimizes the RSS:

$$\mathcal{L}(\hat{\boldsymbol{\beta}}) = \frac{1}{2} \sum_{n=1}^{N} \left(y_n - \hat{\boldsymbol{\beta}}^\top \mathbf{x}_n \right)^2 = \frac{1}{2} \sum_{n=1}^{N} (y_n - \hat{y}_n)^2,$$

Minimizing this loss function is easier when working with matrices rather than sums. Define $\mathbf{y}$ and $\mathbf{X}$ with

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \dots \\ y_N \end{bmatrix} \in \mathbb{R}^N, \quad \mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \dots \\ \mathbf{x}_N^\top \end{bmatrix} \in \mathbb{R}^{N \times (D+1)},$$

which gives $\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}} \in \mathbb{R}^N$. Then, we can equivalently write the loss function as

$$\mathcal{L}(\hat{\boldsymbol{\beta}}) = \frac{1}{2}(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}})^\top (\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}).$$

Parameter Estimation

We can estimate the parameters in the same way as we did for simple linear regression, only this time calculating the derivative of the RSS with respect to the entire parameter vector. First, note the commonly-used matrix derivative below [1].

Math Note

For a symmetric matrix $\mathbf{W}$,

$$\frac{\partial}{\partial \mathbf{s}}(\mathbf{q}-\mathbf{A}\mathbf{s})^\top \mathbf{W}\,(\mathbf{q}-\mathbf{A}\mathbf{s}) = -2\mathbf{A}^\top \mathbf{W}\,(\mathbf{q}-\mathbf{A}\mathbf{s})$$

Applying the result of the Math Note, we get the derivative of the RSS with respect to $\hat{\boldsymbol{\beta}}$ (note that the identity matrix takes the place of $\mathbf{W}$):

$$\begin{aligned}\mathcal{L}(\hat{\boldsymbol{\beta}}) &= \frac{1}{2}(\mathbf{y}-\mathbf{X}\hat{\boldsymbol{\beta}})^\top(\mathbf{y}-\mathbf{X}\hat{\boldsymbol{\beta}}) \\ \frac{\partial \mathcal{L}(\hat{\boldsymbol{\beta}})}{\partial \hat{\boldsymbol{\beta}}} &= -\mathbf{X}^\top(\mathbf{y}-\mathbf{X}\hat{\boldsymbol{\beta}}).\end{aligned}$$

We get our parameter estimates by setting this derivative equal to 0 and solving for $\hat{\boldsymbol{\beta}}$:

$$\begin{aligned}(\mathbf{X}^\top\boldsymbol{\Phi})\hat{\boldsymbol{\beta}} &= \mathbf{X}^\top\mathbf{y} \\ \hat{\boldsymbol{\beta}} &= (\mathbf{X}^\top\mathbf{X})^\top\mathbf{X}^\top\mathbf{y}\end{aligned}$$

[1] A helpful guide for matrix calculus is The Matrix Cookbook

Approach 2: Maximizing Likelihood

1. Simple Linear Regression

Model Structure

Using the maximum likelihood approach, we set up the regression model probabilistically. Since we are treating the target as a random variable, we will capitalize it. As before, we assume

$$Y_n = \beta_0 + \beta_1 x_n + \epsilon_n,$$

only now we give ϵ_n a distribution (we don't do the same for x_n since its value is known). Typically, we assume the ϵ_n are independently Normally distributed with mean 0 and an unknown variance. That is,

$$\epsilon_n \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, \sigma^2).$$

The assumption that the variance is identical across observations is called *homoskedasticity*. This is required for the following derivations, though there are *heteroskedasticity-robust* estimates that do not make this assumption.

Since β_0 and β_1 are fixed parameters and x_n is known, the only source of randomness in Y_n is ϵ_n. Therefore,

$$Y_n \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\beta_0 + \beta_1 x_n, \sigma^2),$$

since a Normal random variable plus a constant is another Normal random variable with a shifted mean.

Parameter Estimation

The task of fitting the linear regression model then consists of estimating the parameters with maximum likelihood. The joint likelihood and log-likelihood across observations are as follows.

$$
\begin{aligned}
L(\beta_0, \beta_1; Y_1, \dots, Y_N) &= \prod_{n=1}^{N} L(\beta_0, \beta_1; Y_n) \\
&= \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(Y_n - (\beta_0 + \beta_1 x_n))^2}{2\sigma^2}\right) \\
&\propto \exp\left(-\sum_{n=1}^{N} \frac{(Y_n - (\beta_0 + \beta_1 x_n))^2}{2\sigma^2}\right) \\
\log L(\beta_0, \beta_1; Y_1, \dots, Y_N) &= -\frac{1}{2\sigma^2} \sum_{n=1}^{N} (Y_n - (\beta_0 + \beta_1 x_n))^2.
\end{aligned}
$$

Our $\hat{\beta}_0$ and $\hat{\beta}_1$ estimates are the values that maximize the log-likelihood given above. Notice that this is equivalent to finding the $\hat{\beta}_0$ and $\hat{\beta}_1$ that minimize the RSS, our loss function from the previous section:

$$
\text{RSS} = \frac{1}{2} \sum_{n=1}^{N} \left(y_n - \left(\hat{\beta}_0 + \hat{\beta}_1 x_n\right)\right)^2.
$$

In other words, we are solving the same optimization problem we did in the last section. Since it's the same problem, it has the same solution! (This can also of course be checked by differentiating and optimizing for $\hat{\beta}_0$ and $\hat{\beta}_1$). Therefore, as with the loss minimization approach, the parameter estimates from the likelihood maximization approach are

$$
\begin{aligned}
\hat{\beta}_0 &= \bar{Y} - \hat{\beta}_1 \bar{x} \\
\hat{\beta}_1 &= \frac{\sum_{n=1}^{N} (x_n - \bar{x})(Y_n - \bar{Y})}{\sum_{n=1}^{N} (x_n - \bar{x})^2}.
\end{aligned}
$$

2. Multiple Regression

Still assuming Normally-distributed errors but adding more than one predictor, we have

$$
Y_n \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\boldsymbol{\beta}^\top \mathbf{x}_n, \sigma^2).
$$

We can then solve the same maximum likelihood problem. Calculating the log-likelihood as we did above for simple linear regression, we have

$$
\begin{aligned}
\log L(\beta_0, \beta_1; Y_1, \dots, Y_N) &= -\frac{1}{2\sigma^2} \sum_{n=1}^{N} \left(Y_n - \boldsymbol{\beta}^\top \mathbf{x}_n\right)^2 \\
&= -\frac{1}{2\sigma^2} (\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}})^\top (\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}).
\end{aligned}
$$

Again, maximizing this quantity is the same as minimizing the RSS, as we did under the loss minimization approach. We therefore obtain the same solution:

$$
\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.
$$

Construction

This section demonstrates how to construct a linear regression model using only `numpy`. To do this, we generate a class named `LinearRegression`. We use this class to train the model and make future predictions.

The first method in the `LinearRegression` class is `fit()`, which takes care of estimating the $\boldsymbol{\beta}$ parameters. This simply consists of calculating

$$
\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}
$$

The `fit` method also makes in-sample predictions with $\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}$ and calculates the training loss with

$$
\mathcal{L}(\hat{\boldsymbol{\beta}}) = \frac{1}{2} \sum_{n=1}^{N} \left(y_n - \hat{y}_n\right)^2.
$$

The second method is `predict()`, which forms out-of-sample predictions. Given a test set of predictors $\mathbf{X}'$, we can form fitted values with $\hat{\mathbf{y}}' = \mathbf{X}' \hat{\boldsymbol{\beta}}$.

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```python
class LinearRegression:

    def fit(self, X, y, intercept = False):

        # record data and dimensions
        if intercept == False: # add intercept (if not already included)
            ones = np.ones(len(X)).reshape(len(X), 1) # column of ones
            X = np.concatenate((ones, X), axis = 1)
        self.X = np.array(X)
        self.y = np.array(y)
        self.N, self.D = self.X.shape

        # estimate parameters
        XtX = np.dot(self.X.T, self.X)
        XtX_inverse = np.linalg.inv(XtX)
        Xty = np.dot(self.X.T, self.y)
        self.beta_hats = np.dot(XtX_inverse, Xty)

        # make in-sample predictions
        self.y_hat = np.dot(self.X, self.beta_hats)

        # calculate loss
        self.L = .5*np.sum((self.y - self.y_hat)**2)

    def predict(self, X_test, intercept = True):

        # form predictions
        self.y_test_hat = np.dot(X_test, self.beta_hats)
```

Let's try out our `LinearRegression` class with some data. Here we use the Boston housing dataset from `sklearn.datasets`. The target variable in this dataset is median neighborhood home value. The predictors are all continuous and represent factors possibly related to the median home value, such as average rooms per house. Hit "Click to show" to see the code that loads this data.

```python
from sklearn import datasets
boston = datasets.load_boston()
X = boston['data']
y = boston['target']
```

With the class built and the data loaded, we are ready to run our regression model. This is as simple as instantiating the model and applying `fit()`, as shown below.

```python
model = LinearRegression() # instantiate model
model.fit(X, y, intercept = False) # fit model
```

Let's then see how well our fitted values model the true target values. The closer the points lie to the 45-degree line, the more accurate the fit. The model seems to do reasonably well; our predictions definitely follow the true values quite well, although we would like the fit to be a bit tighter.

Note

Note the handful of observations with $y = 50$ exactly. This is due to censorship in the data collection process. It appears neighborhoods with average home values above $50,000 were assigned a value of 50 even.

```python
fig, ax = plt.subplots()
sns.scatterplot(model.y, model.y_hat)
ax.set_xlabel(r'$y$', size = 16)
ax.set_ylabel(r'$\hat{y}$', rotation = 0, size = 16, labelpad = 15)
ax.set_title(r'$y$ vs. $\hat{y}$', size = 20, pad = 10)
sns.despine()
```

../../_images/construction_10_0.png

Implementation

This section demonstrates how to fit a regression model in Python in practice. The two most common packages for fitting regression models in Python are `scikit-learn` and `statsmodels`. Both methods are shown before.

First, let's import the data and necessary packages. We'll again be using the Boston housing dataset from `sklearn.datasets`.

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
boston = datasets.load_boston()
X_train = boston['data']
y_train = boston['target']
```

Scikit-Learn

Fitting the model in `scikit-learn` is very similar to how we fit our model from scratch in the previous section. The model is fit in two steps: first instantiate the model and second use the `fit()` method to train it.

```
from sklearn.linear_model import LinearRegression
sklearn_model = LinearRegression()
sklearn_model.fit(X_train, y_train);
```

As before, we can plot our fitted values against the true values. To form predictions with the `scikit-learn` model, we can use the `predict` method. Reassuringly, we get the same plot as before.

```
sklearn_predictions = sklearn_model.predict(X_train)
fig, ax = plt.subplots()
sns.scatterplot(y_train, sklearn_predictions)
ax.set_xlabel(r'$y$', size = 16)
ax.set_ylabel(r'$\hat{y}$', rotation = 0, size = 16, labelpad = 15)
ax.set_title(r'$y$ vs. $\hat{y}$', size = 20, pad = 10)
sns.despine()
```

../../_images/code_7_0.png

We can also check the estimated parameters using the `coef_` attribute as follows (note that only the first few are printed).

```
predictors = boston.feature_names
beta_hats = sklearn_model.coef_
print('\n'.join([f'{predictors[i]}: {round(beta_hats[i], 3)}' for i in range(3)]))
```

```
CRIM: -0.108
ZN: 0.046
INDUS: 0.021
```

Statsmodels

`statsmodels` is another package frequently used for running linear regression in Python. There are two ways to run regression in `statsmodels`. The first uses `numpy` arrays like we did in the previous section. An example is given below.

Note

Note two subtle differences between this model and the models we've previously built. First, we have to manually add a constant to the predictor dataframe in order to give our model an intercept term. Second, we supply the training data when *instantiating* the model, rather than when fitting it.

```
import statsmodels.api as sm

X_train_with_constant = sm.add_constant(X_train)
sm_model1 = sm.OLS(y_train, X_train_with_constant)
sm_fit1 = sm_model1.fit()
sm_predictions1 = sm_fit1.predict(X_train_with_constant)
```

The second way to run regression in `statsmodels` is with `R`-style formulas and `pandas` dataframes. This allows us to identify predictors and target variables by name. An example is given below.

```python
import pandas as pd
df = pd.DataFrame(X_train, columns = boston['feature_names'])
df['target'] = y_train
display(df.head())

formula = 'target ~ ' + ' + '.join(boston['feature_names'])
print('formula:', formula)
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.9
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.9
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.8
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.6
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.9

```
formula: target ~ CRIM + ZN + INDUS + CHAS + NOX + RM + AGE + DIS + RAD + TAX +
PTRATIO + B + LSTAT
```

```python
import statsmodels.formula.api as smf

sm_model2 = smf.ols(formula, data = df)
sm_fit2 = sm_model2.fit()
sm_predictions2 = sm_fit2.predict(df)
```

Concept

Linear regression can be extended in a number of ways to fit various modeling needs. Regularized regression penalizes the magnitude of the regression coefficients to avoid *overfitting*, which is particularly helpful for models using a large number of predictors. Bayesian regression places a prior distribution on the regression coefficients in order to reconcile existing beliefs about these parameters with information gained from new data. Finally, generalized linear models (GLMs) expand on ordinary linear regression by changing the assumed error structure and allowing for the expected value of the target variable to be a nonlinear function of the predictors. These extensions are described, derived, and demonstrated in detail this chapter.

Regularized Regression

Regression models, especially those fit to high-dimensional data, may be prone to overfitting. One way to ameliorate this issue is by penalizing the magnitude of the $\hat{\boldsymbol{\beta}}$ coefficient estimates. This has the effect of shrinking these estimates toward 0, which ideally prevents the model from capturing spurious relationships between weak predictors and the target variable.

This section reviews the two most common methods for regularized regression: Ridge and Lasso.

Ridge Regression

Like ordinary linear regression, Ridge regression estimates the coefficients by minimizing a loss function on the training data. Unlike ordinary linear regression, the loss function for Ridge regression penalizes large values of the $\hat{\boldsymbol{\beta}}$ estimates. Specifically, Ridge regression minimizes the sum of the RSS and the L2 norm of $\hat{\boldsymbol{\beta}}$:

$$\mathcal{L}_{\text{Ridge}}(\hat{\boldsymbol{\beta}}) = \frac{1}{2}\left(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}\right)^\top \left(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}\right) + \frac{\lambda}{2}\sum_{d=1}^{D}\hat{\beta}_d^2.$$

Here, λ is a *tuning parameter* which represents the amount of regularization. A large λ means a greater penalty on the $\hat{\boldsymbol{\beta}}$ estimates, meaning more shrinkage of these estimates toward 0. λ is not estimated by the model but rather chosen before fitting, typically through cross validation.

Note

Note that the Ridge loss function does *not* penalize the magnitude of the intercept estimate, $\hat{\beta}_0$. Intuitively, a greater intercept does not suggest overfitting.

As in ordinary linear regression, we start estimating $\hat{\boldsymbol{\beta}}$ by taking the derivative of the loss function. First note that since $\hat{\beta}_0$ is not penalized,

$$\begin{aligned}\frac{\partial}{\partial \hat{\boldsymbol{\beta}}}\left(\frac{\lambda}{2}\sum_{d=1}^{D}\hat{\beta}_d^2\right) &= \begin{bmatrix} 0 \\ \lambda\hat{\beta}_1 \\ \dots \\ \lambda\hat{\beta}_D \end{bmatrix} \\ &= \lambda I'\hat{\boldsymbol{\beta}},\end{aligned}$$

where I' is the identity matrix of size $D+1$ *except* the first element is a 0. Then, adding in the derivative of the RSS discussed in chapter 1, we get

$$\frac{\partial \mathcal{L}_{\text{Ridge}}(\hat{\boldsymbol{\beta}})}{\partial \hat{\boldsymbol{\beta}}} = -\mathbf{X}^\top\left(\mathbf{y}-\mathbf{X}\hat{\boldsymbol{\beta}}\right) + \lambda I'\hat{\boldsymbol{\beta}}.$$

Setting this equal to 0 and solving for $\hat{\boldsymbol{\beta}}$, we get our estimates:

$$\begin{aligned}\hat{\boldsymbol{\beta}}\left(\mathbf{X}^\top\mathbf{X}+\lambda I'\right) &= \mathbf{X}^\top\mathbf{y} \\ \hat{\boldsymbol{\beta}} &= \left(\mathbf{X}^\top\mathbf{X}+\lambda I'\right)^{-1}\mathbf{X}^\top\mathbf{y},\end{aligned}$$

Lasso Regression

Lasso regression differs from Ridge regression in that its loss function uses the L1 norm for the $\hat{\boldsymbol{\beta}}$ estimates rather than the L2 norm. This means we penalize the sum of absolute values of the $\hat{\beta}$s, rather than the sum of their squares.

$$\mathcal{L}_{\text{Lasso}}(\hat{\boldsymbol{\beta}}) = \frac{1}{2}\left(\mathbf{y}-\mathbf{X}\hat{\boldsymbol{\beta}}\right)^\top\left(\mathbf{y}-\mathbf{X}\hat{\boldsymbol{\beta}}\right) + \lambda\sum_{d=1}^{D}|\hat{\beta}_d|.$$

As usual, let's then calculate the gradient of the loss function with respect to $\hat{\boldsymbol{\beta}}$:

$$\frac{\partial \mathcal{L}(\hat{\boldsymbol{\beta}})}{\partial \hat{\boldsymbol{\beta}}} = -\mathbf{X}^\top\left(\mathbf{y}-\mathbf{X}\hat{\boldsymbol{\beta}}\right) + \lambda I' \,\text{sign}(\hat{\boldsymbol{\beta}}),$$

where again we use I' rather than I since the magnitude of the intercept estimate $\hat{\beta}_0$ is not penalized.

Unfortunately, we cannot find a closed-form solution for the $\hat{\boldsymbol{\beta}}$ that minimize the Lasso loss. Numerous methods exist for estimating the $\hat{\boldsymbol{\beta}}$, though using the gradient calculated above we could easily reach an estimate through gradient descent. The construction in the next section uses this approach.

Bayesian Regression

In the Bayesian approach to statistical inference, we treat our parameters as random variables and assign them a prior distribution. This forces our estimates to reconcile our existing beliefs about these parameters with new information given by the data. This approach can be applied to linear regression by assigning the regression coefficients a prior distribution.

We also may wish to perform Bayesian regression not because of a prior belief about the coefficients but in order to minimize model complexity. By assigning the parameters a prior distribution with mean 0, we force the posterior estimates to be closer to 0 than they would otherwise. This is a form of regularization similar to the Ridge and Lasso methods discussed in the previous section.

The Bayesian Structure

To demonstrate Bayesian regression, we'll follow three typical steps to Bayesian analysis: writing the likelihood, writing the prior density, and using Bayes' Rule to get the posterior density. In the results below, we use the posterior density to calculate the maximum-a-posteriori (MAP)—the equivalent of calculating the $\hat{\boldsymbol{\beta}}$ estimates in ordinary linear regression.

1. The Likelihood

As in the typical regression set-up, let's assume

$$Y_n \overset{\text{i.i.d.}}{\sim} \mathcal{N}\left(\boldsymbol{\beta}^\top \mathbf{x}_n, \sigma^2\right).$$

We can write the collection of observations jointly as

$$\mathbf{y} \sim \mathcal{N}\left(\mathbf{X}\boldsymbol{\beta}, \boldsymbol{\Sigma}\right),$$

where $\mathbf{y} \in \mathbb{R}^N$ and $\boldsymbol{\Sigma} = \sigma^2 I_N \in \mathbb{R}^{N \times N}$ for some *known* scalar σ^2. Note that $\mathbf{y}$ is a vector of random variables —it is not capitalized in order to distinguish it from a matrix.

> **Note**
>
> See this lecture for an example of Bayesian regression without the assumption of known variance.

We can then get our likelihood and log-likelihood using the Multivariate Normal.

$$\begin{aligned}
L(\boldsymbol{\beta}; \mathbf{X}, \mathbf{y}) &= \frac{1}{\sqrt{(2\pi)^N |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\right) \\
&\propto \exp\left(-\frac{1}{2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\right) \\
\log L(\boldsymbol{\beta}; \mathbf{X}, \mathbf{y}) &= -\frac{1}{2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}).
\end{aligned}$$

2. The Prior

Now, let's assign $\boldsymbol{\beta}$ a prior distribution. We typically assume

$$\boldsymbol{\beta} \sim \mathcal{N}(\mathbf{0}, \mathbf{T}),$$

where $\boldsymbol{\beta} \in \mathbb{R}^D$ and $\mathbf{T} = \tau I_D \in \mathbb{R}^{D \times D}$ for some scalar τ. We choose τ (and therefore $\mathbf{T}$) ourselves, with a greater τ giving less weight to the prior.

The prior density is given by

$$\begin{aligned}
p(\boldsymbol{\beta}) &= \frac{1}{\sqrt{(2\pi)^D |\mathbf{T}|}} \exp\left(-\frac{1}{2}\boldsymbol{\beta}^\top \mathbf{T}^{-1}\boldsymbol{\beta}\right) \\
&\propto \exp\left(-\frac{1}{2}\boldsymbol{\beta}^\top \mathbf{T}^{-1}\boldsymbol{\beta}\right) \\
\log p(\boldsymbol{\beta}) &= -\frac{1}{2}\boldsymbol{\beta}^\top \mathbf{T}^{-1}\boldsymbol{\beta}.
\end{aligned}$$

3. The Posterior

We are then interested in a posterior density of $\boldsymbol{\beta}$ given the data, $\mathbf{X}$ and $\mathbf{y}$.

Bayes' rule tells us that the posterior density of the coefficients is proportional to the likelihood of the data times the prior density of the coefficients. Using the two previous results, we have

$$\begin{aligned}
p(\boldsymbol{\beta}|\mathbf{X}, \mathbf{y}) &\propto L(\boldsymbol{\beta}; \mathbf{X}, \mathbf{y})p(\boldsymbol{\beta}) \\
\log p(\boldsymbol{\beta}|\mathbf{X}, \mathbf{y}) &= \log L(\boldsymbol{\beta}; \mathbf{X}, \mathbf{y}) + \log p(\boldsymbol{\beta}) + k \\
&= -\frac{1}{2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) - \frac{1}{2}\boldsymbol{\beta}^\top \mathbf{T}^{-1}\boldsymbol{\beta} + k \\
&= -\frac{1}{2\sigma^2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) - \frac{1}{2\tau}\boldsymbol{\beta}^\top\boldsymbol{\beta} + k
\end{aligned}$$

where k is some constant that we don't care about.

Results

Intuition

Often in the Bayesian setting it is infeasible to obtain the entire posterior distribution. Instead, one typically looks at the maximum-a-posteriori (MAP), the value of the parameters that maximize the posterior density. In our case, the MAP is the $\hat{\boldsymbol{\beta}}$ that maximizes

$$\log p(\hat{\boldsymbol{\beta}}|\mathbf{X}, \mathbf{y}) = -\frac{1}{2\sigma^2}(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}})^\top(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}) - \frac{1}{2\tau}\hat{\boldsymbol{\beta}}^\top\hat{\boldsymbol{\beta}}.$$

This is equivalent to finding the $\hat{\boldsymbol{\beta}}$ that minimizes the following loss function, where $\lambda = 1/\tau$.

$$\begin{aligned} L(\hat{\boldsymbol{\beta}}) &= \frac{1}{2}(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}})^\top(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}) + \frac{\lambda}{2}\hat{\boldsymbol{\beta}}^\top\hat{\boldsymbol{\beta}} \\ &= \frac{1}{2}(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}})^\top(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}) + \frac{\lambda}{2}\sum_{d=0}^{D}\hat{\beta}_d. \end{aligned}$$

Notice that this is extremely close to the Ridge loss function discussed in the previous section—it is not quite equal to the Ridge loss function since it also penalizes the magnitude of the intercept, though this difference could be eliminated by changing the prior distribution of the intercept.

This shows that Bayesian regression with a mean-zero Normal prior distribution is essentially equivalent to Ridge regression. Decreasing τ, just like increasing λ, increases the amount of regularization.

Full Results

Now let's actually derive the MAP by calculating the gradient of the log posterior density.

Math Note

For a symmetric matrix $\mathbf{W}$,

$$\frac{\partial}{\partial \mathbf{s}}(\mathbf{q} - \mathbf{A}\mathbf{s})^\top \mathbf{W}\,(\mathbf{q} - \mathbf{A}\mathbf{s}) = -2\mathbf{A}^\top \mathbf{W}\,(\mathbf{q} - \mathbf{A}\mathbf{s})$$

This implies that

$$\frac{\partial}{\partial \mathbf{s}}\mathbf{s}^\top\mathbf{W}\mathbf{s} = \frac{\partial}{\partial \mathbf{s}}(\mathbf{0} - I\mathbf{s})^\top\mathbf{W}(\mathbf{0} - I\mathbf{s}) = 2\mathbf{W}\mathbf{s}.$$

Using the *Math Note* above, we have

$$\begin{aligned} \log p(\hat{\boldsymbol{\beta}}|\mathbf{X}, \mathbf{y}) &= -\frac{1}{2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top\boldsymbol{\Sigma}^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) - \frac{1}{2}\boldsymbol{\beta}^\top\mathbf{T}^{-1}\boldsymbol{\beta} \\ \frac{\partial}{\partial \boldsymbol{\beta}}\log p(\boldsymbol{\beta}|\mathbf{X}, \mathbf{y}) &= \mathbf{X}^\top\boldsymbol{\Sigma}^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) - \mathbf{T}^{-1}\boldsymbol{\beta}. \end{aligned}$$

We calculate the MAP by setting this gradient equal to 0:

$$\begin{aligned} \hat{\boldsymbol{\beta}} &= \left(\mathbf{X}^\top\boldsymbol{\Sigma}^{-1}\mathbf{X} + \mathbf{T}^{-1}\right)^{-1}\mathbf{X}^\top\boldsymbol{\Sigma}^{-1}\mathbf{y} \\ &= \left(\frac{1}{\sigma^2}\mathbf{X}^\top\mathbf{X} + \frac{1}{\tau}I\right)^{-1}\frac{1}{\sigma^2}\mathbf{X}^\top\mathbf{y}. \end{aligned}$$

GLMs

Ordinary linear regression comes with several assumptions that can be relaxed with a more flexible model class: generalized linear models (GLMs). Specifically, OLS assumes

1. The target variable is a linear function of the input variables
2. The errors are Normally distributed
3. The variance of the errors is constant

When these assumptions are violated, GLMs might be the answer.

GLM Structure

A GLM consists of a *link function* and a *random component*. The *random component* identifies the distribution of the target variable y_n conditional on the input variables $\mathbf{x}_n$. For instance, we might model Y_n as a Poisson random variable where the rate parameter λ_n depends on $\mathbf{x}_n$.

The *link function* specifies how $\mathbf{x}_n$ relates to the expected value of the target variable, $\mu_n = E(Y_n)$. Let η be a linear function of the input variables, i.e. $\eta_n = \boldsymbol{\beta}^\top \mathbf{x}_n$ for some coefficients $\boldsymbol{\beta}$. We then chose a nonlinear link function to relate μ to η. For link function g we have

$$\eta_n = g(\mu_n).$$

In a GLM, we calculate η *before* calculating μ, so we often work with the inverse of g:

$$\mu_n = g^{-1}(\eta_n)$$

> **Note**
>
> Note that because η_n is a function of the data, it will vary for each observation (though the βs will not).

In total then, a GLM assumes

$$\begin{aligned} Y_n &\sim F_{\mu_n} \\ \mu_n &= g^{-1}(\eta_n) \\ \eta_n &= \boldsymbol{\beta}^\top \mathbf{x}_n, \end{aligned}$$

where F is some distribution with mean parameter μ_n.

Fitting a GLM

"Fitting" a GLM, like fitting ordinary linear regression, really consists of estimating the coefficients, $\boldsymbol{\beta}$. Once we know $\boldsymbol{\beta}$, we have η. Once we have a link function, η gives us μ through g^{-1}. A GLM can be fit in these four steps:

1. Specify the distribution of Y_n, indexed by its mean parameter μ_n.
2. Specify the link function $\eta_n = g(\mu_n)$.
3. Identify a loss function. This is typically the negative log-likelihood.
4. Find the $\hat{\boldsymbol{\beta}}$ that minimize that loss function.

In general, we can write the log-likelihood across our observations for a GLM as follows.

$$\log L\left(\{\mu_n\}_{n=1}^N; \{Y_n\}_{n=1}^N\right) = \sum_{n=1}^N \log L(\mu_n; Y_n) = \sum_{n=1}^N \log L(g^{-1}(\eta_n); Y_n) = \sum_{n=1}^N \log L(g^{-1}(\boldsymbol{\beta}^\top \mathbf{x}_n); Y_n).$$

This shows how the log-likelihood depends on $\boldsymbol{\beta}$, the parameters we want to estimate. To fit the GLM, we want to find the $\hat{\boldsymbol{\beta}}$ to maximize this log-likelihood.

Example: Poisson Regression

Step 1

Suppose we choose to model Y_n conditional on $\mathbf{x}_n$ as a Poisson random variable with rate parameter λ_n:

$$Y_n | \mathbf{x}_n \sim \text{Pois}(\lambda_n).$$

Since the expected value of a Poisson random variable is its rate parameter, $E(Y_n) = \mu_n = \lambda_n$.

Step 2

To determine the link function, let's think in terms of its inverse, $\lambda_n = g^{-1}(\eta_n)$. We know that λ_n must be non-negative and η_n could be anywhere in the reals since it is a linear function of $\mathbf{x}_n$. One function that works is

$$\lambda_n = \exp(\eta_n),$$

meaning

$$\eta_n = g(\lambda_n) = \log(\lambda_n).$$

This is the "canonical link" function for Poisson regression. More on that here.

Step 3

Let's derive the negative log-likelihood for the Poisson. Let $\boldsymbol{\lambda} = \begin{bmatrix} \lambda_1, \dots, \lambda_N \end{bmatrix}^\top$.

Math Note

The PMF for $Z \sim \text{Pois}(\lambda)$ is

$$p(z) = \frac{e^{-\lambda}\lambda^z}{z!} \propto e^{-\lambda}\lambda^z.$$

$$\begin{aligned}
L(\lambda; \{Y_n\}_{n=1}^N) &= \prod_{n=1}^N \exp(-\lambda_n)\lambda_n^{Y_n} \\
\log L(\lambda; \{Y_n\}_{n=1}^N) &= \sum_{n=1}^N Y_n \log \lambda_n - \lambda_n.
\end{aligned}$$

Now let's get our loss function, the negative log-likelihood. Recall that this should be in terms of $\boldsymbol{\beta}$ rather than λ since $\boldsymbol{\beta}$ is what we control.

$$\begin{aligned}
\mathcal{L}_N(\boldsymbol{\beta}) &= -\left(\sum_{n=1}^N Y_n \log(\exp(\eta_n)) - \exp(\eta_n)\right) \\
&= \sum_{n=1}^N (\exp(\eta_n) - Y_n\eta_n) \\
&= \sum_{n=1}^N \left(\exp(\boldsymbol{\beta}^\top \mathbf{x}_n) - Y_n\boldsymbol{\beta}^\top \mathbf{x}_n\right).
\end{aligned}$$

Step 4

We obtain $\hat{\boldsymbol{\beta}}$ by minimizing this loss function. Let's take the derivative of the loss function with respect to $\boldsymbol{\beta}$.

$$\frac{\partial \mathcal{L}_N(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \sum_{n=1}^N \left(\exp(\boldsymbol{\beta}^\top \mathbf{x}_n)\mathbf{x}_n - y_n\mathbf{x}_n\right).$$

Ideally, we would solve for $\hat{\boldsymbol{\beta}}$ by setting this gradient equal to 0. Unfortunately, there is no closed-form solution. Instead, we can approximate $\hat{\boldsymbol{\beta}}$ through gradient descent. This is done in the construction section.

Since gradient descent calculates this gradient a large number of times, it's important to calculate it efficiently. Let's see if we can clean this expression up. First recall that $\$\hat{y}_n = \hat{\lambda}_n = \exp(\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n).\$$

The loss function can then be written as

$$\frac{\partial \mathcal{L}_N(\hat{\boldsymbol{\beta}})}{\partial \hat{\boldsymbol{\beta}}} = \sum_{n=1}^N \left(\hat{y}_n\mathbf{x}_n - y_n\mathbf{x}_n\right).$$

Further, this can be written in matrix form as

$$\frac{\partial \mathcal{L}_N(\hat{\boldsymbol{\beta}})}{\partial \hat{\boldsymbol{\beta}}} = \mathbf{X}^\top(\hat{\mathbf{y}} - \mathbf{y}),$$

where $\hat{\mathbf{y}}$ is the vector of fitted values. Finally note that this vector can be calculated as

$$\hat{\mathbf{y}} = \exp(\mathbf{X}\hat{\boldsymbol{\beta}}),$$

where the exponential function is applied element-wise to each observation.

Many other GLMs exist. One important example is logistic regression, the topic of the next chapter.

Construction

This pages in this section construct classes to run the linear regression extensions discussed in the previous section. The first builds a Ridge and Lasso regression model, the second builds a Bayesian regression model, and the third builds a Poisson regression model.

Regularized Regression

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets

boston = datasets.load_boston()
X = boston['data']
y = boston['target']
```

Before building the `RegularizedRegression` class, let's define a few helper functions. The first function standardizes the data by removing the mean and dividing by the standard deviation. This is the equivalent of the `StandardScaler` from `scikit-learn`.

The `sign` function simply returns the sign of each element in an array. This is useful for calculating the gradient in Lasso regression. The `first_element_zero` option makes the function return a 0 (rather than a -1 or 1) for the first element. As discussed in the concept section, this prevents Lasso regression from penalizing the magnitude of the intercept.

```python
def standard_scaler(X):
    means = X.mean(0)
    stds = X.std(0)
    return (X - means)/stds

def sign(x, first_element_zero = False):
    signs = (-1)**(x < 0)
    if first_element_zero:
        signs[0] = 0
    return signs
```

The `RegularizedRegression` class below contains methods for fitting Ridge and Lasso regression. The first method, `record_info`, handles standardization, adds an intercept to the predictors, and records the necessary values. The second, `fit_ridge`, fits Ridge regression using

$$\hat{\beta} = \left(\mathbf{X}^\top \mathbf{X} + \lambda I'\right)^{-1} \mathbf{X}^\top \mathbf{y}.$$

The third method, `fit_lasso`, estimates the regression parameters using gradient descent. The gradient is the derivative of the Lasso loss function:

$$\frac{\partial L(\hat{\beta})}{\partial \hat{\beta}} = -\mathbf{X}^\top \left(\mathbf{y} - \mathbf{X}\hat{\beta}\right) + \lambda I' \,\text{sign}(\hat{\beta}).$$

The gradient descent used here simply adjusts the parameters a fixed number of times (determined by `n_iters`). There many more efficient ways to implement gradient descent, though we use a simple implementation here to keep focus on Lasso regression.

```python
class RegularizedRegression:

    def _record_info(self, X, y, lam, intercept, standardize):

        # standardize
        if standardize == True:
            X = standard_scaler(X)

        # add intercept
        if intercept == False:
            ones = np.ones(len(X)).reshape(len(X), 1) # column of ones
            X = np.concatenate((ones, X), axis = 1) # concatenate

        # record values
        self.X = np.array(X)
        self.y = np.array(y)
        self.N, self.D = self.X.shape
        self.lam = lam

    def fit_ridge(self, X, y, lam = 0, intercept = False, standardize = True):

        # record data and dimensions
        self._record_info(X, y, lam, intercept, standardize)

        # estimate parameters
        XtX = np.dot(self.X.T, self.X)
        I_prime = np.eye(self.D)
        I_prime[0,0] = 0
        XtX_plus_lam_inverse = np.linalg.inv(XtX + self.lam*I_prime)
        Xty = np.dot(self.X.T, self.y)
        self.beta_hats = np.dot(XtX_plus_lam_inverse, Xty)

        # get fitted values
        self.y_hat = np.dot(self.X, self.beta_hats)


    def fit_lasso(self, X, y, lam = 0, n_iters = 2000,
                  lr = 0.0001, intercept = False, standardize = True):

        # record data and dimensions
        self._record_info(X, y, lam, intercept, standardize)

        # estimate parameters
        beta_hats = np.random.randn(self.D)
        for i in range(n_iters):
            dL_dbeta = -self.X.T @ (self.y - (self.X @ beta_hats)) +
self.lam*sign(beta_hats, True)
            beta_hats -= lr*dL_dbeta
        self.beta_hats = beta_hats

        # get fitted values
        self.y_hat = np.dot(self.X, self.beta_hats)
```

The following cell runs Ridge and Lasso regression for the Boston housing dataset. For simplicity, we somewhat arbitrarily choose $\lambda = 10$—in practice, this value should be chosen through cross validation.

```python
# set lambda
lam = 10

# fit ridge
ridge_model = RegularizedRegression()
ridge_model.fit_ridge(X, y, lam)

# fit lasso
lasso_model = RegularizedRegression()
lasso_model.fit_lasso(X, y, lam)
```

The below graphic shows the coefficient estimates using Ridge and Lasso regression with a changing value of λ. Note that $\lambda = 0$ is identical to ordinary linear regression. As expected, the magnitude of the coefficient estimates decreases as λ increases.

```
Xs = ['X'+str(i + 1) for i in range(X.shape[1])]
lams = [10**4, 10**2, 0]

fig, ax = plt.subplots(nrows = 2, ncols = len(lams), figsize = (6*len(lams), 10),
sharey = True)
for i, lam in enumerate(lams):

    ridge_model = RegularizedRegression()
    ridge_model.fit_lasso(X, y, lam)
    ridge_betas = ridge_model.beta_hats[1:]
    sns.barplot(Xs, ridge_betas, ax = ax[0, i], palette = 'PuBu')
    ax[0, i].set(xlabel = 'Regressor', title = fr'Ridge Coefficients with $\lambda = $
{lam}')
    ax[0, i].set(xticks = np.arange(0, len(Xs), 2), xticklabels = Xs[::2])

    lasso_model = RegularizedRegression()
    lasso_model.fit_lasso(X, y, lam)
    lasso_betas = lasso_model.beta_hats[1:]
    sns.barplot(Xs, lasso_betas, ax = ax[1, i], palette = 'PuBu')
    ax[1, i].set(xlabel = 'Regressor', title = fr'Lasso Coefficients with $\lambda = $
{lam}')
    ax[1, i].set(xticks = np.arange(0, len(Xs), 2), xticklabels = Xs[::2])

ax[0,0].set(ylabel = 'Coefficient')
ax[1,0].set(ylabel = 'Coefficient')
plt.subplots_adjust(wspace = 0.2, hspace = 0.4)
sns.despine()
sns.set_context('talk');
```

../../_images/regularized_9_0.png

Bayesian Regression

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
boston = datasets.load_boston()
X = boston['data']
y = boston['target']
```

The `BayesianRegression` class estimates the regression coefficients using

$$\left(\frac{1}{\sigma^2}\mathbf{X}^\top\mathbf{X} + \frac{1}{\tau}I\right)^{-1}\frac{1}{\sigma^2}\mathbf{X}^\top\mathbf{y}.$$

Note that this assumes σ^2 and τ are known. We can determine the influence of the prior distribution by manipulating τ, though there are principled ways to choose τ. There are also principled Bayesian methods to model σ^2 (see here), though for simplicity we will estimate it with the typical OLS estimate:

$$\hat{\sigma}^2 = \frac{SSE}{N - (D+1)},$$

where SSE is the sum of squared errors from an ordinary linear regression, N is the number of observations, and D is the number of predictors. Using the linear regression model from chapter 1, this comes out to about 11.8.

```
class BayesianRegression:

    def fit(self, X, y, sigma_squared, tau, add_intercept = True):

        # record info
        if add_intercept:
            ones = np.ones(len(X)).reshape((len(X),1))
            X = np.append(ones, np.array(X), axis = 1)
        self.X = X
        self.y = y

        # fit
        XtX = np.dot(X.T, X)/sigma_squared
        I = np.eye(X.shape[1])/tau
        inverse = np.linalg.inv(XtX + I)
        Xty = np.dot(X.T, y)/sigma_squared
        self.beta_hats = np.dot(inverse , Xty)

        # fitted values
        self.y_hat = np.dot(X, self.beta_hats)
```

Let's fit a Bayesian regression model on the Boston housing dataset. We'll use $\sigma^2 = 11.8$ and $\tau = 10$.

```
sigma_squared = 11.8
tau = 10
model = BayesianRegression()
model.fit(X, y, sigma_squared, tau)
```

The below plot shows the estimated coefficients for varying levels of τ. A lower value of τ indicates a stronger prior, and therefore a greater pull of the coefficients towards their expected value (in this case, 0). As expected, the estimates approach 0 as τ decreases.

```
Xs = ['X'+str(i + 1) for i in range(X.shape[1])]
taus = [100, 10, 1]

fig, ax = plt.subplots(ncols = len(taus), figsize = (20, 4.5), sharey = True)
for i, tau in enumerate(taus):
    model = BayesianRegression()
    model.fit(X, y, sigma_squared, tau)
    betas = model.beta_hats[1:]
    sns.barplot(Xs, betas, ax = ax[i], palette = 'PuBu')
    ax[i].set(xlabel = 'Regressor', title = fr'Regression Coefficients with $\tau = $
{tau}')
    ax[i].set(xticks = np.arange(0, len(Xs), 2), xticklabels = Xs[::2])

ax[0].set(ylabel = 'Coefficient')
sns.set_context("talk")
sns.despine();
```

../../_images/bayesian_7_0.png

GLMs

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
boston = datasets.load_boston()
X = boston['data']
y = boston['target']
```

In this section, we'll build a class for fitting Poisson regression models. First, let's again create the `standard_scaler` function to standardize our input data.

```
def standard_scaler(X):
    means = X.mean(0)
    stds = X.std(0)
    return (X - means)/stds
```

We saw in the GLM concept page that the gradient of the loss function (the negative log-likelihood) in a Poisson model is given by

$$\frac{\partial \mathcal{L}(\hat{\boldsymbol{\beta}})}{\partial \hat{\boldsymbol{\beta}}} = \mathbf{X}^\top(\hat{\mathbf{y}} - \mathbf{y}),$$

where

$$\hat{\mathbf{y}} = \exp(\mathbf{X}\hat{\boldsymbol{\beta}}).$$

The class below constructs Poisson regression using gradient descent with these results. Again, for simplicity we use a straightforward implementation of gradient descent with a fixed number of iterations and a constant learning rate.

```python
class PoissonRegression:

    def fit(self, X, y, n_iter = 1000, lr = 0.00001, add_intercept = True, standardize
= True):

        # record stuff
        if standardize:
            X = standard_scaler(X)
        if add_intercept:
            ones = np.ones(len(X)).reshape((len(X), 1))
            X = np.append(ones, X, axis = 1)
        self.X = X
        self.y = y

        # get coefficients
        beta_hats = np.zeros(X.shape[1])
        for i in range(n_iter):
            y_hat = np.exp(np.dot(X, beta_hats))
            dLdbeta = np.dot(X.T, y_hat - y)
            beta_hats -= lr*dLdbeta

        # save coefficients and fitted values
        self.beta_hats = beta_hats
        self.y_hat = y_hat
```

Now we can fit the model on the Boston housing dataset, as below.

```python
model = PoissonRegression()
model.fit(X, y)
```

The plot below shows the observed versus fitted values for our target variable. It is worth noting that there does not appear to be a pattern of under-estimating for high target values like we saw in the ordinary linear regression example. In other words, we do not see a pattern in the residuals, suggesting Poisson regression might be a more fitting method for this problem.

```python
fig, ax = plt.subplots()
sns.scatterplot(model.y, model.y_hat)
ax.set_xlabel(r'$y$', size = 16)
ax.set_ylabel(r'$\hat{y}$', rotation = 0, size = 16, labelpad = 15)
ax.set_title(r'$y$ vs. $\hat{y}$', size = 20, pad = 10)
sns.despine()
```

../../_images/GLMs_9_0.png

Implementation

This section shows how the linear regression extensions discussed in this chapter are typically fit in Python. First let's import the Boston housing dataset.

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
boston = datasets.load_boston()
X_train = boston['data']
y_train = boston['target']
```

Regularized Regression

Both Ridge and Lasso regression can be easily fit using `scikit-learn`. A bare-bones implementation is provided below. Note that the regularization parameter `alpha` (which we called λ) is chosen arbitrarily.

```python
from sklearn.linear_model import Ridge, Lasso
alpha = 1

# Ridge
ridge_model = Ridge(alpha = alpha)
ridge_model.fit(X_train, y_train)


# Lasso
lasso_model = Lasso(alpha = alpha)
lasso_model.fit(X_train, y_train);
```

In practice, however, we want to choose `alpha` through cross validation. This is easily implemented in `scikit-learn` by designating a set of `alpha` values to try and fitting the model with `RidgeCV` or `LassoCV`.

```python
from sklearn.linear_model import RidgeCV, LassoCV
alphas = [0.01, 1, 100]

# Ridge
ridgeCV_model = RidgeCV(alphas = alphas)
ridgeCV_model.fit(X_train, y_train)

# Lasso
lassoCV_model = LassoCV(alphas = alphas)
lassoCV_model.fit(X_train, y_train);
```

We can then see which values of `alpha` performed best with the following.

```python
print('Ridge alpha:', ridgeCV.alpha_)
print('Lasso alpha:', lassoCV.alpha_)
```

```
Ridge alpha: 0.01
Lasso alpha: 1.0
```

Bayesian Regression

We can also fit Bayesian regression using `scikit-learn` (though another popular package is `pymc3`). A very straightforward implementation is provided below.

```python
from sklearn.linear_model import BayesianRidge
bayes_model = BayesianRidge()
bayes_model.fit(X_train, y_train);
```

This is not, however, identical to our construction in the previous section since it infers the σ^2 and τ parameters, rather than taking those as fixed inputs. More information can be found here. The hidden chunk below demonstrates a hacky solution for running Bayesian regression in `scikit-learn` using known values for σ^2 and τ, though it is hard to imagine a practical reason to do so

By default, Bayesian regression in `scikit-learn` treats $\alpha = \frac{1}{\sigma^2}$ and $\lambda = \frac{1}{\tau}$ as random variables and assigns them the following prior distributions

$$\begin{aligned}\alpha &\sim \text{Gamma}(\alpha_1, \alpha_2) \\ \lambda &\sim \text{Gamma}(\lambda_1, \lambda_2).\end{aligned}$$

Note that $E(\alpha) = \frac{\alpha_1}{\alpha_2}$ and $E(\lambda) = \frac{\lambda_1}{\lambda_2}$. To *fix* σ^2 and τ, we can provide an extremely strong prior on α and λ, guaranteeing that their estimates will be approximately equal to their expected value.

Suppose we want to use $\sigma^2 = 11.8$ and $\tau = 10$, or equivalently $\alpha = \frac{1}{11.8}$, $\lambda = \frac{1}{10}$. Then let

$$\begin{aligned}\alpha_1 &= 10000 \cdot \frac{1}{11.8}, \\ \alpha_2 &= 10000, \\ \lambda_1 &= 10000 \cdot \frac{1}{10}, \\ \lambda_2 &= 10000.\end{aligned}$$

This guarantees that σ^2 and τ will be approximately equal to their pre-determined values. This can be implemented in `scikit-learn` as follows

```python
big_number = 10**5

# alpha
alpha = 1/11.8
alpha_1 = big_number*alpha
alpha_2 = big_number

# lambda
lam = 1/10
lambda_1 = big_number*lam
lambda_2 = big_number

# fit
bayes_model = BayesianRidge(alpha_1 = alpha_1, alpha_2 = alpha_2, alpha_init = alpha,
                     lambda_1 = lambda_1, lambda_2 = lambda_2, lambda_init = lam)
bayes_model.fit(X_train, y_train);
```

Poisson Regression

GLMs are most commonly fit in Python through the `GLM` class from `statsmodels`. A simple Poisson regression example is given below.

As we saw in the GLM concept section, a GLM is comprised of a random distribution and a link function. We identify the random distribution through the `family` argument to `GLM` (e.g. below, we specify the `Poisson` family). The default link function depends on the random distribution. By default, the Poisson model uses the link function

$$\eta_n = g(\mu_n) = \log(\lambda_n),$$

which is what we use below. For more information on the possible distributions and link functions, check out the `statsmodels` GLM docs.

```python
import statsmodels.api as sm
X_train_with_constant = sm.add_constant(X_train)

poisson_model = sm.GLM(y_train, X_train, family=sm.families.Poisson())
poisson_model.fit();
```

Concept

A *classifier* is a supervised learning algorithm that attempts to identify an observation's membership in one of two or more groups. In other words, the target variable in classification represents a *class* from a finite set rather than a continuous number. Examples include detecting spam emails or identifying hand-written digits.

This chapter and the next cover *discriminative* and *generative* classification, respectively. Discriminative classification directly models an observation's class membership as a function of its input variables. Generative classification instead views the input variables as a function of the observation's class. It first models the prior probability that an observation belongs to a given class, then calculates the probability of observing the observation's input variables conditional on its class, and finally solves for the posterior probability of belonging to a given class using Bayes' Rule. More on that in the following chapter.

The most common method in this chapter by far is logistic regression. This is not, however, the only discriminative classifier. This chapter also introduces two others: the *Perceptron Algorithm* and *Fisher's Linear Discriminant*.

Logistic Regression

In linear regression, we modeled our target variable as a linear combination of the predictors plus a random error term. This meant that the fitted value could be any real number. Since our target in classification is not any real number, the same approach wouldn't make sense in this context. Instead, logistic regression models a *function* of the target variable as a linear combination of the predictors, then converts this function into a fitted value in the desired range.

Binary Logistic Regression

Model Structure

In the binary case, we denote our target variable with $Y_n \in \{0, 1\}$. Let $p_n = P(Y_n = 1)$ be our estimate of the probability that Y_n is in class 1. We want a way to express p_n as a function of the predictors ($\mathbf{x}_n$) that is between 0 and 1. Consider the following function, called the *log-odds* of p_n.

$$f(p_n) = \log\left(\frac{p_n}{1 - p_n}\right).$$

Note that its domain is $(0, 1)$ and its range is all real numbers. This suggests that modeling the log-odds as a linear combination of the predictors—resulting in $f(p_n) \in \mathbb{R}$—would correspond to modeling p_n as a value between 0 and 1. This is exactly what logistic regression does. Specifically, it assumes the following structure.

$$\begin{aligned} f(\hat{p}_n) = \log\left(\frac{\hat{p}_n}{1 - \hat{p}_n}\right) &= \hat{\beta}_0 + \hat{\beta}_1 x_{n1} + \cdots + \hat{\beta}_D x_{nD} \\ &= \hat{\boldsymbol{\beta}}^\top \mathbf{x}_n. \end{aligned}$$

Math Note

The *logistic function* is a common function in statistics and machine learning. The logistic function of z, written as $\sigma(z)$, is given by

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

The derivative of the logistic function is quite nice.

$$\sigma'(z) = \frac{0 + \exp(-z)}{(1 + \exp(-z))^2} = \frac{1}{1 + \exp(-z)} \cdot \frac{\exp(-z)}{1 + \exp(-z)} = \sigma(z)(1 - \sigma(z)).$$

Ultimately, we are interested in $\hat{p}_n$, not the log-odds $f(\hat{p}_n)$. Rearranging the log-odds expression, we find that $\hat{p}_n$ is the logistic function of $\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n$ (see the *Math Note* above for information on the logistic function). That is,

$$\hat{p}_n = \sigma(\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n) = \frac{1}{1 + \exp(-\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n)}.$$

By the derivative of the logistic function, this also implies that

$$\frac{\partial \hat{p}_n}{\partial \hat{\boldsymbol{\beta}}} = \frac{\partial \sigma(\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n)}{\partial \hat{\boldsymbol{\beta}}} = \sigma(\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n)\left(1 - \sigma(\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n)\right) \cdot \mathbf{x}_n$$

Parameter Estimation

We will estimate $\hat{\boldsymbol{\beta}}$ with maximum likelihood. The PMF for $Y_n \sim \text{Bern}(p_n)$ is given by

$$p(y_n) = p_n^{y_n}(1 - p_n)^{1-y_n} = \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)^{y_n}\left(1 - \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)\right)^{1-y_n}.$$

Notice that this gives us the correct probability for $y_n = 0$ and $y_n = 1$.

Now assume we observe the target variables for our training data, meaning $Y_1, \ldots, Y_n$ crystalize into $y_1, \ldots, y_n$. We can write the likelihood and log-likelihood.

$$\begin{aligned} L(\boldsymbol{\beta}; \{y_n, \mathbf{x}_n\}_{n=1}^N) &= \prod_{n=1}^N p(y_n) \\ &= \prod_{n=1}^N \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)^{y_n}\left(1 - \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)\right)^{1-y_n} \\ \log L(\boldsymbol{\beta}; \{y_n, \mathbf{x}_n\}_{n=1}^N) &= \sum_{n=1}^N y_n \log \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n) + (1 - y_n)\log\left(1 - \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)\right) \end{aligned}$$

Next, we want to find the values of $\hat{\boldsymbol{\beta}}$ that maximize this log-likelihood. Using the derivative of the logistic function for $\boldsymbol{\beta}^\top \mathbf{x}_n$ discussed above, we get

$$
\begin{aligned}
\frac{\partial \log L(\boldsymbol{\beta}; \{\mathbf{y}_n, \mathbf{x}_n\}_{n=1}^N)}{\partial \boldsymbol{\beta}} &= \sum_{n=1}^N y_n \frac{1}{\sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)} \cdot \frac{\partial \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)}{\partial \boldsymbol{\beta}} - (1-y_n)\frac{1}{1-\sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)} \cdot \frac{\partial \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)}{\partial \boldsymbol{\beta}} \\
&= \sum_{n=1}^N y_n \left(1 - \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)\right) \cdot \mathbf{x}_n - (1-y_n)\sigma(\boldsymbol{\beta}^\top \mathbf{x}_n) \cdot \mathbf{x}_n \\
&= \sum_{n=1}^N y_n \mathbf{x}_n - \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)\mathbf{x}_n \\
&= \sum_{n=1}^N (y_n - p_n)\mathbf{x}_n.
\end{aligned}
$$

Next, let $\mathbf{p} = \begin{pmatrix} p_1 & p_2 & \dots & p_N \end{pmatrix}^\top$ be the vector of probabilities. Then we can write this derivative in matrix form as

$$
\frac{\partial \log L(\boldsymbol{\beta}; \{y_n, \mathbf{x}_n\}_{n=1}^N)}{\partial \boldsymbol{\beta}} = \mathbf{X}^T(\mathbf{y} - \mathbf{p}).
$$

Ideally, we would find $\hat{\boldsymbol{\beta}}$ by setting this gradient equal to 0 and solving for $\boldsymbol{\beta}$. Unfortunately, there is no closed form solution. Instead, we can estimate $\hat{\boldsymbol{\beta}}$ through gradient descent using the derivative above. Note that gradient descent minimizes a loss function, rather than maximizing a likelihood function. To get a loss function, we would simply take the negative log-likelihood. Alternatively, we could do gradient *ascent* on the log-likelihood.

Multiclass Logistic Regression

Multiclass logistic regression generalizes the binary case into the case where there are three or more possible classes.

Notation

First, let's establish some notation. Suppose there are K classes total. When y_n can fall into three or more classes, it is best to write it as a *one-hot vector*: a vector of all zeros and a single one, with the location of the one indicating the variable's value. For instance,

$$
\mathbf{y}_n = \begin{bmatrix} 0 \\ 1 \\ \dots \\ 0 \end{bmatrix} \in \mathbb{R}^K
$$

indicates that the n^{th} observation belongs to the second of K classes. Similarly, let $\hat{\mathbf{p}}_n$ be a vector of estimated probabilities for observation n, where the j^{th} entry indicates the probability that observation n belongs to class j. Note that this vector must be non-negative and add to 1. For the example above,

$$
\hat{\mathbf{p}}_n = \begin{bmatrix} 0.01 \\ 0.98 \\ \dots \\ 0.00 \end{bmatrix} \in \mathbb{R}^K
$$

would be a pretty good estimate.

Finally, we need to write the coefficients for each class. Suppose we have D predictor variables, including the intercept (i.e. $\mathbf{x}_n \in \mathbb{R}^D$ where the first term in $\mathbf{x}_n$ is an appended 1). We can let $\hat{\boldsymbol{\beta}}_k$ be the length-D vector of coefficient estimates for class k. Alternatively, we can use the matrix

$$
\hat{\mathbf{B}} = \begin{bmatrix} \hat{\boldsymbol{\beta}}_1 & \dots & \hat{\boldsymbol{\beta}}_K \end{bmatrix} \in \mathbb{R}^{D \times K},
$$

to jointly represent the coefficients of all classes.

Model Structure

Let's start by defining $\hat{\mathbf{z}}_n$ as

$$
\hat{\mathbf{z}}_n = \hat{\mathbf{B}}^\top \mathbf{x}_n \in \mathbb{R}^K.
$$

Note that $\hat{\mathbf{z}}_n$ has one entry per class. It seems we might be able to fit $\hat{\mathbf{B}}$ such that the k^{th} element of $\hat{\mathbf{z}}_n$ gives $P(\mathbf{y}_n = k)$. However, it would be difficult to at the same time ensure the entries in $\hat{\mathbf{z}}_n$ sum to 1. Instead, we apply a *softmax* transformation to $\hat{\mathbf{z}}_n$ in order to get our estimated probabilities.

Math Note

For some length-K vector $\mathbf{z}$ and entry k, the *softmax* function is given by

$$\text{softmax}_k(\mathbf{z}) = \frac{\exp(z_k)}{\sum_{j=1}^K \exp(z_j)}.$$

Intuitively, if the k^{th} entry of $\mathbf{z}$ is large relative to the others, $\text{softmax}_k(\mathbf{z})$ will be as well.

If we drop the k from the subscript, the softmax is applied over the entire vector. I.e.,

$$\text{softmax}(\mathbf{z}) = \begin{bmatrix} \text{softmax}_1(\mathbf{z}) & \dots & \text{softmax}_K(\mathbf{z}) \end{bmatrix}^\top$$

To obtain a valid set of probability estimates for $\hat{\mathbf{p}}_n$, we apply the softmax function to $\hat{\mathbf{z}}_n$. That is,

$$\hat{\mathbf{p}}_n = \text{softmax}(\hat{\mathbf{z}}_n) = \text{softmax}(\hat{\mathbf{B}}^\top \mathbf{x}_n).$$

Let $\hat{p}_{nk}$, the k^{th} entry in $\hat{\mathbf{p}}_n$ give the probability that observation n is in class k.

Parameter Estimation

Now let's see how the estimates in $\hat{\mathbf{B}}$ are actually fit.

The Likelihood Function

As in binary logistic regression, we estimate $\hat{\mathbf{B}}$ by maximizing the (log) likelihood. Let I_{nk} be an indicator that equals 1 if observation n is in class k and 0 otherwise. The likelihood and log-likelihood are

$$\begin{aligned}
L(\mathbf{B}; \{\mathbf{y}_n, \mathbf{x}_n\}_{n=1}^N) &= \prod_{n=1}^N \prod_{k=1}^K p_{nk}^{I_{nk}} \\
\log L(\mathbf{B}; \{\mathbf{y}_n, \mathbf{x}_n\}_{n=1}^N) &= \sum_{n=1}^N \sum_{k=1}^K I_{nk} \log p_{nk} \\
&= \sum_{n=1}^N \sum_{k=1}^K I_{nk} \log\big(\text{sigmoid}_k(\mathbf{z}_n)\big) \\
&= \sum_{n=1}^N \sum_{k=1}^K I_{nk} \left(z_{nk} - \log\left(\sum_{i=1}^K \exp(z_{ni}) \right) \right),
\end{aligned}$$

where the last equality comes from the fact that

$$\log(\text{sigmoid}_k(\mathbf{z}_n)) = \log\left(\frac{\exp(z_{nk})}{\sum_{j=1}^K \exp(z_{nk})} \right) = z_{nk} - \log\left(\sum_{j=1}^K \exp(z_{nj}) \right).$$

The Derivative

Now let's look at the derivative. Specifically, let's look at the derivative of the log-likelihood with respect to the coefficients from the j^{th} class, $\boldsymbol{\beta}_j$. Note that

$$\begin{cases} \frac{\partial z_{nk}}{\partial \boldsymbol{\beta}_j} = \mathbf{x}_n, & j = k \\ \frac{\partial z_{nk}}{\partial \boldsymbol{\beta}_j} = 0, & \text{otherwise.} \end{cases}$$

This implies that

$$\frac{\partial}{\partial \boldsymbol{\beta}_j} \sum_{k=1}^K I_{nk} z_{nk} = I_{nj} \mathbf{x}_n,$$

since the derivative is automatically 0 for all terms but the j^{th} and $\mathbf{x}_n$ if $I_{nj} = 1$. Then,

$$
\begin{aligned}
\frac{\partial}{\partial \beta_j} \log L(\mathbf{B}; \{\mathbf{y}_n, \mathbf{x}_n\}_{n=1}^N) &= \sum_{n=1}^N \left(I_{nj}\mathbf{x}_n - \sum_{k=1}^K I_{nk} \frac{\exp(z_{nj})\mathbf{x}_n}{\sum_{i=1}^K \exp(z_{ni})} \right) \\
&= \sum_{n=1}^N \left(I_{nj} - \sum_{k=1}^K I_{nk} \text{softmax}_j(\mathbf{z}_n) \right) \mathbf{x}_n \\
&= \sum_{n=1}^N \left(I_{nj} - p_{nj} \sum_{k=1}^K I_{nk} \right) \mathbf{x}_n \\
&= \sum_{n=1}^N (I_{nj} - p_{nj})\mathbf{x}_n.
\end{aligned}
$$

In the last step, we drop the $\sum_{k=1}^K I_{nk}$ since this must equal 1. This gives us the gradient of the loss function with respect to a given class's coefficients, which is enough to build our model. It is possible, however, to simplify these expressions further, which is useful for gradient descent. These simplifications are given below.

Simplifying

This gradient above can also be written more compactly in matrix format. Let

$$
\mathbf{i}_j = \begin{bmatrix} I_{1j} \\ \dots \\ I_{nj} \end{bmatrix}, \quad \mathbf{p}'_j = \begin{bmatrix} p_{1j} \\ \dots \\ p_{nj} \end{bmatrix}
$$

identify whether each observation was in class j and give the probability that the observation is in class j, respectively.

Note

Note that we use $\mathbf{p}'$ rather than $\mathbf{p}$ since $\mathbf{p}_n$ was used to represent the probability that observation n belonged to a series of classes while $\mathbf{p}'_j$ refers to the probability that a series of observations belong to class j.

Then, we can write

$$
\frac{\partial}{\partial \beta_j} \log L(\mathbf{B}; \{\mathbf{y}_n, \mathbf{x}_n\}_{n=1}^N) = \mathbf{X}^\top (\mathbf{i}_j - \mathbf{p}'_j).
$$

Further, we can simultaneously represent the derivative of the loss function with respect to *each* of the class's coefficients. Let

$$
\mathbf{I} = \begin{bmatrix} \mathbf{i}_1 & \dots & \mathbf{i}_K \end{bmatrix} \in \mathbb{R}^{N \times K}, \quad \mathbf{P} = \begin{bmatrix} \mathbf{p}'_1 & \dots & \mathbf{p}'_K \end{bmatrix} \in \mathbb{R}^{N \times K}.
$$

We can then write

$$
\frac{\partial}{\partial \mathbf{B}} \log L(\mathbf{B}; \{\mathbf{y}_n, \mathbf{x}_n\}_{n=1}^N) = \mathbf{X}^\top (\mathbf{I} - \mathbf{P}) \in \mathbb{R}^{D \times K}.
$$

Finally, we can also write $\hat{\mathbf{P}}$ (the estimate of $\mathbf{P}$) as a matrix product, which will make calculations more efficient. Let

$$
\hat{\mathbf{Z}} = \mathbf{X}\hat{\mathbf{B}} \in \mathbb{R}^{N \times K},
$$

where the n^{th} row is equal to $\hat{\mathbf{z}}_n$. Then,

$$
\hat{\mathbf{P}} = \text{softmax}(\hat{\mathbf{Z}}) \in \mathbb{R}^{N \times K},
$$

where the softmax function is applied to each row.

The Perceptron Algorithm

The *perceptron algorithm* is a simple classification method that plays an important historical role in the development of the much more flexible neural network. The perceptron is a *linear binary classifier*—*linear* since it separates the input variable space linearly and *binary* since it places observations into one of two classes.

Model Structure

It is most convenient to represent our binary target variable as $y_n \in \{-1, +1\}$. For example, an email might be marked as $+1$ if it is spam and -1 otherwise. As usual, suppose we have one or more predictors per observation. We obtain our feature vector $\mathbf{x}_n$ by concatenating a leading 1 to this collection of predictors.

Consider the following function, which is an example of an *activation function*:

$$f(x) = \begin{cases} +1, & x \geq 0 \\ -1, & x < 0. \end{cases}$$

The perceptron applies this activation function to a linear combination of $\mathbf{x}_n$ in order to return a fitted value. That is,

$$\hat{y}_n = f(\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n).$$

In words, the perceptron predicts $+1$ if $\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n \geq 0$ and -1 otherwise. Simple enough!

Note that an observation is correctly classified if $y_n \hat{y}_n = 1$ and misclassified if $y_n \hat{y}_n = -1$. Then let $\mathcal{M}$ be the set of misclassified observations, i.e. all $n \in \{1, \dots, N\}$ for which $y_n \hat{y}_n = -1$.

Parameter Estimation

As usual, we calculate the $\hat{\boldsymbol{\beta}}$ as the set of coefficients to minimize some loss function. Specifically, the perceptron attempts to minimize the *perceptron criterion*, defined as

$$\mathcal{L}(\hat{\boldsymbol{\beta}}) = -\sum_{n \in \mathcal{M}} y_n(\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n).$$

The perceptron criterion does not penalize correctly classified observations but penalizes misclassified observations based on the magnitude of $\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n$—that is, how wrong they were.

The gradient of the perceptron criterion is

$$\frac{\partial \mathcal{L}(\hat{\boldsymbol{\beta}})}{\partial \hat{\boldsymbol{\beta}}} = -\sum_{n \in \mathcal{M}} y_n \mathbf{x}_n.$$

We obviously can't set this equal to 0 and solve for $\hat{\boldsymbol{\beta}}$, so we have to estimate $\hat{\boldsymbol{\beta}}$ through gradient descent. Specifically, we could use the following procedure, where η is the learning rate.

1. Randomly instantiate $\hat{\boldsymbol{\beta}}$
2. Until convergence or some stopping rule is reached:
    1. For $n \in \{1, \dots, N\}$:
        1. $\hat{y}_n = f(\hat{\boldsymbol{\beta}}^\top \mathbf{x}_n)$
        2. If $y_n \hat{y}_n = -1, \quad \hat{\boldsymbol{\beta}} \leftarrow \hat{\boldsymbol{\beta}} + \eta \cdot y_n \mathbf{x}_n$.

It can be shown that convergence is guaranteed in the linearly separable case but not otherwise. If the classes are not linearly separable, some stopping rule will have to be determined.

Fisher's Linear Discriminant

Intuitively, a good classifier is one that bunches together observations in the same class and separates observations between classes. *Fisher's linear discriminant* attempts to do this through dimensionality reduction. Specifically, it projects data points onto a single dimension and classifies them according to their location along this dimension. As we will see, its goal is to find the projection that that maximizes the ratio of between-class variation to within-class variation. Fisher's linear discriminant can be applied to multiclass tasks, but we'll only review the binary case here.

Model Structure

As usual, suppose we have a vector of one or more predictors per observation, $\mathbf{x}_n$. However we do *not* append a 1 to this vector. I.e., there is no bias term built into the vector of predictors. Then, we can project $\mathbf{x}_n$ to one dimension with

$$f(\mathbf{x}_n) = \boldsymbol{\beta}^\top \mathbf{x}_n.$$

Once we've chosen our $\boldsymbol{\beta}$, we can classify observation n according to whether $f(\mathbf{x}_n)$ is greater than some cutoff value. For instance, consider the data on the left below. Given the vector $\boldsymbol{\beta}^\top = \begin{bmatrix} 1 & -1 \end{bmatrix}$ (shown in red), we could classify observations as dark blue if $\boldsymbol{\beta}^\top \mathbf{x}_n \geq 2$ and light blue otherwise. The image on the right shows the projections using $\boldsymbol{\beta}$. Using the cutoff $\boldsymbol{\beta}^\top \mathbf{x}_n \geq 2$, we see that most cases are correctly classified though some are misclassified. We can improve the model in two ways: either changing $\boldsymbol{\beta}$ or changing the cutoff.

download-2

In practice, the linear discriminant will tell us $\boldsymbol{\beta}$ but won't tell us the cutoff value. Instead, the discriminant will rank the $f(\mathbf{x}_n)$ so that the classes are separated as much as possible. It is up to us to choose the cutoff value.

Fisher Criterion

The *Fisher criterion* quantifies how well a parameter vector $\boldsymbol{\beta}$ classifies observations by rewarding between-class variation and penalizing within-class variation. The only variation it considers, however, is in the single dimension we project along. For each observation, we have

$$f(\mathbf{x}_n) = \boldsymbol{\beta}^\top \mathbf{x}_n.$$

Let N_k be the number of observations and $\mathcal{S}_k$ be the set of observations in class k for $k \in \{0, 1\}$. Then let

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n \in \mathcal{S}_k} \mathbf{x}_n$$

be the mean vector (also known as the *centroid*) of the predictors in class k. This class-mean is also projected along our single dimension with

$$\mu_k = \boldsymbol{\beta}^\top \boldsymbol{\mu}_k.$$

A simple way to measure how well $\boldsymbol{\beta}$ separates classes is with the magnitude of the difference between μ_2 and μ_1. To assess similarity *within* a class, we use

$$\sigma_k^2 = \sum_{n \in \mathcal{S}_k} (f(\mathbf{x}_n) - \mu_k)^2,$$

the within-class sum of squared differences between the projections of the observations and the projection of the class-mean. We are then ready to introduce the Fisher criterion:

$$F(\boldsymbol{\beta}) = \frac{(\mu_2 - \mu_1)^2}{\sigma_1^2 + \sigma_2^2}.$$

Intuitively, an increase in $F(\boldsymbol{\beta})$ implies the between-class variation has increased relative to the within-class variation.

Let's write $F(\boldsymbol{\beta})$ as an explicit function of $\boldsymbol{\beta}$. Starting with the numerator, we have

$$\begin{aligned}
(\mu_2 - \mu_1)^2 &= \left(\boldsymbol{\beta}^\top (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\right)^2 \\
&= \left(\boldsymbol{\beta}^\top (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\right) \cdot \left(\boldsymbol{\beta}^\top (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\right) \\
&= \boldsymbol{\beta}^\top (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^\top \boldsymbol{\beta} \\
&= \boldsymbol{\beta}^\top \boldsymbol{\Sigma}_b \boldsymbol{\beta},
\end{aligned}$$

where $\boldsymbol{\Sigma}_b = (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^\top$ is the *between class* covariance matrix. Then for the denominator, we have

$$\begin{aligned}
\sigma_1^2 + \sigma_2^2 &= \sum_{n \in \mathcal{S}_1} \left(\boldsymbol{\beta}^\top (\mathbf{x}_n - \boldsymbol{\mu}_1)\right)^2 + \sum_{n \in \mathcal{S}_2} \left(\boldsymbol{\beta}^\top (\mathbf{x}_n - \boldsymbol{\mu}_2)\right)^2 \\
&= \sum_{n \in \mathcal{S}_1} \boldsymbol{\beta}^\top (\mathbf{x}_n - \boldsymbol{\mu}_1)(\mathbf{x}_n - \boldsymbol{\mu}_1)^\top \boldsymbol{\beta} + \sum_{n \in \mathcal{S}_2} \boldsymbol{\beta}^\top (\mathbf{x}_n - \boldsymbol{\mu}_2)(\mathbf{x}_n - \boldsymbol{\mu}_2)^\top \boldsymbol{\beta} \\
&= \boldsymbol{\beta}^\top \left(\sum_{n \in \mathcal{S}_1} (\mathbf{x}_n - \boldsymbol{\mu}_1)(\mathbf{x}_n - \boldsymbol{\mu}_1)^\top + \sum_{n \in \mathcal{S}_2} (\mathbf{x}_n - \boldsymbol{\mu}_2)(\mathbf{x}_n - \boldsymbol{\mu}_2)^\top \right) \boldsymbol{\beta} \\
&= \boldsymbol{\beta}^\top \boldsymbol{\Sigma}_w \boldsymbol{\beta},
\end{aligned}$$

where $\boldsymbol{\Sigma}_w = \sum_{n \in \mathcal{S}_1} (\mathbf{x}_n - \boldsymbol{\mu}_1)(\mathbf{x}_n - \boldsymbol{\mu}_1)^\top + \sum_{n \in \mathcal{S}_2} (\mathbf{x}_n - \boldsymbol{\mu}_2)(\mathbf{x}_n - \boldsymbol{\mu}_2)^\top$ is the *within class* covariance matrix. In total, then, we have

$$F(\boldsymbol{\beta}) = \frac{\boldsymbol{\beta}^\top \boldsymbol{\Sigma}_b \boldsymbol{\beta}}{\boldsymbol{\beta}^\top \boldsymbol{\Sigma}_w \boldsymbol{\beta}}.$$

Parameter Estimation

Finally, we can find the $\boldsymbol{\beta}$ to optimize $F(\boldsymbol{\beta})$. Importantly, note that the magnitude of $\boldsymbol{\beta}$ is unimportant since we simply want to rank the $f(\mathbf{x}_n) = \boldsymbol{\beta}^\top \mathbf{x}_n$ values and using a vector proportional to $\boldsymbol{\beta}$ will not change this ranking.

> **Math Note**
>
> For a symmetric matrix $\mathbf{W}$ and a vector $\mathbf{s}$, we have
>
> $$\frac{\partial \mathbf{s}^\top \mathbf{W} \mathbf{s}}{\partial \mathbf{s}} = 2\mathbf{W}\mathbf{s}.$$
>
> Notice that $\boldsymbol{\Sigma}_w$ is symmetric since its $(i, j)^{\text{th}}$ element is
>
> $$\sum_{n \in S_1} (x_{ni} - \mu_{1i})(x_{nj} - \mu_{1j}) + \sum_{n \in S_2} (x_{ni} - \mu_{2i})(x_{nj} - \mu_{2j}),$$
>
> which is equivalent to its $(j, i)^{\text{th}}$ element.

By the quotient rule and the math note above,

$$\frac{\partial F(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \frac{2\boldsymbol{\Sigma}_b \boldsymbol{\beta}\left(\boldsymbol{\beta}^\top \boldsymbol{\Sigma}_w \boldsymbol{\beta}\right) - 2\boldsymbol{\Sigma}_w \boldsymbol{\beta}\left(\boldsymbol{\beta}^\top \boldsymbol{\Sigma}_b \boldsymbol{\beta}\right)}{(\boldsymbol{\beta}^\top \boldsymbol{\Sigma}_w \boldsymbol{\beta})^2}.$$

We then set this equal to 0. Note that the denominator is just a scalar, so it goes away.

$$\begin{aligned} \mathbf{0} &= \boldsymbol{\Sigma}_b \boldsymbol{\beta}\left(\boldsymbol{\beta}^\top \boldsymbol{\Sigma}_w \boldsymbol{\beta}\right) - \boldsymbol{\Sigma}_w \boldsymbol{\beta}\left(\boldsymbol{\beta}^\top \boldsymbol{\Sigma}_b \boldsymbol{\beta}\right) \\ \boldsymbol{\Sigma}_w \boldsymbol{\beta}\left(\boldsymbol{\beta}^\top \boldsymbol{\Sigma}_b \boldsymbol{\beta}\right) &= \boldsymbol{\Sigma}_b \boldsymbol{\beta}\left(\boldsymbol{\beta}^\top \boldsymbol{\Sigma}_w \boldsymbol{\beta}\right). \end{aligned}$$

Since we only care about the direction of $\boldsymbol{\beta}$ and not its magnitude, we can make some simplifications. First, we can ignore $\boldsymbol{\beta}^\top \boldsymbol{\Sigma}_b \boldsymbol{\beta}$ and $\boldsymbol{\beta}^\top \boldsymbol{\Sigma}_b \boldsymbol{\beta}$ since they are just constants. Second, we can note that $\boldsymbol{\Sigma}_b \boldsymbol{\beta}$ is proportional to $\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1$, as shown below:

$$\boldsymbol{\Sigma}_b \boldsymbol{\beta} = (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^\top \boldsymbol{\beta} = (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)k \propto (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1),$$

where k is some constant. Therefore, our solution becomes

$$\hat{\boldsymbol{\beta}} \propto \boldsymbol{\Sigma}_w^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1).$$

The image below on the left shows the $\hat{\boldsymbol{\beta}}$ (in red) found by Fisher's linear discriminant. On the right, we again see the projections of these datapoints from $\hat{\boldsymbol{\beta}}$. The cutoff is chosen to be around 0.05. Note that this discriminator, unlike the one above, successfully separates the two classes!

Construction

In this section, we construct the three classifiers covered in the previous section. Binary and multiclass logistic regression are covered first, followed by the perceptron algorithm, and finally Fisher's linear discriminant.

Logistic Regression

```
import numpy as np
np.set_printoptions(suppress=True)
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
```

In this section we will construct binary and multiclass logistic regression models. We will try our binary model on the breast cancer dataset and the multiclass model on the wine dataset.

Binary Logistic Regression

```
# import data
cancer = datasets.load_breast_cancer()
X = cancer['data']
y = cancer['target']
```

Let's first define some helper functions: the logistic function and a standardization function, equivalent to `scikit-learn`'s `StandardScaler`

```
def logistic(z):
    return (1 + np.exp(-z))**(-1)

def standard_scaler(X):
    mean = X.mean(0)
    sd = X.std(0)
    return (X - mean)/sd
```

The binary logistic regression class is defined below. First, it (optionally) standardizes and adds an intercept term. Then it estimates $\boldsymbol{\beta}$ with gradient descent, using the gradient of the negative log-likelihood derived in the concept section,

$$\frac{\partial \mathcal{L}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \frac{\partial - \log L(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\mathbf{X}^\top(\mathbf{y} - \mathbf{p}).$$

```
class BinaryLogisticRegression:

    def fit(self, X, y, n_iter, lr, standardize = True, has_intercept = False):

        ### Record Info ###
        if standardize:
            X = standard_scaler(X)
        if not has_intercept:
            ones = np.ones(X.shape[0]).reshape(-1, 1)
            X = np.concatenate((ones, X), axis = 1)
        self.X = X
        self.N, self.D = X.shape
        self.y = y
        self.n_iter = n_iter
        self.lr = lr

        ### Calculate Beta ###
        beta = np.random.randn(self.D)
        for i in range(n_iter):
            p = logistic(np.dot(self.X, beta)) # vector of probabilities
            gradient = -np.dot(self.X.T, (self.y-p)) # gradient
            beta -= self.lr*gradient

        ### Return Values ###
        self.beta = beta
        self.p = logistic(np.dot(self.X, self.beta))
        self.yhat = self.p.round()
```

The following instantiates and fits our logistic regression model, then assesses the in-sample accuracy. Note here that we predict observations to be from class 1 if we estimate $P(Y_n = 1)$ to be above 0.5, though this is not required.

```
binary_model = BinaryLogisticRegression()
binary_model.fit(X, y, n_iter = 10**4, lr = 0.0001)
print('In-sample accuracy: '  + str(np.mean(binary_model.yhat == binary_model.y)))
```

```
In-sample accuracy: 0.9894551845342706
```

Finally, the graph below shows a distribution of the *estimated* $P(Y_n = 1)$ based on each observation's *true* class. This demonstrates that our model is quite confident of its predictions.

```
fig, ax = plt.subplots()
sns.distplot(binary_model.p[binary_model.yhat == 0], kde = False, bins = 8, label =
'Class 0', color = 'cornflowerblue')
sns.distplot(binary_model.p[binary_model.yhat == 1], kde = False, bins = 8, label =
'Class 1', color = 'darkblue')
ax.legend(loc = 9, bbox_to_anchor = (0,0,1.59,.9))
ax.set_xlabel(r'Estimated $P(Y_n = 1)$', size = 14)
ax.set_title(r'Estimated $P(Y_n = 1)$ by True Class', size = 16)
sns.despine()
```

../../_images/logistic_regression_12_0.png

Multiclass Logistic Regression

```
# import data
wine = datasets.load_wine()
X = wine['data']
y = wine['target']
```

Before fitting our multiclass logistic regression model, let's again define some helper functions. The first (which we don't actually use) shows a simple implementation of the softmax function. The second applies the softmax function to each row of a matrix. An example of this is shown for the matrix

$$\mathbf{Z} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

The third function returns the I matrix discussed in the concept section, whose $(n, k)^{\text{th}}$ element is a 1 if the n^{th} observation belongs to the k^{th} class and a 0 otherwise. An example is shown for

$$\mathbf{y} = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 \end{bmatrix}^\top.$$

```
def softmax(z):
    return np.exp(z)/(np.exp(z).sum())

def softmax_byrow(Z):
    return (np.exp(Z)/(np.exp(Z).sum(1)[:,None]))

def make_I_matrix(y):
    I = np.zeros(shape = (len(y), len(np.unique(y))), dtype = int)
    for j, target in enumerate(np.unique(y)):
        I[:,j] = (y == target)
    return I


Z_test = np.array([[1, 1],
              [0,1]])
print('Softmax for Z:\n', softmax_byrow(Z_test).round(2))

y_test = np.array([0,0,1,1,2])
print('I matrix of [0,0,1,1,2]:\n', make_I_matrix(y_test), end = '\n\n')
```

```
Softmax for Z:
 [[0.5  0.5 ]
 [0.27 0.73]]
I matrix of [0,0,1,1,2]:
 [[1 0 0]
 [1 0 0]
 [0 1 0]
 [0 1 0]
 [0 0 1]]
```

The multiclass logistic regression model is constructed below. After standardizing and adding an intercept, we estimate $\hat{\mathbf{B}}$ through gradient descent. Again, we use the gradient discussed in the concept section,

$$\frac{\partial \mathcal{L}(\mathbf{B})}{\partial \mathbf{B}} = \frac{\partial - \log L(\mathbf{B})}{\partial \mathbf{B}} = \mathbf{X}^\top(\mathbf{I} - \mathbf{P}).$$

```
class MulticlassLogisticRegression:

    def fit(self, X, y, n_iter, lr, standardize = True, has_intercept = False):

        ### Record Info ###
        if standardize:
            X = standard_scaler(X)
        if not has_intercept:
            ones = np.ones(X.shape[0]).reshape(-1, 1)
            X = np.concatenate((ones, X), axis = 1)
        self.X = X
        self.N, self.D = X.shape
        self.y = y
        self.K = len(np.unique(y))
        self.n_iter = n_iter
        self.lr = lr

        ### Fit B ###
        B = np.random.randn(self.D*self.K).reshape((self.D, self.K))
        self.I = make_I_matrix(self.y)
        for i in range(n_iter):
            Z = np.dot(self.X, B)
            P = softmax_byrow(Z)
            gradient = np.dot(self.X.T, self.I - P)
            B += lr*gradient

        ### Return Values ###
        self.B = B
        self.Z = np.dot(self.X, B)
        self.P = softmax_byrow(self.Z)
        self.yhat = self.P.argmax(1)
```

The multiclass model is instantiated and fit below. The `yhat` value returns the class with the greatest estimated probability. We are again able to classify all observations correctly.

```
multiclass_model = MulticlassLogisticRegression()
multiclass_model.fit(X, y, 10**4, 0.0001)
print('In-sample accuracy: '  + str(np.mean(multiclass_model.yhat == y)))
```

```
In-sample accuracy: 1.0
```

The plots show the distribution of our estimates of the probability that each observation belongs to the class it actually belongs to. E.g. for observations of class 1, we plot $P(y_n = 1)$. The fact that most counts are close to 1 shows that again our model is confident in its predictions.

```
fig, ax = plt.subplots(1, 3, figsize = (17, 5))
for i, y in enumerate(np.unique(y)):
    sns.distplot(multiclass_model.P[multiclass_model.y == y, i],
                 hist_kws=dict(edgecolor="darkblue"),
                 color = 'cornflowerblue',
                 bins = 15,
                 kde = False,
                 ax = ax[i]);
    ax[i].set_xlabel(xlabel = fr'$P(y = {y})$', size = 14)
    ax[i].set_title('Histogram for Observations in Class '+ str(y), size = 16)
sns.despine()
```

../../_images/logistic_regression_22_0.png

The Perceptron Algorithm

```
import numpy as np
np.set_printoptions(suppress=True)
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets

# import data
cancer = datasets.load_breast_cancer()
X = cancer['data']
y = cancer['target']
```

Before constructing the perceptron, let's define a few helper functions. The `sign` function returns `1` for positive numbers and `-1` for non-positive numbers, which will be useful since the perceptron classifies according to

$$\text{sign}(\boldsymbol{\beta}^\top \mathbf{x}_n).$$

Next, the `to_binary` function can be used to convert predictions in $\{-1, +1\}$ to their equivalents in $\{0, 1\}$, which is useful since the perceptron algorithm uses the former though binary data is typically stored as the latter. Finally, the `standard_scaler` standardizes our features, similar to `scikit-learn`'s `StandardScaler`.

Note

Note that we don't actually need to use the `sign` function. Instead, we could deem an observation correctly classified if $y_n \hat{y}_n \geq 0$ and misclassified otherwise. We use it here to be consistent with the derivation in the content section.

```
def sign(a):
    return (-1)**(a < 0)

def to_binary(y):
        return y > 0

def standard_scaler(X):
    mean = X.mean(0)
    sd = X.std(0)
    return (X - mean)/sd
```

The perceptron is implemented below. As usual, we optionally standardize and add an intercept term. Then we fit $\hat{\beta}$ with the algorithm introduced in the concept section.

This implementation tracks whether the perceptron has converged (i.e. all training algorithms are fitted correctly) and stops fitting if so. If not, it will run until `n_iters` is reached.

```
class Perceptron:

    def fit(self, X, y, n_iter = 10**3, lr = 0.001, add_intercept = True, standardize = 
True):

        # Add Info #
        if standardize:
            X = standard_scaler(X)
        if add_intercept:
            ones = np.ones(len(X)).reshape(-1, 1)
        self.X = X
        self.N, self.D = self.X.shape
        self.y = y
        self.n_iter = n_iter
        self.lr = lr
        self.converged = False

        # Fit #
        beta = np.random.randn(self.D)/5
        for i in range(int(self.n_iter)):

            # Form predictions
            yhat = to_binary(sign(np.dot(self.X, beta)))

            # Check for convergence
            if np.all(yhat == sign(self.y)):
                self.converged = True
                self.iterations_until_convergence = i
                break

            # Otherwise, adjust
            for n in range(self.N):
                yhat_n = sign(np.dot(beta, self.X[n]))
                if (self.y[n]*yhat_n == -1):
                    beta += self.lr * self.y[n]*self.X[n]

        # Return Values #
        self.beta = beta
        self.yhat = to_binary(sign(np.dot(self.X, self.beta)))
```

Now we can fit the model. We'll again use the breast cancer dataset from `sklearn.datasets`. We can also check whether the perceptron converged and, if so, after how many iterations.

```
perceptron = Perceptron()
perceptron.fit(X, y, n_iter = 1e3, lr = 0.01)
```

```
if perceptron.converged:
    print(f"Converged after {perceptron.iterations_until_convergence} iterations")
else:
    print("Not converged")
```

```
Not converged
```

```
np.mean(perceptron.yhat == perceptron.y)
```

```
0.9736379613356766
```

Fisher's Linear Discriminant

```
import numpy as np
np.set_printoptions(suppress=True)
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
```

Since it is largely geometric, the Linear Discriminant won't look like other methods we've seen (no gradients!). First we take the class means and calculate $\mathbf{\Sigma}_w$ as described in the concept section. Estimating $\hat{\boldsymbol{\beta}}$ is then as simple as calculating $\mathbf{\Sigma}_w^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)$. Let's demonstrate the model using the breast cancer dataset.

```
# import data
cancer = datasets.load_breast_cancer()
X = cancer['data']
y = cancer['target']
```

Note

Note that the value we return for each observation, $f(\mathbf{x}_n)$, is not a fitted value. Instead, we look at the distribution of the $f(\mathbf{x}_n)$ by class and determine a classification rule based on this distribution. For instance, we might predict observation n came from class 1 if $f(\mathbf{x}_n) \geq c$ (for some constant c) and from class 0 otherwise.

```
class FisherLinearDiscriminant:

    def fit(self, X, y):
        ## Save stuff
        self.X = X
        self.y = y
        self.N, self.D = self.X.shape

        ## Calculate class means
        X0 = X[y == 0]
        X1 = X[y == 1]
        mu0 = X0.mean(0)
        mu1 = X1.mean(0)

        ## Sigma_w
        Sigma_w = np.empty((self.D, self.D))
        for x0 in X0:
            x0_minus_mu0 = (x0 - mu0).reshape(-1, 1)
            Sigma_w += np.dot(x0_minus_mu0, x0_minus_mu0.T)
        for x1 in X1:
            x1_minus_mu1 = (x1 - mu1).reshape(-1, 1)
            Sigma_w += np.dot(x1_minus_mu1, x1_minus_mu1.T)
        Sigma_w_inverse = np.linalg.inv(Sigma_w)

        ## Beta
        self.beta = np.dot(Sigma_w_inverse, mu1 - mu0)
        self.f = np.dot(X, self.beta)
```

We can now fit the model on the breast cancer dataset, as below.

```
model = FisherLinearDiscriminant()
model.fit(X, y);
```

Once we have fit the model, we can look at the distribution of $f(\mathbf{x}_n)$ by class. We hope to see a significant separation between classes and a significant clustering within classes. The histogram below shows that we've nearly separated the two classes and the two classes are decently clustered. We would presumably choose a cutoff somewhere between $f(x) = -.09$ and $f(x) = -.08$.

```
fig, ax = plt.subplots(figsize = (7,5))
sns.distplot(model.f[model.y == 0], bins = 25, kde = False,
             color = 'cornflowerblue', label = 'Class 0')
sns.distplot(model.f[model.y == 1], bins = 25, kde = False,
             color = 'darkblue', label = 'Class 1')
ax.set_xlabel(r"$f\hspace{.25}(x_n)$", size = 14)
ax.set_title(r"Histogram of $f\hspace{.25}(x_n)$ by Class", size = 16)
ax.legend()
sns.despine()
```

../../_images/fisher_discriminant_9_0.png

Implementation

This section will demonstrate how to fit the discriminative classifiers discussed in this chapter with `scikit-learn`. Note that other libraries are frequently used—e.g. `statsmodels` for logistic regresssion and `tensorflow` for the perceptron.

For binary tasks, we'll be using the breast cancer dataset and for multiclass tasks, we'll be using the wine dataset.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets

# import data
cancer = datasets.load_breast_cancer()
X_cancer = cancer['data']
y_cancer = cancer['target']
wine = datasets.load_wine()
X_wine = wine['data']
y_wine = wine['target']
```

Logistic Regression

Binary Logistic Regression

A standard `scikit-learn` implementation of binary logistic regression is shown below. Note the two arguments set when instantiating the model: `C` is a regularization term where a higher `C` indicates *less* penalty on the magnitude of the coefficients and `max_iter` determines the maximum number of iterations the solver will use. We set `C` to be arbitrarily high such that there is effectively no regulariation and `max_iter` to be 1,000, which is enough for this model to converge.

```
from sklearn.linear_model import LogisticRegression
binary_model = LogisticRegression(C = 10**5, max_iter = 1e5)
binary_model.fit(X_cancer, y_cancer)
```

```
LogisticRegression(C=100000, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100000.0,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

`scikit-learn`'s logistic regression model can return two forms of predictions: the predicted classes or the predicted probabilities. The `.predict()` method predicts an observation for each class while `.predict_proba()` gives the probability for all classes included in the training set (in this case, just 0 and 1).

```
y_hats = binary_model.predict(X_cancer)
p_hats = binary_model.predict_proba(X_cancer)
print(f'Training accuracy: {binary_model.score(X_cancer, y_cancer)}')
```

```
Training accuracy: 0.984182776801406
```

Multiclass Logistic Regression

Multiclass logistic regression can be fit in `scikit-learn` as below. In fact, no arguments need to be changed in order to fit a multiclass model versus a binary one. However, the implementation below adds one new argument. Setting `multiclass` equal to 'multinomial' tells the model explicitly to follow the algorithm introduced in the concept section. This will be done by default for non-binary problems unless the `solver` is set to 'liblinear'. In that case, it will fit a "one-versus-rest" model.

```
from sklearn.linear_model import LogisticRegression
multiclass_model = LogisticRegression(multi_class = 'multinomial', C = 10**5, max_iter
= 10**4)
multiclass_model.fit(X_wine, y_wine)
```

```
LogisticRegression(C=100000, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=10000,
                   multi_class='multinomial', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

Again, we can see the predicted classes and predicted probabilities for each class, as below.

```
y_hats = multiclass_model.predict(X_wine)
p_hats = multiclass_model.predict_proba(X_wine)
print(f'Training accuracy: {multiclass_model.score(X_wine, y_wine)}')
```

```
Training accuracy: 1.0
```

The Perceptron Algorithm

The perceptron algorithm is implemented below. This algorithm is rarely used in practice but serves as an important part of neural networks, the topic of Chapter 7.

```
from sklearn.linear_model import Perceptron
perceptron = Perceptron()
perceptron.fit(X_cancer, y_cancer);
```

Fisher's Linear Discriminant

Finally, we fit Fisher's Linear Discriminant with the `LinearDiscriminantAnalysis` class from `scikit-learn`. This class can also be viewed as a generative model, which is discussed in the next chapter, but the implementation below reduces to the discriminative classifier derived in the concept section. Specifying `n_components = 1` tells the model to reduce the data to one dimension. This is the equivalent of generating the

$$f(\mathbf{x}_n) = \boldsymbol{\beta}^\top \mathbf{x}_n$$

transformations that we saw in the concept section. We can then see if the two classes are separated by checking that either 1) $f(\mathbf{x}_n) < f(\mathbf{x}_m)$ for all n in class 0 and m in class 1 or 2) $f(\mathbf{x}_n) > f(\mathbf{x}_m)$ for all n in class 0 and m in class 1. Equivalently, we can see that the two classes are not separated in the histogram below.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis(n_components = 1)
lda.fit(X_cancer, y_cancer);

f0 = np.dot(X_cancer, lda.coef_[0])[y_cancer == 0]
f1 = np.dot(X_cancer, lda.coef_[0])[y_cancer == 1]
print('Separated:', (min(f0) > max(f1)) | (max(f0) < min(f1)))
```

```
Separated: False
```

```python
fig, ax = plt.subplots(figsize = (7,5))
sns.distplot(f0, bins = 25, kde = False,
             color = 'cornflowerblue', label = 'Class 0')
sns.distplot(f1, bins = 25, kde = False,
             color = 'darkblue', label = 'Class 1')
ax.set_xlabel(r"$f\hspace{.25}(x_n)$", size = 14)
ax.set_title(r"Histogram of $f\hspace{.25}(x_n)$ by Class", size = 16)
ax.legend()
sns.despine()
```

../../_images/code_20_0.png

Concept

Discriminative classifiers, as we saw in the previous chapter, model a target variable as a direct function of one or more predictors. Generative classifiers, the subject of this chapter, instead view the predictors as being generated according to their class—i.e., they see the predictors as a function of the target, rather than the other way around. They then use Bayes' rule to turn $P(\mathbf{x}_n|Y_n = k)$ into $P(Y_n = k|\mathbf{x}_n)$.

In generative classifiers, we view both the target and the predictors as random variables. We will therefore refer to the target variable with Y_n, but in order to avoid confusing it with a matrix, we refer to the predictor vector with $\mathbf{x}_n$.

Generative models can be broken down into the three following steps. Suppose we have a classification task with K unordered classes, represented by $k = 1, \dots, K$.

1. Estimate the density of the predictors conditional on the target belonging to each class. I.e., estimate $p(\mathbf{x}_n|Y_n = k)$ for $k = 1, \dots, K$.
2. Estimate the prior probability that a target belongs to any given class. I.e., estimate $P(Y_n = k)$ for $k = 1, \dots, K$. This is also written as $p(Y_n)$.
3. Using Bayes' rule, calculate the posterior probability that the target belongs to any given class. I.e., calculate $p(Y_n = k|\mathbf{x}_n) \propto p(\mathbf{x}_n|Y_n = k)p(Y_n = k)$ for $k = 1, \dots, K$.

We then classify observation n as being from the class for which $P(Y_n = k|\mathbf{x}_n)$ is greatest. In math,

$$\hat{Y}_n = \underset{k}{\text{arg max}}\ p(Y_n = k|\mathbf{x}_n).$$

Note that we do not need $p(\mathbf{x}_n)$, which would be the denominator in the Bayes' rule formula, since it would be equal across classes.

Note

This chapter is oriented differently from the others. The main methods discussed—Linear Discriminant Analysis, Quadratic Discriminant Analysis, and Naive Bayes—share much of the same structure. Rather than introducing each individually, we describe them together and note (in section 2.2) how they differ.

1. Model Structure

A generative classifier models two sources of randomness. First, we assume that out of the K possible classes, each observation belongs to class k independently with probability π_k. In other words, letting $\boldsymbol{\pi} = \begin{bmatrix} \pi_1 & \dots & \pi_K \end{bmatrix}^\top \in \mathbb{R}^K$, we assume the prior

$$y_n \overset{\text{i.i.d.}}{\sim} \text{Cat}(\boldsymbol{\pi}).$$

See the math note below on the Categorical distribution.

Math Note

A random variable which takes on one of K discrete and unordered outcomes with probabilities $\pi_1, \dots, \pi_K$ follows the Categorical distribution with parameter $\boldsymbol{\pi} = \begin{bmatrix} \pi_1 & \dots & \pi_K \end{bmatrix}^\top$, written $\text{Cat}(\boldsymbol{\pi})$. For instance, a single die roll is distributed $\text{Cat}(\boldsymbol{\pi})$ for $\boldsymbol{\pi} = \begin{bmatrix} 1/6 \dots 1/6 \end{bmatrix}^\top$.

The density for $Y \sim \text{Cat}(\mathbf{p})$ is defined as

$$\begin{aligned} P(Y = 1) &= p_1 \\ &\dots \\ P(Y = K) &= p_K. \end{aligned}$$

This can be written more compactly as

$$p(y) = \prod_{k=1}^{K} p_k^{I_k}$$

where I_k is an indicator that equals 1 if $y = k$ and 0 otherwise.

We then assume some distribution for $\mathbf{x}_n$ conditional on observation n's class, Y_n. We typically assume all the $\mathbf{x}_n$ come from the same *family* of distributions, though the parameters depend on their class. For instance, we might have

$$\begin{aligned} \mathbf{x}_n | (Y_n = 1) &\sim \text{MVN}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1), \\ &\dots \\ \mathbf{x}_n | (Y_n = K) &\sim \text{MVN}(\boldsymbol{\mu}_K, \boldsymbol{\Sigma}_K), \end{aligned}$$

though we wouldn't let one conditional distribution be Multivariate Normal and another be Multivariate t. Note that it is possible, however, for the individual variables within the random vector $\mathbf{x}_n$ to follow different distributions. For instance, if $\mathbf{x}_n = \begin{bmatrix} x_{n1} & x_{n2} \end{bmatrix}^\top$, we might have

$$\begin{aligned} x_{n1} | (Y_n = k) &\sim \text{Bin}(n, p_k) \\ x_{n2} | (Y_n = k) &\sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \end{aligned}$$

The machine learning task is to estimate the parameters of these models—$\boldsymbol{\pi}$ for Y_n and whatever parameters might index the possible distributions of $\mathbf{x}_n | Y_n$, in this case $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ for $k = 1, \dots, K$. Once that's done, we can estimate $p(Y_n = k)$ and $p(\mathbf{x}_n | Y_n = k)$ for each class and, through Bayes' rule, choose the class that maximizes $p(Y_n = k | \mathbf{x}_n)$.

2. Parameter Estimation

2.1 Class Priors

Let's start by deriving the estimates for $\boldsymbol{\pi}$, the class priors. Let I_{nk} be an indicator which equals 1 if $Y_n = k$ and 0 otherwise. Then the joint likelihood and log-likelihood are given by

$$\begin{aligned} L\left(\boldsymbol{\pi}; \{\mathbf{x}_n, Y_n\}_{n=1}^N\right) &= \prod_{n=1}^{N} \prod_{k=1}^{K} \pi_k^{I_{nk}} \\ \log L\left(\boldsymbol{\pi}; \{\mathbf{x}_n, Y_n\}_{n=1}^N\right) &= \sum_{n=1}^{N} \sum_{k=1}^{K} I_{nk} \log(\pi_k) \\ &= \sum_{k=1}^{K} N_k \log(\pi_k), \end{aligned}$$

where $N_k = \sum_{n=1}^N I_{nk}$ gives the number of observations in class k for $k = 1, \dots, K$.

Math Note

The *Lagrangian function* provides a method for optimizing a function $f(\mathbf{x})$ subject to the constraint $g(\mathbf{x}) = 0$. The Lagrangian is given by

$$\mathcal{L}(\lambda, \mathbf{x}) = f(\mathbf{x}) - \lambda g(\mathbf{x}).$$

λ is known as the *Lagrange multiplier*. The critical points of $f(\mathbf{x})$ (subject to the equality constraint) are found by setting the gradients of $\mathcal{L}(\lambda, \mathbf{x})$ with respect to λ and $\mathbf{x}$ equal to 0.

Noting the constraint $\sum_{k=1}^K \pi_k = 1$ (or equivalently $\sum_{k=1}^K \pi_k - 1 = 0$), we can maximize the log-likelihood with the following Lagrangian.

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\pi}) &= \sum_{k=1}^K N_k \log(\pi_k) - \lambda(\sum_{k=1}^K \pi_k - 1). \\
\frac{\partial \mathcal{L}(\boldsymbol{\pi})}{\partial \pi_k} &= \frac{N_k}{\pi_k} - \lambda, \quad \forall\, k \in \{1, \dots, K\} \\
\frac{\partial \mathcal{L}(\boldsymbol{\pi})}{\partial \lambda} &= 1 - \sum_{k=1}^K \pi_k.
\end{aligned}$$

This system of equations gives an intuitive solution:

$$\hat{\pi}_k = \frac{N_k}{N}, \ \lambda = N,$$

which says that our estimate of $p(Y_n = k)$ is just the sample fraction of observations from class k.

2.2 Data Likelihood

The next step is to model the conditional distribution of $\mathbf{x}_n$ given Y_n so that we can estimate this distribution's parameters. This of course depends on the family of distributions we choose to model $\mathbf{x}_n$. Three common approaches are detailed below.

2.2.1 Linear Discriminative Analysis (LDA)

In LDA, we assume

$$\mathbf{x}_n | (Y_n = k) \sim \text{MVN}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}),$$

for $k = 1, \dots, K$. Note that each class has the same covariance matrix but a unique mean vector.

Let's derive the parameters in this case. First, let's find the likelihood and log-likelihood. Note that we can write the joint likelihood as follows,

$$L\left(\{\boldsymbol{\mu}_k\}_{k=1}^K, \boldsymbol{\Sigma}\right) = \prod_{n=1}^N \prod_{k=1}^K \Big(p\left(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}\right)\Big)^{I_{nk}},$$

since $(p(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}))^{I_{nk}}$ equals 1 if $y_n \neq k$ and $p(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma})$ otherwise. Then we plug in the Multivariate Normal PDF (dropping multiplicative constants) and take the log, as follows.

$$\begin{aligned}
L\left(\{\boldsymbol{\mu}_k\}_{k=1}^K, \boldsymbol{\Sigma}\right) &= \prod_{n=1}^N \prod_{k=1}^K \left(\frac{1}{\sqrt{|\boldsymbol{\Sigma}|}}\exp\left\{-\frac{1}{2}(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k)\right\}\right)^{I_{nk}} \\
\log L\left(\{\boldsymbol{\mu}_k\}_{k=1}^K, \boldsymbol{\Sigma}\right) &= \sum_{n=1}^N \sum_{k=1}^K I_{nk}\left(-\frac{1}{2}\log|\boldsymbol{\Sigma}| - \frac{1}{2}(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k)\right)
\end{aligned}$$

Math Note

The following matrix derivatives will be of use for maximizing the above log-likelihood.

For any invertible matrix $\mathbf{W}$,

$$\frac{\partial |\mathbf{W}|}{\partial \mathbf{W}} = |\mathbf{W}|\mathbf{W}^{-\top}, \tag{1}$$

where $\mathbf{W}^{-\top} = (\mathbf{W}^{-1})^\top$. It follows that

$$\frac{\partial \log |\mathbf{W}|}{\partial \mathbf{W}} = \mathbf{W}^{-\top}. \tag{2}$$

We also have

$$\frac{\partial \mathbf{x}^\top \mathbf{W}^{-1}\mathbf{x}}{\partial \mathbf{W}} = -\mathbf{W}^{-\top}\mathbf{x}\mathbf{x}^\top\mathbf{W}^{-\top}. \tag{3}$$

For any symmetric matrix $\mathbf{A}$,

$$\frac{\partial (\mathbf{x}-\mathbf{s})^\top \mathbf{A}(\mathbf{x}-\mathbf{s})}{\partial \mathbf{s}} = -2\mathbf{A}(\mathbf{x}-\mathbf{s}). \tag{4}$$

These results come from the Matrix Cookbook.

Let's start by estimating $\mathbf{\Sigma}$. First, simplify the log-likelihood to make the gradient with respect to $\mathbf{\Sigma}$ more apparent.

$$\log L\left(\{\boldsymbol{\mu}_k\}_{k=1}^K, \mathbf{\Sigma}\right) = -\frac{N}{2}\log|\mathbf{\Sigma}| - \frac{1}{2}\sum_{n=1}^N\sum_{k=1}^K I_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top \mathbf{\Sigma}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k).$$

Then, using equations (2) and (3) from the *Math Note*, we get

$$\frac{\partial \log L\left(\{\boldsymbol{\mu}_k\}_{k=1}^K, \mathbf{\Sigma}\right)}{\partial \mathbf{\Sigma}} = -\frac{N}{2}\mathbf{\Sigma}^{-\top} + \frac{1}{2}\sum_{n=1}^N\sum_{k=1}^K I_{nk}\mathbf{\Sigma}^{-\top}(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top\mathbf{\Sigma}^{-\top}.$$

Finally, we set this equal to 0 and multiply by $\mathbf{\Sigma}^{-1}$ on the left to solve for $\hat{\mathbf{\Sigma}}$:

$$\begin{aligned} 0 &= -\frac{N}{2} + \frac{1}{2}\left(\sum_{n=1}^N\sum_{k=1}^K I_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top\right)\mathbf{\Sigma}^{-\top} \\ \mathbf{\Sigma}^\top &= \frac{1}{N}\sum_{n=1}^N\sum_{k=1}^K I_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top \\ \hat{\mathbf{\Sigma}} &= \frac{1}{N}\mathbf{S}_T, \end{aligned}$$

where $\mathbf{S}_T = \sum_{n=1}^N\sum_{k=1}^K I_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top$.

Now, to estimate the $\boldsymbol{\mu}_k$, let's look at each class individually. Let N_k be the number of observations in class k and C_k be the set of observations in class k. Looking only at terms involving $\boldsymbol{\mu}_k$, we get

$$\log L(\boldsymbol{\mu}_k, \mathbf{\Sigma}) = -\frac{1}{2}\sum_{n\in C_k}\Big(\log|\mathbf{\Sigma}| + (\mathbf{x}_n - \boldsymbol{\mu}_k)^\top\mathbf{\Sigma}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k)\Big).$$

Using equation (4) from the *Math Note*, we calculate the gradient as

$$\frac{\partial \log L(\boldsymbol{\mu}_k, \mathbf{\Sigma})}{\partial \boldsymbol{\mu}_k} = \sum_{n\in C_k}\mathbf{\Sigma}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k).$$

Setting this gradient equal to 0 and solving, we obtain our $\boldsymbol{\mu}_k$ estimate:

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{N_k}\sum_{n\in C_k}\mathbf{x}_n = \bar{\mathbf{x}}_k,$$

where $\bar{\mathbf{x}}_k$ is the element-wise sample mean of all $\mathbf{x}_n$ in class k.

QDA looks very similar to LDA but assumes each class has its *own* covariance matrix:

$$\mathbf{x}_n | (Y_n = k) \sim \text{MVN}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

for $k = 1, \dots, K$. The log-likelihood is the same as in LDA except we replace $\boldsymbol{\Sigma}$ with $\boldsymbol{\Sigma}_k$:

$$\log L\left(\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^K\right) = \sum_{n=1}^N \sum_{k=1}^K I_{nk} \left(-\frac{1}{2}\log|\boldsymbol{\Sigma}_k| - \frac{1}{2}(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k)\right).$$

Again, let's look at the parameters for each class individually. The log-likelihood for class k is given by

$$\log L(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = -\frac{1}{2} \sum_{n \in C_k} \Big(\log|\boldsymbol{\Sigma}_k| + (\mathbf{x}_n - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k)\Big).$$

We could take the gradient of this log-likelihood with respect to $\boldsymbol{\mu}_k$ and set it equal to 0 to solve for $\hat{\boldsymbol{\mu}}_k$. However, we can also note that our $\hat{\boldsymbol{\mu}}_k$ estimate from the LDA approach will hold since this expression didn't depend on the covariance term (which is the only thing we've changed). Therefore, we again get

$$\hat{\boldsymbol{\mu}}_k = \bar{\mathbf{x}}_k.$$

To estimate the $\boldsymbol{\Sigma}_k$, we take the gradient of the log-likelihood for class k.

$$\frac{\partial \log L(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\partial \boldsymbol{\Sigma}_k} = -\frac{1}{2} \sum_{n \in C_k} \left(\boldsymbol{\Sigma}_k^{-\top} - \boldsymbol{\Sigma}_k^{-\top}(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-\top}\right).$$

Then we set this equal to 0 to solve for $\hat{\boldsymbol{\Sigma}}_k$:

$$\begin{aligned}
\sum_{n \in C_k} \boldsymbol{\Sigma}_k^{-\top} &= \sum_{n \in C_k} \boldsymbol{\Sigma}_k^{-\top}(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-\top} \\
N_k I &= \sum_{n \in C_k} (\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-\top} \\
\boldsymbol{\Sigma}_k^\top &= \frac{1}{N_k} \sum_{n \in C_k} (\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top \\
\hat{\boldsymbol{\Sigma}}_k &= \frac{1}{N_k} \mathbf{S}_k,
\end{aligned}$$

where $\mathbf{S}_k = \sum_{n \in C_k} (\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^\top$.

2.2.3 Naive Bayes

Naive Bayes assumes the random variables within $\mathbf{x}_n$ are independent conditional on the class of observation n. I.e. if $\mathbf{x}_n \in \mathbb{R}^D$, Naive Bayes assumes

$$p(\mathbf{x}_n | Y_n) = p(x_{n1}|Y_n) \cdot p(x_{n2}|Y_n) \cdot \ldots \cdot p(x_{nD}|Y_n).$$

This makes estimating $p(\mathbf{x}_n|Y_n)$ very easy—to estimate the parameters of $p(x_{nd}|Y_n)$, we can ignore all the variables in $\mathbf{x}_n$ other than x_{nd}!

As an example, assume $\mathbf{x}_n \in \mathbb{R}^2$ and we use the following model (where for simplicity n and σ_k^2 are known).

$$\begin{aligned}
x_{n1} | (Y_n = k) &\sim \mathcal{N}(\mu_k, \sigma_k^2) \\
x_{n2} | (Y_n = k) &\sim \text{Bin}(n, p_k).
\end{aligned}$$

Let the $\boldsymbol{\theta}_k = (\mu_k, \sigma_k^2, p_k)$ contain all the parameters for class k . The joint likelihood function would become

$$\begin{aligned}
L(\{\boldsymbol{\theta}_k\}_{k=1}^K) &= \prod_{n=1}^N \prod_{k=1}^K (p(\mathbf{x}|Y_n, \boldsymbol{\theta}_k))^{I_{nk}} \\
L(\{\boldsymbol{\theta}_k\}_{k=1}^K) &= \prod_{n=1}^N \prod_{k=1}^K \left(p(x_{n1}|\mu_k, \sigma_k^2) \cdot p(x_{n2}|p_k)\right)^{I_{nk}},
\end{aligned}$$

where the two are equal because of the Naive Bayes conditional independence assumption. This allows us to easily find maximum likelihood estimates. The rest of this sub-section demonstrates how those estimates would be found, though it is nothing beyond ordinary maximum likelihood estimation.

The log-likelihood is given by

$$\log L(\{\theta_k\}_{k=1}^K) = \sum_{n=1}^N \sum_{k=1}^K I_{nk} \left(\log p(x_{n1}|\mu_k, \sigma_k^2) + \log p(x_{n2}|p_k)\right).$$

As before, we estimate the parameters in each class by looking only at the terms in that class. Let's look at the log-likelihood for class k:

$$\begin{aligned}\log L(\theta_k) &= \sum_{n \in C_k} \log p(x_{n1}|\mu_k, \sigma_k^2) + \log p(x_{n2}|p_k) \\ &= \sum_{n \in C_k} -\frac{(x_{n1} - \mu_k)^2}{2\sigma_k^2} + x_{n2}\log(p_k) + (1 - x_{n2})\log(1 - p_k).\end{aligned}$$

Taking the derivative with respect to p_k, we're left with

$$\frac{\partial \log L(\theta_k)}{\partial p_k} = \sum_{n \in C_k} \frac{x_{n2}}{p_k} - \frac{1 - x_{n2}}{1 - p_k},$$

which, will give us $\hat{p}_k = \frac{1}{N_k}\sum_{n \in C_k} x_{n2}$ as usual. The same process would again give typical results for μ_k and and σ_k^2.

3. Making Classifications

Regardless of our modeling choices for $p(\mathbf{x}_n|Y_n)$, classifying new observations is easy. Consider a test observation $\mathbf{x}_0$. For $k = 1, \ldots, K$, we use Bayes' rule to calculate

$$\begin{aligned}p(Y_0 = k|\mathbf{x}_0) &\propto p(\mathbf{x}_0|Y_0 = k)p(Y_0 = k) \\ &= \hat{p}(\mathbf{x}_0|Y_0 = k)\hat{\pi}_k,\end{aligned}$$

where $\hat{p}$ gives the estimated density of $\mathbf{x}_0$ conditional on Y_0. We then predict $Y_0 = k$ for whichever value k maximizes the above expression.

Construction

In this section, we build LDA, QDA, and Naive Bayes classifiers. We will demo these classes on the wine dataset.

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets

wine = datasets.load_wine()
X, y = wine.data, wine.target
```

LDA

An implementation of linear discriminant analysis (LDA) is given below. The main method is `.fit()`. This method makes three important estimates. For each k, we estimate π_k, the class prior probability. For each class we also estimate the mean of the data in that class, $\boldsymbol{\mu}_k$. Finally, we estimate the overall covariance matrix across classes, $\boldsymbol{\Sigma}$. The formulas for these estimates are detailed in the concept section.

The second two methods, `.mvn_density()` and `.classify()` are for classifying new observations. `.mvn_density()` just calculates the density (up to a multiplicative constant) of a Multivariate Normal sample provided the mean vector and covariance matrix. `.classify()` actually makes the classifications for each test observation. It calculates the density for each class, $p(\mathbf{x}_n|Y_n = k)$, and multiplies this by the prior class probability, $p(Y_n = k) = \pi_k$, to get a posterior class probability, $p(Y_n = k|\mathbf{x}_n)$. It then predicts the class with the highest posterior probability.

```
class LDA:

    ## Fitting the model
    def fit(self, X, y):

        ## Record info
        self.N, self.D = X.shape
        self.X = X
        self.y = y

        ## Get prior probabilities
        self.unique_y, unique_y_counts = np.unique(self.y, return_counts = True) #
returns unique y and counts
        self.pi_ks = unique_y_counts/self.N

        ## Get mu for each class and overall Sigma
        self.mu_ks = []
        self.Sigma = np.zeros((self.D, self.D))
        for i, k in enumerate(self.unique_y):

            X_k = self.X[self.y == k]
            mu_k = X_k.mean(0).reshape(self.D, 1)
            self.mu_ks.append(mu_k)

            for x_n in X_k:
                x_n = x_n.reshape(-1,1)
                x_n_minus_mu_k = (x_n - mu_k)
                self.Sigma += np.dot(x_n_minus_mu_k, x_n_minus_mu_k.T)

        self.Sigma /= self.N


    ## Making classifications

    def _mvn_density(self, x_n, mu_k, Sigma):
        x_n_minus_mu_k = (x_n - mu_k)
        density = np.exp(-(1/2)*x_n_minus_mu_k.T @ np.linalg.inv(Sigma) @
x_n_minus_mu_k)
        return density

    def classify(self, X_test):

        y_n = np.empty(len(X_test))
        for i, x_n in enumerate(X_test):

            x_n = x_n.reshape(-1, 1)
            p_ks = np.empty(len(self.unique_y))

            for j, k in enumerate(self.unique_y):
                p_x_given_y = self._mvn_density(x_n, self.mu_ks[j], self.Sigma)
                p_y_given_x = self.pi_ks[j]*p_x_given_y
                p_ks[j] = p_y_given_x

            y_n[i] = self.unique_y[np.argmax(p_ks)]

        return y_n
```

We fit the LDA model below and classify the training observations. As the output shows, we have 100% training accuracy.

```
lda = LDA()
lda.fit(X, y)
yhat = lda.classify(X)
np.mean(yhat == y)
```

```
1.0
```

The function below visualizes class predictions based on the input values for a model with $\mathbf{x}_n \in \mathbb{R}^2$. To apply this function, we build a model with only two columns from the `wine` dataset. We see that the decision boundaries are linear, as we expect from LDA.

```python
def graph_boundaries(X, model, model_title, n0 = 100, n1 = 100, figsize = (7, 5),
label_every = 4):

        # Generate X for plotting
        d0_range = np.linspace(X[:,0].min(), X[:,0].max(), n0)
        d1_range = np.linspace(X[:,1].min(), X[:,1].max(), n1)
        X_plot = np.array(np.meshgrid(d0_range, d1_range)).T.reshape(-1, 2)

        # Get class predictions
        y_plot = model.classify(X_plot).astype(int)

        # Plot
        fig, ax = plt.subplots(figsize = figsize)
        sns.heatmap(y_plot.reshape(n0, n1).T,
                   cmap = sns.color_palette('Pastel1', 3),
                   cbar_kws = {'ticks':sorted(np.unique(y_plot))})
        xticks, yticks = ax.get_xticks(), ax.get_yticks()
        ax.set(xticks = xticks[::label_every], xticklabels = d0_range.round(2)
[::label_every],
               yticks = yticks[::label_every], yticklabels = d1_range.round(2)
[::label_every])
        ax.set(xlabel = 'X1', ylabel = 'X2', title = model_title + ' Predictions by X1
and X2')
        ax.set_xticklabels(ax.get_xticklabels(), rotation=0)
```

```python
X_2d = X.copy()[:,2:4]
lda_2d = LDA()
lda_2d.fit(X_2d, y)
graph_boundaries(X_2d, lda_2d, 'LDA')
```

../../_images/construction_10_01.png

QDA

The QDA model is implemented below. It is nearly identical to LDA except the covariance matrices $\mathbf{\Sigma}_k$ are estimated separately. Again see the concept section for details.

```python
class QDA:

    ## Fitting the model

    def fit(self, X, y):

        ## Record info
        self.N, self.D = X.shape
        self.X = X
        self.y = y


        ## Get prior probabilities
        self.unique_y, unique_y_counts = np.unique(self.y, return_counts = True) #
returns unique y and counts
        self.pi_ks = unique_y_counts/self.N


        ## Get mu and Sigma for each class
        self.mu_ks = []
        self.Sigma_ks = []
        for i, k in enumerate(self.unique_y):

            X_k = self.X[self.y == k]
            mu_k = X_k.mean(0).reshape(self.D, 1)
            self.mu_ks.append(mu_k)

            Sigma_k = np.zeros((self.D, self.D))
            for x_n in X_k:
                x_n = x_n.reshape(-1,1)
                x_n_minus_mu_k = (x_n - mu_k)
                Sigma_k += np.dot(x_n_minus_mu_k, x_n_minus_mu_k.T)
            self.Sigma_ks.append(Sigma_k/len(X_k))

    ## Making classifications

    def _mvn_density(self, x_n, mu_k, Sigma_k):
        x_n_minus_mu_k = (x_n - mu_k)
        density = np.linalg.det(Sigma_k)**(-1/2) * np.exp(-(1/2)*x_n_minus_mu_k.T @
np.linalg.inv(Sigma_k) @ x_n_minus_mu_k)
        return density

    def classify(self, X_test):

        y_n = np.empty(len(X_test))
        for i, x_n in enumerate(X_test):

            x_n = x_n.reshape(-1, 1)
            p_ks = np.empty(len(self.unique_y))

            for j, k in enumerate(self.unique_y):

                p_x_given_y = self._mvn_density(x_n, self.mu_ks[j], self.Sigma_ks[j])
                p_y_given_x = self.pi_ks[j]*p_x_given_y
                p_ks[j] = p_y_given_x

            y_n[i] = self.unique_y[np.argmax(p_ks)]

        return y_n
```

```python
qda = QDA()
qda.fit(X, y)
yhat = qda.classify(X)
np.mean(yhat == y)
```

```
0.9943820224719101
```

The below plot shows predictions based on the input variables for the QDA model. As expected, the decision boundaries are quadratic, rather than linear. We also see that the area corresponding to class 2 is much smaller than the other areas. This suggests that either there were fewer observations in class 2 or the estimated variance of the input variables for observations in class 2 was smaller than the variance for observations in other classes .

```python
qda_2d = QDA()
qda_2d.fit(X_2d, y)
graph_boundaries(X_2d, qda_2d, 'QDA')
```

../../_images/construction_16_0.png

Naive Bayes

Finally, we implement a Naive Bayes model below. This model allows us to assign each variable in our dataset a distribution, though by default they are all assumed to be Normal. Since each variable has its own distribution, estimating the model's parameters is more involved. For each variable and each class, we estimate the parameters separately through the `_estimate_class_parameters`. The structure below allows for Normal, Bernoulli, and Poisson distributions, though any distribution could be implemented.

Again, we make predictions by calculating $p(Y_n = k|\mathbf{x}_n)$ for $k = 1, \dots, K$ through Bayes' rule and predicting the class with the highest posterior probability. Since each variable can have its own distribution, this problem is also more involved. The `_get_class_probability` method calculates the probability density of a test observation's input variables. By the conditional independence assumption, this is just the product of the individual densities.

Naive Bayes performs worse than LDA or QDA on the training data, suggesting the conditional independence assumption might be inappropriate for this problem.

```python
class NaiveBayes:

    ######## Fit Model ########

    def _estimate_class_parameters(self, X_k):

        class_parameters = []

        for d in range(self.D):
            X_kd = X_k[:,d] # only the dth column and the kth class

            if self.distributions[d] == 'normal':
                mu = np.mean(X_kd)
                sigma2 = np.var(X_kd)
                class_parameters.append([mu, sigma2])

            if self.distributions[d] == 'bernoulli':
                p = np.mean(X_kd)
                class_parameters.append(p)

            if self.distributions[d] == 'poisson':
                lam = np.mean(X_kd)
                class_parameters.append(p)

        return class_parameters

    def fit(self, X, y, distributions = None):

        ## Record info
        self.N, self.D = X.shape
        self.X = X
        self.y = y
        if distributions is None:
            distributions = ['normal' for i in range(len(y))]
        self.distributions = distributions


        ## Get prior probabilities
        self.unique_y, unique_y_counts = np.unique(self.y, return_counts = True) #
returns unique y and counts
        self.pi_ks = unique_y_counts/self.N


        ## Estimate parameters
        self.parameters = []
        for i, k in enumerate(self.unique_y):
            X_k = self.X[self.y == k]
            self.parameters.append(self._estimate_class_parameters(X_k))


    ######## Make Classifications ########

    def _get_class_probability(self, x_n, j):

        class_parameters = self.parameters[j] # j is index of kth class
        class_probability = 1

        for d in range(self.D):
            x_nd = x_n[d] # just the dth variable in observation x_n

            if self.distributions[d] == 'normal':
                mu, sigma2 = class_parameters[d]
                class_probability *= sigma2**(-1/2)*np.exp(-(x_nd - mu)**2/sigma2)

            if self.distributions[d] == 'bernoulli':
                p = class_parameters[d]
                class_probability *= (p**x_nd)*(1-p)**(1-x_nd)

            if self.distributions[d] == 'poisson':
                lam = class_parameters[d]
                class_probability *= np.exp(-lam)*lam**x_nd

        return class_probability

    def classify(self, X_test):

        y_n = np.empty(len(X_test))
        for i, x_n in enumerate(X_test): # loop through test observations

            x_n = x_n.reshape(-1, 1)
            p_ks = np.empty(len(self.unique_y))

            for j, k in enumerate(self.unique_y): # loop through classes

                p_x_given_y = self._get_class_probability(x_n, j)
                p_y_given_x = self.pi_ks[j]*p_x_given_y # bayes' rule
                p_ks[j] = p_y_given_x

            y_n[i] = self.unique_y[np.argmax(p_ks)]

        return y_n
```

```
nb = NaiveBayes()
nb.fit(X, y)
yhat = nb.classify(X)
np.mean(yhat == y)
```

```
0.9775280898876404
```

```
nb_2d = NaiveBayes()
nb_2d.fit(X_2d, y)
graph_boundaries(X_2d, nb_2d, 'Naive Bayes')
```

../../_images/construction_21_0.png

Implementation

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
wine = datasets.load_wine()
X, y = wine.data, wine.target
```

The code below shows `scikit-learn` implementations of LDA, QDA, and Naive Bayes using the wine dataset. Note that the Naive Bayes implementation assumes *all* variables follow a Normal distribution, unlike the construction in the previous section.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis,
QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB

lda = LinearDiscriminantAnalysis()
lda.fit(X, y);

qda = QuadraticDiscriminantAnalysis()
qda.fit(X, y);

nb = GaussianNB()
nb.fit(X, y);
```

Next, let's check that these `scikit-learn` implementations return the same decision boundaries as our constructions in the previous section. The code to create these graphs is written below.

```
def graph_boundaries(X, model, model_title, n0 = 1000, n1 = 1000, figsize = (7, 5),
label_every = 4):

        # Generate X for plotting
        d0_range = np.linspace(X[:,0].min(), X[:,0].max(), n0)
        d1_range = np.linspace(X[:,1].min(), X[:,1].max(), n1)
        X_plot = np.array(np.meshgrid(d0_range, d1_range)).T.reshape(-1, 2)

        # Get class predictions
        y_plot = model.predict(X_plot).astype(int)

        # Plot
        fig, ax = plt.subplots(figsize = figsize)
        sns.heatmap(y_plot.reshape(n0, n1).T,
                    cmap = sns.color_palette('Pastel1', 3),
                    cbar_kws = {'ticks':sorted(np.unique(y_plot))})
        xticks, yticks = ax.get_xticks(), ax.get_yticks()
        ax.set(xticks = xticks[::label_every], xticklabels = d0_range.round(2)
[::label_every],
               yticks = yticks[::label_every], yticklabels = d1_range.round(2)
[::label_every])
        ax.set(xlabel = 'X1', ylabel = 'X2', title = model_title + ' Predictions by X1
and X2')
        ax.set_xticklabels(ax.get_xticklabels(), rotation=0)
```

```
X_2d = X.copy()[:,2:4]
lda_2d = LinearDiscriminantAnalysis()
lda_2d.fit(X_2d, y);
graph_boundaries(X_2d, lda_2d, 'LDA')
```

../../_images/code_6_0.png

```
qda_2d = QuadraticDiscriminantAnalysis()
qda_2d.fit(X_2d, y);
graph_boundaries(X_2d, qda_2d, 'QDA')
```

../../_images/code_7_01.png

```
nb_2d = GaussianNB()
nb_2d.fit(X_2d, y);
graph_boundaries(X_2d, nb_2d, 'Naive Bayes')
```

../../_images/code_8_0.png

Concept

A decision tree is an interpretable machine learning method for regression and classification. Trees iteratively split samples of the training data based on the value of a chosen predictor; the goal of each split is to create two sub-samples, or "children," with greater *purity* of the target variable than their "parent". For classification tasks, purity means the first child should have observations primarily of one class and the second should have observations primarily of another. For regression tasks, purity means the first child should have observations with high values of the target variable and the second should have observations with low values.

An example of a classification decision tree using the penguins dataset is given below. The tree attempts to classify a penguin's species—*Adelie*, *Gentoo*, or *Chinstrap*—from information about its flippers and bill. The first "node" shows that there are 333 training observations, 146 *Adelie*, 68 *Gentoo*, and 119 *Chinstrap*. We first split based on whether the penguin's flipper length is less than or equal to 206.5 mm. If so, the penguin moves to the node on the left and if not, it moves to the node on the right. We then repeat this process for each of the child nodes.

tree

Once we've reached the bottom of the tree, we make our predictions. For instance, if a test observation has a `flipper_length` of 210 and a `bill_depth` of 12, we would follow the tree and classify it as a *Gentoo*. This simple decision process makes trees very interpretable. However, they may suffer in terms of precision (accuracy of predictions) and robustness (sensitivity to variable training data).

This chapter demonstrates how decision trees are built. The first section covers regression tasks, where the target variable is quantitative, and the second covers classification tasks, where the target variable is categorical.

Regression Trees

Decision trees are deeply rooted in tree-based terminology. Before discussing decision trees in depth, let's go over some of this vocabulary.

- **Node**: A node is comprised of a sample of data and a decision rule.
- **Parent**, **Child**: A parent is a node in a tree associated with exactly two child nodes. Observations directed to a parent node are next directed to one of that parent's two children nodes.
- **Rule**: Each parent node has a rule that determines toward which of the child nodes an observation will be directed. The rule is based on the value of one (and only one) of an observation's predictors.
- **Buds**: Buds are nodes that have not yet been split. In other words, they are children that are eligible to become parents. Nodes are only considered buds during the training process. Nodes that are never split are considered leaves.
- **Leaf**: Leaves, also known as *terminal nodes*, are nodes that are never split. Observations move through a decision tree until reaching a leaf. The fitted value of a test observation is determined by the training observations that land in the same leaf.

Let's return to the tree introduced in the previous page to see these vocabulary terms in action. Each square in this tree is a node. The sample of data for node A is the collection of penguins with flipper lengths under 206.5mm and the decision rule for node A is whether the bill length is less than or equal to 43.35mm. Node A is an example of a parent node, and nodes B and C are examples of child nodes. Before node A was split into child nodes B and C, it was an example of a bud. Since B and C are never split, they are examples of leaves.

tree

The rest of this section describes how to fit a decision tree. Our tree starts with an initial node containing all the training data. We then assign this node a rule, which leads to the creation of two child nodes. Training observations are assigned to one of these two child nodes based on their response to the rule. How these rules are made is discussed in the first sub-section below.

Next, these child nodes are considered *buds* and are ready to become parent nodes. Each bud is assigned a rule and split into two child nodes. We then have four *buds* and each is once again split. We continue this process until some stopping rule is reached (discussed later). Finally, we run test observations through the tree and assign them fitted values according to the leaf they fall in.

In this section we'll discuss how to build a tree as well as how to choose the tree's optimal hyperparameters.

Building a Tree

Building a tree consists of iteratively creating rules to split nodes. We'll first discuss rules in greater depth, then introduce a tree's objective function, then cover the splitting process, and finally go over making predictions with a built tree. Suppose we have predictors $\mathbf{x}_n \in \mathbb{R}^D$ and a quantitative target variable y_n for $n = 1, \ldots, N$.

Defining Rules

Let's first clarify what a rule actually is. Rules in a decision tree determine how observations from a parent node are divided between two child nodes. Each rule is based on only *one* predictor from the parent node's training sample. Let x_{nd} be the d^{th} predictor in $\mathbf{x}_n$. If x_{nd} is quantitative, a rule is of the form

$$x_{nd} \leq t,$$

for some *threshold* value t. If x_{nd} is categorical, a rule is of the form

$$x_{nd} \in \mathcal{S},$$

where $\mathcal{S}$ is a set of possible values of the d^{th} predictor.

Let's introduce a little notation to clear things up. Let $\mathcal{N}_m$ represent the tree's m^{th} node and $\mathcal{R}_m$ be the rule for $\mathcal{N}_m$. Then let $\mathcal{C}_m^L$ and $\mathcal{C}_m^R$ be the child nodes of $\mathcal{N}_m$. As a convention, suppose observations are directed to $\mathcal{C}_m^L$ if they satisfy the rule and $\mathcal{C}_m^R$ otherwise. For instance, let

$$\mathcal{R}_m = (x_{n2} \leq 4).$$

Then observation n that passes through $\mathcal{N}_m$ will go to $\mathcal{C}_m^L$ if $x_{n2} \leq 4$ and $\mathcal{C}_m^R$ otherwise. On a diagram like the one above, the child node for observations satisfying the rule is often to the left while the node for those not satisfying the rule is to the right, hence the superscripts L and R

The Objective

Consider first a single node $\mathcal{N}_m$. Let $n \in \mathcal{N}_m$ be the collection of training observations that pass through the node and $\bar{y}_m$ be the sample mean of these observations. The residual sum of squares (RSS) for $\mathcal{N}_m$ is defined as

$$RSS_m = \sum_{n \in \mathcal{N}_m} \left(y_n - \bar{y}_m\right)^2.$$

The loss function for the entire tree is the RSS across buds (if still being fit) or across leaves (if finished fitting). Letting I_m be an indicator that node m is a leaf or bud (i.e. *not* a parent), the total loss for the tree is written as

$$RSS_T = \sum_m \sum_{n \in \mathcal{N}_m} I_m RSS_m.$$

In choosing splits, we hope to reduce RSS_T as much as possible. We can write the reduction in RSS_T from splitting bud $\mathcal{N}_m$ into children $\mathcal{C}_m^L$ and $\mathcal{C}_m^R$ as the RSS of the bud minus the sum of the RSS of its children, or

$$
\begin{aligned}
\Delta RSS_T &= RSS_m - \left(RSS_{\mathcal{C}_m^L} + RSS_{\mathcal{C}_m^R}\right) \\
&= \sum_{n \in \mathcal{N}_m} (y_n - \bar{y}_m)^2 - \left(\sum_{n \in \mathcal{C}_m^L} \left(y_n - \bar{y}_m^L\right)^2 + \sum_{n \in \mathcal{C}_m^R} \left(y_n - \bar{y}_m^R\right)^2 \right),
\end{aligned}
$$

where $\bar{y}_m^L$ and $\bar{y}_m^R$ are the sample mean of the child nodes.

Making Splits

Finding the optimal sequence of splits to minimize RSS_T becomes a combinatorially infeasible task as we add predictors and observations. Instead, we take a greedy approach known as *recursive binary splitting.* When splitting each bud, we consider all possible predictors and all possible ways to split that predictor. If the predictor is quantitative, this means considering all possible thresholds for splitting. If the predictor is categorical, this means considering all ways to split the categories into two groups. After considering all possible splits, we choose the one with the greatest reduction in the bud's RSS. This process is detailed below.

Note that once a node is split, what happens to one of its children is independent of what happens to the other. We can therefore build the tree in layers—first splitting the initial node, then splitting each of the initial node's children, then splitting all four of those children's children, etc.

For each layer (starting with only the initial node), we split with the following process.

- For each bud m on that layer,
  - For each predictor d,
    - If the predictor is quantitative:
      - For each value of that predictor among the observations in the bud [1],
        - Let this value be the threshold value t and consider the reduction in RSS_m from splitting at this threshold. I.e. consider the reduction in RSS_m from the rule

$$\mathcal{R} = x_{nd} \leq t.$$

    - If the predictor is categorical:
      - Rank the categories according to the average value of the target variable for observations in each category. If there are V distinct values, index these categories $c_1, \dots, c_V$ in ascending order.
      - For $v = 1, \dots, V - 1$ [1],
        - Consider the set $\mathcal{S} = \{c_1, \dots, c_v\}$ and consider the reduction in RSS_m from the rule

$$\mathcal{R} = x_{nd} \in \mathcal{S}.$$

      In words, we consider the set of categories corresponding to the lowest average target variable and split if the observation's d^{th} predictor falls in this set. We then add one more category to this set and repeat.
  - Choose the predictor, and value (if quantitative) or set (if categorical) with the greatest reduction in RSS_m and split accordingly.

We repeat this process until some stopping rule is reached.

Making Predictions

Making predictions with a built decision tree is very straightforward. For each test observation, we simply run the observation through the built tree. Once it reaches a leaf (a terminal node), we predict its target variable to be the sample mean of the training observations in that leaf.

Choosing Hyperparameters

As with any bias-variance tradeoff issue, we want our tree to learn valuable patterns from the data without reading into the idiosyncrasies of our particular training set. Without any sort of regulation, a tree will minimize this total RSS by moving each training observation into its own leaf, causing obvious overfitting. Below are three common methods for limiting the depth of a tree.

Size Regulation

A simple way to limit a tree's size is to directly regulate its depth, the size of its terminal nodes, or both.

We can define the *depth* of a node as the number of parent nodes that have come before it. For instance, the initial node has depth 0, the children of the first split have depth 1, and the children of the second split have depth 2. We can prevent a tree from overfitting by setting a maximum depth. To do this, we simply restrict nodes of this depth from being split.

The *size* of a node is defined as the number of training observations in that node. We can also prevent a tree from overfitting by setting a minimum size for parent nodes or child nodes. When setting a minimum size for parent nodes, we restrict nodes from splitting if they are below a certain size. When setting a minimum size for child nodes, we ban any splits that would create child nodes smaller than a certain size.

These parameters are typically chosen with cross validation.

Minimum Reduction in RSS

Another simple method for avoiding overfitting is to require that any split decrease the residual sum of squares, RSS_T, by a certain amount. Ideally, this allows the tree to separate groups that are sufficiently dissimilar while preventing it from making needless splits. Again, the exact required reduction in RSS_T should be chosen through cross validation.

One potential pitfall of this method is that early splits might elicit small decreases in RSS_T while later splits will elicit much larger ones. By requiring a minimum reduction in RSS_T, however, we might not reach these later splits. For instance, consider the dataset below with input variables a and b and target variable y. The first split, whether by a or b, will not reduce RSS_T at all while the second split would reduce it to 0.

a	b	y
0	0	0
0	1	10
1	0	10
1	1	0

Pruning

A third strategy to manage a decision tree's size is through *pruning*. To prune a tree, first intentionally overfit it by, for instance, setting a low minimum node size or a high maximum depth. Then, just like pruning a literal tree, we cut back the unnecessary branches. Specifically, one by one we undue the least helpful split by re-joining terminal leaves. At any given step, we undue the split that causes the least increase in RSS_T.

When pruning a tree, we want to balance tree size with residual error. Define the size of a tree (not to be confused with the size of a *node*) to equal its number of terminal leaves, denoted $|T|$ for tree T. The residual error for tree T is defined by RSS_T. To compare possible trees when pruning, we use the following loss function

$$\mathcal{L}(T) = RSS_T + \lambda|T|,$$

where λ is a regularization parameter. We choose the tree T that minimizes $\mathcal{L}(T)$.

Increasing λ increases the amount of regularization and limits overfitting by penalizing larger trees. Increasing λ too much, however, can cause undercutting by preventing the tree from making needed splits. How then do we choose λ? With cross validation! Specifically, for each validation fold, we overfit the tree and prune it back using a range of λ values. Note that λ does not affect *how* we prune the tree, only *which* pruned tree we prefer. For each value of λ, we calculate the average validation accuracy (likely using RSS_T calculated on the validation set) across folds. We choose the value of λ that maximizes this accuracy and the tree which is preferred by that value of λ.

[1](1,2) For a quantitative predictor, we actually consider all but the largest. If we chose the rule $R = x_{nd} \leq t$ and t was the largest value, this would not leave any observations in the second child. Similarly, for a categorical predictor, we do not consider the V^{th} category since this would not leave any observations for the second child.

Classification Trees

Building a classification tree is essentially identical to building a regression tree but optimizing a different loss function—one fitting for a categorical target variable. For that reason, this section only covers the details unique to classification trees, rather than demonstrating how one is built from scratch. To understand the tree-building process in general, see the previous section.

Suppose for the following that we have data $\{\mathbf{x}_n, y_n\}_{n=1}^N$ with predictor variables $\mathbf{x}_n \in \mathbb{R}^D$ and a categorical target variable $y_n \in \{1, \dots, K\}$.

Building a Tree

The Objective

Two common loss functions for a classification are the *Gini index* and the *cross-entropy*. Let $n \in \mathcal{N}_m$ be the collection of training observations that pass through node m and let $\hat{y}_{mk}$ be the fraction of these observations in class k for $k = 1, \dots, K$. The Gini index for $\mathcal{N}_m$ is defined as

$$\mathcal{L}_G(\mathcal{N}_m) = \sum_{k=1}^K \hat{p}_{mk}(1 - \hat{p}_{mk}),$$

and the cross-entropy is defined as

$$\mathcal{L}_E(\mathcal{N}_m) = -\sum_{k=1}^K \hat{p}_{mk} \log \hat{p}_{mk}.$$

The Gini index and cross-entropy are measures of *impurity*—they are higher for nodes with more equal representation of different classes and lower for nodes represented largely by a single class. As a node becomes more pure, these loss measures tend toward zero.

In order to evaluate the purity of a *split* (rather than that of a *node*), we use the weighted Gini index or weighted cross-entropy. Consider a split S_m of bud $\mathcal{N}_m$ which creates children $\mathcal{C}_m^L$ and $\mathcal{C}_m^R$. Let the fraction of training observations going to $\mathcal{C}_m^L$ be f_L and the fraction going to $\mathcal{C}_m^R$ be f_R. The weighted loss (whether with the Gini index or the cross-entropy) is defined as

$$\mathcal{L}(S_m) = f_L \cdot \mathcal{L}(\mathcal{C}_m^L) + f_R \cdot \mathcal{L}(\mathcal{C}_m^R).$$

The lower the weighted loss the better.

Making Splits

As with regression trees, we will make splits one layer at a time. When splitting bud m, we use the same procedure as in regression trees: we calculate the loss from splitting the node at each value of each predictor and make the split with the lowest loss.

For quantitative predictors, the procedure is identical to the regression tree procedure except we aim to minimize $\mathcal{L}(S_m)$ rather than maximally reducing RSS_m. For categorical predictors, we cannot rank the categories according to the average value of the target variable as we did for regression trees because the target is not continuous! If our target is binary, we can rank the predictor's categories according to the fraction of the corresponding target variables in class 1 versus class 0 and proceed in the same was as we did for regression trees.

If the target is not binary, we are out of luck. One (potentially computationally-intensive) method is to try all possible binary groupings of the categorical value and see what grouping minimizes $\mathcal{L}(S_m)$. Another would be a one-versus-rest approach where we only consider isolating one category at a time from the rest. Suppose we had a predictor with four categories, A, B, C, and D. The first method requires the following 7 splits while the second method requires only the first four splits.

$$\begin{aligned} A &\text{ vs. } B, C, D \\ B &\text{ vs. } A, C, D \\ C &\text{ vs. } A, B, D \\ D &\text{ vs. } A, C, D \\ A, B &\text{ vs. } C, D \\ A, C &\text{ vs. } B, D \\ A, D &\text{ vs. } B, C \end{aligned}$$

Making Predictions

Classifying test observations with a fully-grown tree is very straightforward. First, run an observation through the tree and observe which leaf it lands in. Then classify it according to the most common class in that leaf.

For large enough leaves, we can also estimate the probability that the test observation belongs to any given class: if test observation j lands in leaf m, we can estimate $p(y_j = k)$ with $\hat{p}_{mk}$ for each k.

Choosing Hyperparameters

In the regression tree section, we discussed three methods for managing a tree's size to balance the bias-variance tradeoff. The same three methods can be used for classification trees with slight modifications, which we cover next. For a full overview on these methods, please review the regression tree section.

Size Regulation

We can again use cross validation to fix the maximum depth of a tree or the minimum size of its terminal nodes. Unlike with regression trees, however, it is common to use a different loss function for cross validation than we do for building the tree. Specifically, we typically build classification trees with the Gini index or cross-entropy but use the *misclassification rate* to determine the hyperparameters with cross validation. The *misclassification rate* is simply the percent of observations we incorrectly classify. This is typically a more desirable metric to minimize than the Gini index or cross-entropy since it tells us more about our ultimate goal of correctly classifying test observations.

To conduct cross validation, then, we would build the tree using the Gini index or cross-entropy for a set of hyperparameters, then pick the tree with the lowest misclassification rate on validation samples.

Maximum Split Loss

Another regularization method for regression trees was to require that each split reduce the RSS by a certain amount. An equivalent approach for classification trees is to require that each split have a weighted loss below some minimum threshold. This threshold should be chosen through cross validation, again likely with the misclassification rate as the loss function.

Pruning

To prune a regression tree, we first fit a large tree, then searched for a sub-tree that could achieve a low RSS without growing too large and possibly overfitting. Specifically, we looked for the sub-tree T that minimized the following loss function, where $|T|$ gives the number of terminal leaves and λ is a regularization parameter:

$$\mathcal{L}(T) = RSS_T + \lambda |T|.$$

We prune a classification tree in a nearly identical fashion. First, we grow an intentionally overfit tree, T_0. We then consider all splits leading to terminal nodes and undo the one with the greatest loss by re-joining its child nodes. I.e. we undo the split at the node m with the greatest $\mathcal{L}(S_m)$ among nodes leading to leaves. We then repeat this process iteratively until we are left with the tree containing only the initial node. For each tree, we record its size (the number of terminal leaves) and its misclassification rate, $\mathcal{M}_T$. We then choose the tree that minimizes

$$\mathcal{L}(T) = \mathcal{M}_T + \lambda |T|,$$

where again λ is chosen through cross validation. For a fuller overview of how we use cross validation to choose λ, see the pruning section in the regression tree page.

Construction

The next two pages demonstrate how to construct a regression tree and classification tree from scratch. These models are slightly more code-heavy than previous models we've seen due to their non-parametric and recursive nature.

Regression Trees

```python
## Import packages
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets

## Load data
tips = sns.load_dataset('tips')
X = np.array(tips.drop(columns = 'tip'))
y = np.array(tips['tip'])

## Train-test split
np.random.seed(1)
test_frac = 0.25
test_size = int(len(y)*test_frac)
test_idxs = np.random.choice(np.arange(len(y)), test_size, replace = False)
X_train = np.delete(X, test_idxs, 0)
y_train = np.delete(y, test_idxs, 0)
X_test = X[test_idxs]
y_test = y[test_idxs]
```

Because of their iterative structure, their exhaustive searching, and their non-parametric nature, decision trees are more computationally complex than methods we've seen previously. To clear things up, the construction code is divided into three sections: helper functions, helper classes, and the main decision tree regressor class.

We will build our regression tree on the tips dataset from `seaborn`. This dataset has a continuous target variable (tip amount) with both quantitative and categorical predictors. Recall that trees are able to handle categorical predictors without creating one-hot encoded variables, unlike other methods we've seen.

1. Helper Functions

Below are three helper functions we will use in our regression tree. `RSS_reduction()` measures how much a split reduces a parent node's RSS by subtracting the sum of the child RSS values from the parent RSS. We will use this to determine the best split for any given bud.

The next function, `sort_x_by_y()`, returns a sorted list of the unique categories in some predictor `x` according to the mean of the corresponding target values `y`. We will use this to search for splits on categorical predictors.

Finally, `all_rows_equal()` checks if all of a bud's rows (observations) are equal across all predictors. If this is the case, this bud will not be split and instead becomes a terminal leaf.

```python
def RSS_reduction(child_L, child_R, parent):
    rss_parent = sum((parent - np.mean(parent))**2)
    rss_child_L = sum((child_L - np.mean(child_L))**2)
    rss_child_R = sum((child_R - np.mean(child_R))**2)
    return rss_parent - (rss_child_L + rss_child_R)

def sort_x_by_y(x, y):
    unique_xs = np.unique(x)
    y_mean_by_x = np.array([y[x == unique_x].mean() for unique_x in unique_xs])
    ordered_xs = unique_xs[np.argsort(y_mean_by_x)]
    return ordered_xs

def all_rows_equal(X):
    return (X == X[0]).all()
```

2. Helper Classes

Below are two helpful classes for our main regression tree class. The first, `Node`, represents nodes within our tree. They identify a node's ID and the ID of its parent, its sample of the predictors and the target variable, its size, its depth, and whether or not it is a leaf.

The second, Splitter, is used to identify the best split of any given bud. It identifies the split's reduction in RSS; the variable, d, used to make the split; the variable's data type; and the threshold, t, (if quantitative) or the set of values, L_values, corresponding to the left child node (if categorical). The Splitter class fills in these values with its _replace_split() method. This method is called if we find a split better than the best split so far.

```
class Node:

    def __init__(self, Xsub, ysub, ID, depth = 0, parent_ID = None, leaf = True):
        self.ID = ID
        self.Xsub = Xsub
        self.ysub = ysub
        self.size = len(ysub)
        self.depth = depth
        self.parent_ID = parent_ID
        self.leaf = leaf

class Splitter:

    def __init__(self):
        self.rss_reduction = 0
        self.no_split = True

    def _replace_split(self, rss_reduction, d, dtype = 'quant', t = None, L_values =
None):
        self.rss_reduction = rss_reduction
        self.d = d
        self.dtype = dtype
        self.t = t
        self.L_values = L_values
        self.no_split = False
```

3. Main Class

We are now ready to build our main class, the decision tree regressor. The class serves two primary functions: training the model and forming predictions with a trained model. The training is comprised of four methods: fit(), _build(), _find_split(), and _make_split(). These methods are covered next.

- fit(): After instantiating a decision tree object, the user provides training data and calls the fit() method. This is the only training method that the user directly calls. The method first records the data and regularization parameters. Then it instantiates a dictionary to store the nodes, called nodes_dict, in which nodes are indexed by their IDs. Finally, it calls the _build() method to build the tree.
- _build(): As its name suggests, the _build() method actually builds our tree. It relies on the _find_split() and _make_split() methods discussed next. The method iterates through layers of the tree (starting with just the initial node), splitting each eligible bud before proceeding to the next layer. The eligible buds are tracked by the eligible_buds dictionary. A bud is eligible for splitting if it does not already have children (i.e. is a leaf); if it is not smaller than min_size, a regularization parameter provided by the user; if its observations are not identical across all predictors; and if it has more than one unique value of the target variable. For each eligible bud, we find a split and make a split, as discussed below. This process continues until there are no eligible buds or we have reached the tree's maximum depth, determined by the argument max_depth.
- _find_split(): When looping through eligible buds in the _build() method, we determine their splits with the _find_split() method. This method instantaites an object of the Splitter class described above. It then loops through all predictors and all possible splits of that predictor to determine which split most reduces the bud's RSS. If the predictor is quantitative, we loop through each unique value and calculate the RSS reduction from splitting at that threshold. If the predictor is categorical, we first call sort_x_by_y() to order the categories of x, and then calculate the RSS reduction for the following splits, where the ordered categories are given by $c_1, \dots, c_V$.

$$
\begin{aligned}
\{c_1\} &\text{ vs. } \{c_2, \dots, c_V\} \\
\{c_1, c_2\} &\text{ vs. } \{c_3, \dots, c_V\} \\
&\dots \\
\{c_1, \dots c_{V-1}\} &\text{ vs. } \{c_V\}.
\end{aligned}
$$

- _make_split(): Once identified, splits are actually conducted with the _make_split() method. This method updates the parent node with the split information and creates the two child nodes. For the parent node, we record which predictor was used to make the split and how. We also add the IDs of the child nodes. For each child node, we record the training observations passing through the node, its ID, its parent's ID, and its size and depth

Finally, we use our built tree to form predictions. This is a two step process. First, we use the `_get_leaf_means()` method to calculate the average of the target variable among training observations landing in each leaf. Then we use `predict()` to run each test observation through the tree and return a fitted value: the mean target variable in the corresponding leaf. Note that the user only directly calls `predict()`, which itself calls `_get_leaf_means()`.

Note

Note that the `_fit()` and `_find_split()` methods refer to an argument `C`. This parameter is used by random forests, which are discussed in chapter 6. This argument can be ignored for the purposes of building a decision tree.

```
class DecisionTreeRegressor:

    #############################
    ######## 1. TRAINING ########
    #############################

    ######### FIT ##########
    def fit(self, X, y, max_depth = 100, min_size = 2, C = None):

        ## Add data
        self.X = X
        self.y = y
        self.N, self.D = self.X.shape
        dtypes = [np.array(list(self.X[:,d])).dtype for d in range(self.D)]
        self.dtypes = ['quant' if (dtype == float or dtype == int) else 'cat' for
dtype in dtypes]

        ## Add regularization parameters
        self.max_depth = max_depth
        self.min_size = min_size
        self.C = C

        ## Initialize nodes
        self.nodes_dict = {}
        self.current_ID = 0
        initial_node = Node(Xsub = X, ysub = y, ID = self.current_ID, parent_ID =
None)
        self.nodes_dict[self.current_ID] = initial_node
        self.current_ID += 1

        ## Build
        self._build()

    ###### BUILD TREE ######
    def _build(self):

        eligible_buds = self.nodes_dict
        for layer in range(self.max_depth):

            ## Find eligible nodes for layer iteration
            eligible_buds = {ID:node for (ID, node) in self.nodes_dict.items() if
                                (node.leaf == True) &
                                (node.size >= self.min_size) &
                                (~all_rows_equal(node.Xsub)) &
                                (len(np.unique(node.ysub)) > 1)}
            if len(eligible_buds) == 0:
                break

            ## split each eligible parent
            for ID, bud in eligible_buds.items():

                ## Find split
                self._find_split(bud)

                ## Make split
                if not self.splitter.no_split: # could be no split for Random Forest
                    self._make_split()


    ###### FIND SPLIT ######
    def _find_split(self, bud):

        ## Instantiate splitter
        splitter = Splitter()
        splitter.bud_ID = bud.ID

        ## Gather eligible predictors (for Random Forests)
        if self.C is None:
            eligible_predictors = np.arange(self.D)
        else:
            eligible_predictors = np.random.choice(np.arange(self.D), self.C, replace =
False)

        ## For each (eligible) predictor...
        for d in sorted(eligible_predictors):
            Xsub_d = bud.Xsub[:,d]
            dtype = self.dtypes[d]
            if len(np.unique(Xsub_d)) == 1:
                continue

            ## For each threshold value...
            if dtype == 'quant':
                for t in np.unique(Xsub_d)[:-1]:
                    ysub_L = bud.ysub[Xsub_d <= t]
                    ysub_R = bud.ysub[Xsub_d > t]
                    rss_reduction = RSS_reduction(ysub_L, ysub_R, bud.ysub)
                    if rss_reduction > splitter.rss_reduction:
                        splitter._replace_split(rss_reduction, d, dtype = 'quant', t =
t)
            else:
                ordered_x = sort_x_by_y(Xsub_d, bud.ysub)
                for i in range(len(ordered_x) - 1):
                    L_values = ordered_x[:i+1]
                    ysub_L = bud.ysub[np.isin(Xsub_d, L_values)]
                    ysub_R = bud.ysub[~np.isin(Xsub_d, L_values)]
                    rss_reduction = RSS_reduction(ysub_L, ysub_R, bud.ysub)
                    if rss_reduction > splitter.rss_reduction:
                        splitter._replace_split(rss_reduction, d, dtype = 'cat',
L_values = L_values)

        ## Save splitter
        self.splitter = splitter

    ###### MAKE SPLIT ######
    def _make_split(self):
        ## Update parent node
        parent_node = self.nodes_dict[self.splitter.bud_ID]
        parent_node.leaf = False
        parent_node.child_L = self.current_ID
        parent_node.child_R = self.current_ID + 1
        parent_node.d = self.splitter.d
        parent_node.dtype = self.splitter.dtype
        parent_node.t = self.splitter.t
        parent_node.L_values = self.splitter.L_values

        ## Get X and y data for children
        if parent_node.dtype == 'quant':
            L_condition = parent_node.Xsub[:,parent_node.d] <= parent_node.t

        else:
            L_condition = np.isin(parent_node.Xsub[:,parent_node.d],
parent_node.L_values)
        Xchild_L = parent_node.Xsub[L_condition]
        ychild_L = parent_node.ysub[L_condition]
        Xchild_R = parent_node.Xsub[~L_condition]
        ychild_R = parent_node.ysub[~L_condition]


        ## Create child nodes
        child_node_L = Node(Xchild_L, ychild_L, depth = parent_node.depth + 1,
                            ID = self.current_ID, parent_ID = parent_node.ID)
        child_node_R = Node(Xchild_R, ychild_R, depth = parent_node.depth + 1,
                            ID = self.current_ID+1, parent_ID = parent_node.ID)
        self.nodes_dict[self.current_ID] = child_node_L
        self.nodes_dict[self.current_ID + 1] = child_node_R
        self.current_ID += 2



    #############################
    ####### 2. PREDICTING #######
    #############################

    ###### LEAF MEANS ######
    def _get_leaf_means(self):
        self.leaf_means = {}
        for node_ID, node in self.nodes_dict.items():
            if node.leaf:
                self.leaf_means[node_ID] = node.ysub.mean()


    ####### PREDICT ########
    def predict(self, X_test):

        ## Calculate leaf means
        self._get_leaf_means()

        yhat = []
        for x in X_test:
            node = self.nodes_dict[0]
            while not node.leaf:
                if node.dtype == 'quant':
                    if x[node.d] <= node.t:
                        node = self.nodes_dict[node.child_L]
                    else:
                        node = self.nodes_dict[node.child_R]
                else:
                    if x[node.d] in node.L_values:
                        node = self.nodes_dict[node.child_L]
                    else:
                        node = self.nodes_dict[node.child_R]
            yhat.append(self.leaf_means[node.ID])
        return np.array(yhat)
```

A regression tree is built below on the `tips` dataset. We also plot the target variable from the test observations against their fitted values.

```
## Build model
tree = DecisionTreeRegressor()
tree.fit(X_train, y_train, max_depth = 7, min_size = 5)
y_test_hat = tree.predict(X_test)

## Visualize predictions
if __name__ == '__main__':
    fig, ax = plt.subplots(figsize = (7, 5))
    sns.scatterplot(y_test, tree.predict(X_test))
    ax.set(xlabel = r'$y$', ylabel = r'$\hat{y}$', title = r'Test Sample $y$ vs.
$\hat{y}$')
    sns.despine()
```

../../_images/regression_tree_14_0.png

Classification Trees

The construction of a classification tree is very similar to that of a regression tree. For a fuller description of the code below, please see the regression tree code on the previous page.

```
## Import packages
import numpy as np
from itertools import combinations
import matplotlib.pyplot as plt
import seaborn as sns

## Load data
penguins = sns.load_dataset('penguins')
penguins.dropna(inplace = True)
X = np.array(penguins.drop(columns = 'species'))
y = np.array(penguins['species'])

## Train-test split
np.random.seed(123)
test_frac = 0.25
test_size = int(len(y)*test_frac)
test_idxs = np.random.choice(np.arange(len(y)), test_size, replace = False)
X_train = np.delete(X, test_idxs, 0)
y_train = np.delete(y, test_idxs, 0)
X_test = X[test_idxs]
y_test = y[test_idxs]
```

We will build our classification tree on the penguins dataset from `seaborn`. This dataset has a categorical target variable—penguin breed—with both quantitative and categorical predictors.

1. Helper Functions

Let's first create our loss functions. The Gini index and cross-entropy calculate the loss for a single node while the `split_loss()` function creates the weighted loss of a split.

```
## Loss Functions
def gini_index(y):
    size = len(y)
    classes, counts = np.unique(y, return_counts = True)
    pmk = counts/size
    return np.sum(pmk*(1-pmk))

def cross_entropy(y):
    size = len(y)
    classes, counts = np.unique(y, return_counts = True)
    pmk = counts/size
    return -np.sum(pmk*np.log2(pmk))

def split_loss(child1, child2, loss = cross_entropy):
    return (len(child1)*loss(child1) + len(child2)*loss(child2))/(len(child1) +
len(child2))
```

Next, let's define a few miscellaneous helper functions. As in the regression tree construction, `all_rows_equal()` checks if all of a bud's rows (observations) are equal across all predictors. If this is the case, this bud will not be split and instead becomes a terminal leaf. The second function, `possible_splits()`, returns all possible ways to divide the classes in a categorical predictor into two. Specifically, it returns all possible sets of values which can be used to funnel observations into the "left" child node. An example is given below for a predictor with four categories, a through d. The set $\{a, b\}$, for instance, would imply observations where that predictor equals a or b go to the left child and other observations go to the right child. (Note that this function requires the `itertools` package).

```python
## Helper Functions
def all_rows_equal(X):
    return (X == X[0]).all()

def possible_splits(x):
    L_values = []
    for i in range(1, int(np.floor(len(x)/2)) + 1):
        L_values.extend(list(combinations(x, i)))
    return L_values

possible_splits(['a','b','c','d'])
```

```
[('a',),
 ('b',),
 ('c',),
 ('d',),
 ('a', 'b'),
 ('a', 'c'),
 ('a', 'd'),
 ('b', 'c'),
 ('b', 'd'),
 ('c', 'd')]
```

2. Helper Classes

Next, we define two classes to help our main decision tree classifier. These classes are essentially identical to those discussed in the regression tree page. The only difference is the loss function used to evaluate a split.

```python
class Node:

    def __init__(self, Xsub, ysub, ID, obs, depth = 0, parent_ID = None, leaf = True):
        self.Xsub = Xsub
        self.ysub = ysub
        self.ID = ID
        self.obs = obs
        self.size = len(ysub)
        self.depth = depth
        self.parent_ID = parent_ID
        self.leaf = leaf


class Splitter:

    def __init__(self):
        self.loss = np.inf
        self.no_split = True

    def _replace_split(self, Xsub_d, loss, d, dtype = 'quant', t = None, L_values =
None):
        self.loss = loss
        self.d = d
        self.dtype = dtype
        self.t = t
        self.L_values = L_values
        self.no_split = False
        if dtype == 'quant':
            self.L_obs = self.obs[Xsub_d <= t]
            self.R_obs = self.obs[Xsub_d > t]
        else:
            self.L_obs = self.obs[np.isin(Xsub_d, L_values)]
            self.R_obs = self.obs[~np.isin(Xsub_d, L_values)]
```

3. Main Class

Finally, we create the main class for our classification tree. This again is essentially identical to the regression tree class. In addition to differing in the loss function used to evaluate splits, this tree differs from the regression tree in how it forms predictions. In regression trees, the fitted value for a test observation was the average target variable of the training observations landing in the same leaf. In the classification tree, since our target variable is categorical, we instead use the most common class among training observations landing in the same leaf.

```
class DecisionTreeClassifier:

    #############################
    ######## 1. TRAINING ########
    #############################

    ######### FIT ##########
    def fit(self, X, y, loss_func = cross_entropy, max_depth = 100, min_size = 2, C = 
None):

        ## Add data
        self.X = X
        self.y = y
        self.N, self.D = self.X.shape
        dtypes = [np.array(list(self.X[:,d])).dtype for d in range(self.D)]
        self.dtypes = ['quant' if (dtype == float or dtype == int) else 'cat' for
dtype in dtypes]

        ## Add model parameters
        self.loss_func = loss_func
        self.max_depth = max_depth
        self.min_size = min_size
        self.C = C

        ## Initialize nodes
        self.nodes_dict = {}
        self.current_ID = 0
        initial_node = Node(Xsub = X, ysub = y, ID = self.current_ID, obs = 
np.arange(self.N), parent_ID = None)
        self.nodes_dict[self.current_ID] = initial_node
        self.current_ID += 1

        # Build
        self._build()

    ###### BUILD TREE ######
    def _build(self):

        eligible_buds = self.nodes_dict
        for layer in range(self.max_depth):

            ## Find eligible nodes for layer iteration
            eligible_buds = {ID:node for (ID, node) in self.nodes_dict.items() if
                                (node.leaf == True) &
                                (node.size >= self.min_size) &
                                (~all_rows_equal(node.Xsub)) &
                                (len(np.unique(node.ysub)) > 1)}
            if len(eligible_buds) == 0:
                break

            ## split each eligible parent
            for ID, bud in eligible_buds.items():

                ## Find split
                self._find_split(bud)

                ## Make split
                if not self.splitter.no_split:
                    self._make_split()

    ###### FIND SPLIT ######
    def _find_split(self, bud):

        ## Instantiate splitter
        splitter = Splitter()
        splitter.bud_ID = bud.ID
        splitter.obs = bud.obs

        ## For each (eligible) predictor...
        if self.C is None:
            eligible_predictors = np.arange(self.D)
        else:
            eligible_predictors = np.random.choice(np.arange(self.D), self.C, replace = 
False)
        for d in sorted(eligible_predictors):
            Xsub_d = bud.Xsub[:,d]
            dtype = self.dtypes[d]
            if len(np.unique(Xsub_d)) == 1:
                continue

            ## For each value...
            if dtype == 'quant':
                for t in np.unique(Xsub_d)[:-1]:
                    ysub_L = bud.ysub[Xsub_d <= t]
                    ysub_R = bud.ysub[Xsub_d > t]
                    loss = split_loss(ysub_L, ysub_R, loss = self.loss_func)
                    if loss < splitter.loss:
                        splitter._replace_split(Xsub_d, loss, d, 'quant', t = t)
            else:
                for L_values in possible_splits(np.unique(Xsub_d)):
                    ysub_L = bud.ysub[np.isin(Xsub_d, L_values)]
                    ysub_R = bud.ysub[~np.isin(Xsub_d, L_values)]
                    loss = split_loss(ysub_L, ysub_R, loss = self.loss_func)
                    if loss < splitter.loss:
                        splitter._replace_split(Xsub_d, loss, d, 'cat', L_values =
L_values)

        ## Save splitter
        self.splitter = splitter

    ###### MAKE SPLIT ######
    def _make_split(self):

        ## Update parent node
        parent_node = self.nodes_dict[self.splitter.bud_ID]
        parent_node.leaf = False
        parent_node.child_L = self.current_ID
        parent_node.child_R = self.current_ID + 1
        parent_node.d = self.splitter.d
        parent_node.dtype = self.splitter.dtype
        parent_node.t = self.splitter.t
        parent_node.L_values = self.splitter.L_values
        parent_node.L_obs, parent_node.R_obs = self.splitter.L_obs, self.splitter.R_obs

        ## Get X and y data for children
        if parent_node.dtype == 'quant':
            L_condition = parent_node.Xsub[:,parent_node.d] <= parent_node.t
        else:
            L_condition = np.isin(parent_node.Xsub[:,parent_node.d],
parent_node.L_values)
        Xchild_L = parent_node.Xsub[L_condition]
        ychild_L = parent_node.ysub[L_condition]
        Xchild_R = parent_node.Xsub[~L_condition]
        ychild_R = parent_node.ysub[~L_condition]

        ## Create child nodes
        child_node_L = Node(Xchild_L, ychild_L, obs = parent_node.L_obs, depth =
parent_node.depth + 1,
                            ID = self.current_ID, parent_ID = parent_node.ID)
        child_node_R = Node(Xchild_R, ychild_R, obs = parent_node.R_obs, depth =
parent_node.depth + 1,
                            ID = self.current_ID+1, parent_ID = parent_node.ID)
        self.nodes_dict[self.current_ID] = child_node_L
        self.nodes_dict[self.current_ID + 1] = child_node_R
        self.current_ID += 2


    #############################
    ####### 2. PREDICTING #######
    #############################

    ###### LEAF MODES ######
    def _get_leaf_modes(self):
        self.leaf_modes = {}
        for node_ID, node in self.nodes_dict.items():
            if node.leaf:
                values, counts = np.unique(node.ysub, return_counts=True)
                self.leaf_modes[node_ID] = values[np.argmax(counts)]

    ####### PREDICT ########
    def predict(self, X_test):

        # Calculate leaf modes
        self._get_leaf_modes()

        yhat = []
        for x in X_test:
            node = self.nodes_dict[0]
            while not node.leaf:
                if node.dtype == 'quant':
                    if x[node.d] <= node.t:
                        node = self.nodes_dict[node.child_L]
                    else:
                        node = self.nodes_dict[node.child_R]
                else:
                    if x[node.d] in node.L_values:
                        node = self.nodes_dict[node.child_L]
                    else:
                        node = self.nodes_dict[node.child_R]
            yhat.append(self.leaf_modes[node.ID])
        return np.array(yhat)
```

A classificaiton tree is built on the penguins dataset. We evaluate the predictions on a test set and find that roughly 95% of observations are correctly classified.

```python
## Build classifier
tree = DecisionTreeClassifier()
tree.fit(X_train, y_train, max_depth = 10, min_size = 10)
y_test_hat = tree.predict(X_test)

## Evaluate on test data
np.mean(y_test_hat == y_test)
```

```
0.9518072289156626
```

Implementation

This section demonstrates how to fit regression and decision trees with `scikit-learn`. We will again use the tips dataset for the regression tree and the penguins dataset for the classification tree.

```python
## Import packages
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

1. Regression Tree

Let's start by loading in the data. We'll keep things as `pandas` dataframes rather than `numpy` arrays now.

```python
## Load tips data
tips = sns.load_dataset('tips')
X = tips.drop(columns = 'tip')
y = tips['tip']

## Train-test split
np.random.seed(1)
test_frac = 0.25
test_size = int(len(y)*test_frac)
test_idxs = np.random.choice(np.arange(len(y)), test_size, replace = False)
X_train = X.drop(test_idxs)
y_train = y.drop(test_idxs)
X_test = X.loc[test_idxs]
y_test = y.loc[test_idxs]
```

We can then fit the regression tree with the `DecisionTreeRegressor` class. Unfortunately, `scikit-learn` does not currently support categorical predictors. Instead, we have to first convert these predictors to dummy variables. Note that this implies that splits of categorical variables can only separate one value from the rest. For instance, a variable with discrete values a, b, c, and d could not be split as $\{a, b\}$ versus $\{c, d\}$.

```python
from sklearn.tree import DecisionTreeRegressor

## Get dummies
X_train = pd.get_dummies(X_train, drop_first = True)
X_test = pd.get_dummies(X_test, drop_first = True)

## Build model
dtr = DecisionTreeRegressor(max_depth = 7, min_samples_split = 5)
dtr.fit(X_train, y_train)
y_test_hat = dtr.predict(X_test)

## Visualize predictions
fig, ax = plt.subplots(figsize = (7, 5))
sns.scatterplot(y_test, y_test_hat)
ax.set(xlabel = r'$y$', ylabel = r'$\hat{y}$', title = r'Test Sample $y$ vs.
$\hat{y}$')
sns.despine()
```

../../_images/code_7_02.png

2. Classification Tree

The classification tree implementation in `scikit-learn` is nearly identical. The corresponding code is provided below.

```
## Load penguins data
penguins = sns.load_dataset('penguins')
penguins = penguins.dropna().reset_index(drop = True)
X = penguins.drop(columns = 'species')
y = penguins['species']

## Train-test split
np.random.seed(1)
test_frac = 0.25
test_size = int(len(y)*test_frac)
test_idxs = np.random.choice(np.arange(len(y)), test_size, replace = False)
X_train = X.drop(test_idxs)
y_train = y.drop(test_idxs)
X_test = X.loc[test_idxs]
y_test = y.loc[test_idxs]
```

```
from sklearn.tree import DecisionTreeClassifier

## Get dummies
X_train = pd.get_dummies(X_train, drop_first = True)
X_test = pd.get_dummies(X_test, drop_first = True)

## Build model
dtc = DecisionTreeClassifier(max_depth = 10, min_samples_split = 10)
dtc.fit(X_train, y_train)
y_test_hat = dtc.predict(X_test)

## Observe Accuracy
np.mean(y_test_hat == y_test)
```

```
0.9036144578313253
```

Concept

Due to their high variance, decision trees often fail to reach a level of precision comparable to other predictive algorithms. In the previous chapter, we introduced several ways to minimize the variance of a single decision tree, such as through pruning or direct size regulation. This chapter discusses another approach: *ensemble methods*. Ensemble methods combine the output of multiple simple models, often called "learners", in order to create a final model with lower variance.

We will introduce ensemble methods in the context of tree-based learners, though ensemble methods can be applied to a wide range of learning algorithms. That said, the structure of decision trees makes ensemble methods particularly valuable. Here we discuss three tree-based ensemble methods: *bagging*, *random forests*, and *boosting*.

An example demonstrating the power of ensemble methods is given below. Using the `tips` dataset from `scikit-learn`, we build several tree-based learners. Then for $b = 1, 2, \dots, 30$, we use bagging to average the results of the b learners. For each bagged model, we also calculate the out-of-sample RSS. The blue line below shows the RSS for each bagged model—which clearly decreases with b—and the red line shows the RSS for a single decision tree. By averaging many trees, rather than relying on a single one, we are able to improve the precision of our model.

Bagging

Bagging, short for *bootstrap aggregating*, combines the results of several learners trained on bootstrapped samples of the training data. The process of bagging is very simple yet often quite powerful.

Bootstrapping

Bootstrapping is commonly used in statistical problems to approximate the variance of an estimator. Suppose we observe dataset $\mathcal{D}$ and we are interested in some parameter θ (such as the mean or median). Estimating θ is one thing, but understanding the *variance* of that estimate is another. For difficult problems, the bootstrap helps find this variance.

The variance of an estimator tells us how much it would vary from sample to sample. Suppose we drew B datasets of equal size, $\mathcal{D}_1$ through $\mathcal{D}_B$, and each time estimated of θ. We would then be able to calculate the variance of our estimate since we have B observed estimates. In practice however, we are stuck with only one dataset. The bootstrap enables us to approximate the variance of our estimate with only one dataset. This procedure is outlined below.

Bootstrap Procedure

Suppose we have a dataset $\mathcal{D} = \{x_1, x_2, \dots, x_N\}$ of size N. Suppose also that we choose to estimate the parameter θ with some function of the data, $\hat{\theta} = f(\mathcal{D})$. For some large value B, do the following.

1. For $b = 1, 2, \dots, B$,
    1. Sample N observations from $\mathcal{D}$ *with* replacement. Call this sample $\mathcal{D}^*_b$.
    2. Estimate θ on the bootstrapped sample, $\hat{\theta}^*_b = f(\mathcal{D}^*_b)$.
2. Calculate the sample variance of the bootstrapped estimates, $\{\hat{\theta}^*_1, \dots, \hat{\theta}^*_B\}$. Let this be our estimate of the variance of $\hat{\theta}$.

The same bootstrap procedure can be extended to serve a very different purpose. Bagging uses the bootstrap to *reduce*—rather than *estimate*—the variance of a model.

Bagging for Decision Trees

A bagging model trains many learners on bootstrapped samples of the training data and aggregates the results into one final model. We will outline this procedure for decision trees, though the approach works for other learners.

Bagging Procedure

Given a training dataset $\mathcal{D} = \{\mathbf{x}_n, y_n\}_{n=1}^N$ and a separate test set $\mathcal{T} = \{\mathbf{x}_t\}_{t=1}^T$, we build and deploy a bagging model with the following procedure. The first step builds the model (the learners) and the second generates fitted values.

1. For $b = 1, 2, \dots, B$,
    1. Draw a bootstrapped sample $\mathcal{D}^*_b$.
    2. Build a decision tree T^*_b to the bootstrapped sample.
2. For test observation $t = 1, 2, \dots, T$,
    1. For $b = 1, 2, \dots, B$,
        1. Calculate $\hat{y}^*_{tb}$, observation t's fitted value according to tree T^*_b.
    2. Combine the $\hat{y}^*_{tb}$ into a single fitted value, $\hat{y}_t$.

How exactly we combine the results of the learners into a single fitted value (the second part of the second step) depends on the target variable. For a continuous target variable, we typically average the learners' predictions. For a categorical target variable, we typically use the class that receives the plurality vote.

To recap bagging, we simply build a large number of learners on bootstrapped samples of the data and combine the predictions of these learners into one single fitted value. As if creating new datasets, the bootstrap provides our model with more training samples in order to improve its estimates.

Random Forests

A random forest is a slight extension to the bagging approach for decision trees that can further decrease overfitting and improve out-of-sample precision. Unlike bagging, random forests are exclusively designed for decision trees (hence the name).

Like bagging, a random forest combines the predictions of several base learners, each trained on a bootstrapped sample of the original training set. Random forests, however, add one additional regulatory step: at each split within each tree, we only consider splitting a randomly-chosen subset of the predictors. In other words, we explicitly prohibit the trees from considering some of the predictors in each split.

It might seem counter-intuitive to restrict the amount of information available to our model while training. Recall that the advantage of bootstrapping is to provide the model with a greater sense of the training data by "generating" different datasets. If the base learners are too similar, however, we don't gain anything by averaging them. Randomly choosing eligible predictors at each split *de-correlates* the trees, serving to further differentiate the trees from one another. We then gain more information by averaging over trees which can result in significant improvements in precision.

The specific steps to building a random forest are provided below.

Random Forest Procedure

Suppose we have dataset $\mathcal{D} = \{\mathbf{x}_n, y_n\}_{n=1}^N$ and test set $\mathcal{T} = \{\mathbf{x}_t\}_{t=1}^T$ where the predictors are D-dimensional (i.e. $\mathbf{x} \in \mathbb{R}^D$).

1. For $b = 1, 2, \dots, B$,
    1. Draw a bootstrapped sample $\mathcal{D}_b^*$.
    2. Build a decision tree T_b^* to the bootstrapped sample. At each split, only consider splitting along C of the D predictors (with $C \leq D$).
2. For test observation $t = 1, 2, \dots, T$,
    1. For $b = 1, 2, \dots, B$,
        1. Calculate $\hat{y}_{tb}^*$, observation t's fitted value according to tree T_b^*.
    2. Combine the $\hat{y}_{tb}^*$ into a single fitted value, $\hat{y}_t$.

The only difference between the random forest procedure above and the bagging procedure in the previous section is the restricted use of predictors in the second part of the first step. If $C = D$, there is no difference between these two approaches.

To recap, random forests average the results of several decision trees and add two sources of randomness to ensure differentiation between the base learners: randomness in which observations are sampled via the bootstrapping and randomness in which predictors are considered at each split.

Boosting

Like bagging and random forests, boosting combines multiple weak learners into one improved model. Unlike bagging and random forests, however, boosting trains these weak learners *sequentially*, each one learning from the mistakes of the last. Rather than a single model, "boosting" refers to a class of sequential learning methods.

Here we discuss two specific algorithms for conducting boosting. The first, known as *Discrete AdaBoost* (or just *AdaBoost*), is used for binary classification. The second, known as *AdaBoost.R2*, is used for regression.

AdaBoost for Binary Classification

Weighted Classification Trees

Weak learners in a boosting model learn from the mistakes of previous iterations by increasing the *weights* of observations that previous learners struggled with. How exactly we fit a *weighted* learner depends on the type of learner. Fortunately, for classification trees this can be done with just a slight adjustment to the loss function.

Consider a classification tree with classes $k = 1, \dots, K$ and a node $\mathcal{N}_m$. Let $\hat{p}_{mk}$ give the fraction of the observations in $\mathcal{N}_m$ belonging to class k. Recall that the Gini index is defined as

$$\mathcal{L}_G(\mathcal{N}_m) = \sum_{k=1}^K \hat{p}_{mk}(1 - \hat{p}_{mk}),$$

and the cross entropy as

$$\mathcal{L}_E(\mathcal{N}_m) = -\sum_{k=1}^K \hat{p}_{mk} \log \hat{p}_{mk},$$

The classification tree then chooses the split S_m that minimizes the combined loss of the child nodes,

$$\mathcal{L}(S_m) = f_L \cdot \mathcal{L}(\mathcal{C}_m^L) + f_R \cdot \mathcal{L}(\mathcal{C}_m^R),$$

where C_m^L and C_m^R are the left and right child nodes and f_L and f_R are the fraction of observations going to each.

To allow for observation weighting, we simply change our definition of $\hat{p}_{mk}$ to the following:

$$\hat{p}_{mk} = \frac{\sum_{n \in \mathcal{N}_m} w_n I(y_n = k)}{\sum_{n \in \mathcal{N}_m} w_n}.$$

This forces the tree to prioritize isolating the classes that were poorly classified by previous learners. The Gini index and cross-entropy are then defined as before and the rest of the tree is fit in the same way.

The Discrete AdaBoost Algorithm

The *Discrete AdaBoost* algorithm is outlined below. We start by setting the weights equal for each observation. Then we build T weighted weak learners (decision trees in our case). Each observation's weight is updated according to two factors: the strength of the iteration's learner and the accuracy of the learner on that observation. Specifically, the update for the weight on observation n at iteration t takes the form

$$w_n^{t+1} = w_n \exp(\alpha^t I_n^t),$$

where α^t is a measure of the accuracy of the learner and I_n^t indicates the learner misclassified observation n. This implies that the weight for a correctly-classified observation does not change while the weight for a misclassified observation increases, particularly if the model is accurate.

> **Note**
>
> Notation for boosting can get confusing since we iterate over learners and through observations. For the rest of this page, let superscripts refer to the model iteration and subscripts refer to the observation. For instance, w_n^t refers to the weight of observation n in learner t.

Discrete AdaBoost Algorithm

Define the target variable to be $y_n \in \{-1, +1\}$.

1. Initialize the weights with $w_n^1 = \frac{1}{N}$ for $n = 1, 2, \dots, N$.
2. For $t = 1, \dots, T$,
    - Build weak learner t using weights $\mathbf{w}^t$.
    - Use weak learner t to calculate fitted values $f^t(\mathbf{x}_n) \in \{-1, +1\}$ for $n = 1, 2, \dots, N$. Let I_n^t equal 1 If $f^t(\mathbf{x}_n) \neq y_n$ and 0 otherwise. That is, I_n^t indicates whether learner t misclassifies observation n.
    - Calculate the weighted error for learner t:

    $$\epsilon^t = \frac{\sum_{n=1}^N w_n^t I_n^t}{\sum_{n=1}^N w_n^t}.$$

    - Calculate the accuracy measure for learner t:

    $$\alpha^t = \log\left(\frac{1-\epsilon^t}{\epsilon^t}\right).$$

    - Update the weighting with

    $$w_n^{t+1} = w_n^t \exp(\alpha^t I_n^t),$$

    for $n = 1, 2, \dots, N$.
3. Calculate the overall fitted values with $\hat{y}_n = \text{sign}\left(\sum_{t=1}^T \alpha^t f^t(\mathbf{x}_n)\right)$.

Note that our final fitted value for observation n, $\hat{y}_n$, is a weighted vote of all the individual fitted values where higher-performing learners are given more weight.

AdaBoost For Regression

We can apply the same principle of learning from our mistakes to regression tasks as well. A common algorithm for doing so is *AdaBoost.R2*, shown below. Like *AdaBoost*, this algorithm uses weights to emphasize observations that previous learners struggled with. Unlike *AdaBoost*, however, it does not incorporate these weights into the loss function directly. Instead, every iteration, it draws bootstrap samples from the training data where observations with greater weights are more likely to be drawn.

We then fit a weak learner to the bootstrapped sample, calculate the fitted values on the *original sample* (i.e. *not* the bootstrapped sample), and use the residuals to assess the quality of the weak learner. As in *AdaBoost*, we update the weights each iteration based on the quality of the learner (determined by β^t) and the accuracy of the learner on the observation (determined by L_n).

After iterating through many weak learners, we form our fitted values. Specifically, for each observation we use the weighted median (defined below) of the weak learners' predictions, weighted by $\log(1/\beta^t)$ for $t = 1, \dots, T$.

> **Note**
>
> **Weighted Median**
>
> Consider a set of values $\{v_1, v_2, \dots, v_N\}$ and a set of corresponding weights $\{s_1, s_2, \dots, s_N\}$. Normalize the weights so they add to 1. To calculate the weighted median, sort the values and weights in ascending order of the values; call these $\{v^{(1)}, v^{(2)}, \dots, v^{(N)}\}$ and $\{s^{(1)}, s^{(2)}, \dots, s^{(N)}\}$ where, for instance, $v^{(1)}$ is the smallest value and $s^{(1)}$ is its corresponding weight. The weighted median is then the value corresponding to the first weight such that the cumulative sum of the weights is greater than or equal to 0.5.
>
> As an example, consider the values $\mathbf{v} = \{10, 30, 20, 40\}$ and corresponding weights $\mathbf{s} = \{0.4, 0.6, 0.2, 0.8\}$. First normalize the weights to sum to 1, giving $\mathbf{s} = \{0.2, 0.3, 0.1, 0.4\}$. Then sort the values and the weights, giving $\mathbf{v} = \{10, 20, 30, 40\}$ and $\mathbf{s} = \{0.2, 0.1, 0.3, 0.4\}$. Then, since the third element is the first for which the cumulative sum of the weights is at least 0.5, the weighted median is the third of the sorted values, 30.

Though the arithmetic below gets somewhat messy, the concept is simple: iteratively fit a weak learner, see where the learner struggles, and emphasize the observations where it failed (where the amount of emphasis depends on the overall strength of the learner).

AdaBoost.R2 Algorithm

1. Initialize the weights with $w_n^1 = \frac{1}{N}$ for $n = 1, 2, \dots, N$.
2. For $t = 1, 2, \dots, T$ or while $\bar{L}^t$, as defined below, is less than or equal to 0.5,
    - Draw a sample of size N from the training data with replacement and with probability w_n^t for $n = 1, 2, \dots, N$.
    - Fit weak learner t to the resampled data and calculate the fitted values on the original dataset. Denote these fitted values with $f^t(\mathbf{x}_n)$ for $n = 1, 2, \dots, N$.
    - Calculate the observation error L_n^t for $n = 1, 2, \dots, N$:

$$\begin{aligned} D^t &= \max_n \{|y_n - f^t(\mathbf{x}_n)|\} \\ L_n^t &= \frac{|y_n - f^t(\mathbf{x}_n)|}{D^t} \end{aligned}$$

    - Calculate the model error $\bar{L}^t$:

$$\bar{L}^t = \sum_{n=1}^{N} L_n^t w_n^t$$

      If $\bar{L}^t \geq 0.5$, end iteration and set T equal to $t - 1$.
    - Let $\beta^t = \frac{\bar{L}^t}{1-\bar{L}^t}$. The lower β^t, the greater our confidence in the model.
    - Let $Z^t = \sum_{n=1}^{N} w_n^t (\beta^t)^{1-L_n^t}$ be a normalizing constant and update the model weights with

$$w_n^{t+1} = \frac{w_n^t (\beta^t)^{1-L_n^t}}{Z^t},$$

      which increases the weight for observations with a greater error L_n^t.
3. Set the overall fitted value for observation n equal to the weighted median of $f^t(\mathbf{x}_n)$ for $t = 1, 2, \dots, T$ using weights $\log(1/\beta^t)$ for model t.

In the boosting construction, we implement *AdaBoost.R2* using decision tree regressors though many other weak learners may be used.

Construction

This section demonstrates constructions of bagging models, random forests, and boosting for classification and regression. Each of these relies on the decision tree constructions from the last chapter. Bagging and random forests involve simple combinations of decision trees with only minor adjustments to the tree structure; therefore, the code for those sections is limited. The section on boosting involves more original code.

Bagging

Bagging can be used for regression or classification, though we will demonstrate a regression bagging model here. Since this model is based on decision tree regressors, we'll first import our regression tree construction from the previous chapter. We'll also import numpy and the visualization packages.

```
## Import decision trees
import import_ipynb
import regression_tree as rt;

## Import numpy and visualization packages
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
```

We will build our bagging model on the tips dataset from `scikit-learn`. The hidden code cell below loads that data and does a train-test split.

```
## Load data
tips = sns.load_dataset('tips')
X = np.array(tips.drop(columns = 'tip'))
y = np.array(tips['tip'])

## Train-test split
np.random.seed(1)
test_frac = 0.25
test_size = int(len(y)*test_frac)
test_idxs = np.random.choice(np.arange(len(y)), test_size, replace = False)
X_train = np.delete(X, test_idxs, 0)
y_train = np.delete(y, test_idxs, 0)
X_test = X[test_idxs]
y_test = y[test_idxs]
```

Now we can get right into the bagging class. To fit the `Bagger` object, we provide training data, the number of bootstraps (`B`), and size regulation parameters for the decision trees. The object then takes `B` bootstraps of the data, each time fitting a decision tree regressor. To form predictions with the `Bagger` object, we simply run test observations through each bootstrapped tree and average the fitted values.

```
class Bagger:

    def fit(self, X_train, y_train, B, max_depth = 100, min_size = 2, seed = None):

        self.X_train = X_train
        self.N, self.D = X_train.shape
        self.y_train = y_train
        self.B = B
        self.seed = seed
        self.trees = []

        np.random.seed(seed)
        for b in range(self.B):

            sample = np.random.choice(np.arange(self.N), size = self.N, replace =
True)
            X_train_b = X_train[sample]
            y_train_b = y_train[sample]

            tree = rt.DecisionTreeRegressor()
            tree.fit(X_train_b, y_train_b, max_depth = max_depth, min_size = min_size)
            self.trees.append(tree)


    def predict(self, X_test):

        y_test_hats = np.empty((len(self.trees), len(X_test)))
        for i, tree in enumerate(self.trees):
            y_test_hats[i] = tree.predict(X_test)

        return y_test_hats.mean(0)
```

We can now fit the bagging model and display the observed versus fitted values.

```
## Build model
bagger = Bagger()
bagger.fit(X_train, y_train, B = 30, max_depth = 20, min_size = 5, seed = 123)
y_test_hat = bagger.predict(X_test)

## Plot
fig, ax = plt.subplots(figsize = (7, 5))
sns.scatterplot(y_test, y_test_hat)
ax.set(xlabel = r'$y$', ylabel = r'$\hat{y}$', title = r'Observed vs. Fitted Values for
Bagging')
sns.despine()
```

../../_images/bagging_8_0.png

Random Forests

Fitting the random forest is essentially identical to fitting the bagging model with one exception: we tell the decision trees to consider only a subset of the predictors at each split. The code to do this is actually written in the decision tree construction section.

As in the bagging model, let's import the regression tree construction and load the tips dataset. This is done in the hidden code cell below.

```
## Import decision trees
import import_ipynb
import regression_tree as rt;

## Import numpy and visualization packages
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets

## Load data
tips = sns.load_dataset('tips')
X = np.array(tips.drop(columns = 'tip'))
y = np.array(tips['tip'])

## Train-test split
np.random.seed(1)
test_frac = 0.25
test_size = int(len(y)*test_frac)
test_idxs = np.random.choice(np.arange(len(y)), test_size, replace = False)
X_train = np.delete(X, test_idxs, 0)
y_train = np.delete(y, test_idxs, 0)
X_test = X[test_idxs]
y_test = y[test_idxs]
```

We now build the `RandomForest` class in the same way we built the `Bagger` with one difference: we add the argument `C` which controls the number of predictors the decision trees consider at each split.

```
class RandomForest:

    def fit(self, X_train, y_train, B, C, max_depth = 100, min_size = 2, seed = None):

        self.X_train = X_train
        self.N, self.D = X_train.shape
        self.y_train = y_train
        self.B = B
        self.C = C
        self.seed = seed
        self.trees = []

        np.random.seed(seed)
        for b in range(self.B):

            sample = np.random.choice(np.arange(self.N), size = self.N, replace =
True)
            X_train_b = X_train[sample]
            y_train_b = y_train[sample]

            tree = rt.DecisionTreeRegressor()
            tree.fit(X_train_b, y_train_b, max_depth = max_depth, min_size = min_size,
C = C)
            self.trees.append(tree)


    def predict(self, X_test):

        y_test_hats = np.empty((len(self.trees), len(X_test)))
        for i, tree in enumerate(self.trees):
            y_test_hats[i] = tree.predict(X_test)


        return y_test_hats.mean(0)
```

We can now build a random forest and compare the observed with fitted values, as follows.

```
## Build model
rf = RandomForest()
rf.fit(X_train, y_train, B = 30, C = 4, max_depth = 20, min_size = 5, seed = 123)
y_test_hat = rf.predict(X_test)

## Plot
fig, ax = plt.subplots(figsize = (7, 5))
sns.scatterplot(y_test, y_test_hat)
ax.set(xlabel = r'$y$', ylabel = r'$\hat{y}$', title = r'Random Forest Observed vs.
Fitted Values')
sns.despine()
```

../../_images/random_forests_6_0.png

Boosting

In this section, we will construct a boosting classifier with the `AdaBoost` algorithm and a boosting regressor with the `AdaBoost.R2` algorithm. These algorithms can use a variety of weak learners but we will use decision tree classifiers and regressors, constructed in Chapter 5.

```python
## Import decision trees
import import_ipynb
import classification_tree as ct;

## Import numpy and visualization packages
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
```

1. Classification with AdaBoost

The following is a construction of the binary AdaBoost classifier introduced in the concept section. Let's again use the penguins dataset from `seaborn`, but rather than predicting the penguin's species (a multiclass problem), we'll predict whether the species is *Adelie* (a binary problem). The data is loaded and split into train vs. test with the hidden code cell below.

```python
## Load data
penguins = sns.load_dataset('penguins')
penguins.dropna(inplace = True)
X = np.array(penguins.drop(columns = ['species', 'island']))
y = 1*np.array(penguins['species'] == 'Adelie')
y[y == 0] = -1

## Train-test split
np.random.seed(123)
test_frac = 0.25
test_size = int(len(y)*test_frac)
test_idxs = np.random.choice(np.arange(len(y)), test_size, replace = False)
X_train = np.delete(X, test_idxs, 0)
y_train = np.delete(y, test_idxs, 0)
X_test = X[test_idxs]
y_test = y[test_idxs]
```

Recall that AdaBoost fits *weighted* weak learners. Let's start by defining the weighted loss functions introduced in the concept section. The helper function `get_weighted_pmk()` calculates

$$\hat{p}_{mk} = \frac{\sum_{n=1}^{N} w_n I(\mathbf{x}_n \in \mathcal{N}_m)}{\sum_{n=1}^{N} w_n}$$

for each class k. The `gini_index()` and `cross_entropy()` functions then call this function and return the appropriate loss.

```python
## Loss Functions
def get_weighted_pmk(y, weights):
    ks = np.unique(y)
    weighted_pmk = [sum(weights[y == k]) for k in ks]
    return(np.array(weighted_pmk)/sum(weights))

def gini_index(y, weights):
    weighted_pmk = get_weighted_pmk(y, weights)
    return np.sum( weighted_pmk*(1-weighted_pmk) )

def cross_entropy(y, weights):
    weighted_pmk = get_weighted_pmk(y, weights)
    return -np.sum(weighted_pmk*np.log2(weighted_pmk))

def split_loss(child1, child2, weights1, weights2, loss = cross_entropy):
    return (len(child1)*loss(child1, weights1) + len(child2)*loss(child2,
weights2))/(len(child1) + len(child2))
```

In order to incorporate observation weights, we have to make slight adjustments to the `DecisionTreeClassifier` class. In the class we previously constructed, the data from parent nodes was split and funneled anonymously to one of two child nodes. This alone will not allow us to incorporate weights. Instead, we need to also track the ID of each observation so we can track its weight. This is done with the `DecisionTreeClassifier` class defined in the hidden cell below, which is mostly a reconstruction of the class defined in Chapter 5.

```python
## Helper Classes
class Node:

    def __init__(self, Xsub, ysub, observations, ID, depth = 0, parent_ID = None, leaf
= True):
        self.Xsub = Xsub
        self.ysub = ysub
        self.observations = observations
        self.ID = ID
        self.size = len(ysub)
        self.depth = depth
        self.parent_ID = parent_ID
        self.leaf = leaf


class Splitter:

    def __init__(self):
        self.loss = np.inf
        self.no_split = True

    def _replace_split(self, loss, d, dtype = 'quant', t = None, L_values = None):
        self.loss = loss
        self.d = d
        self.dtype = dtype
        self.t = t
        self.L_values = L_values
        self.no_split = False


## Main Class
class DecisionTreeClassifier:

    #############################
    ######## 1. TRAINING ########
    #############################

    ######### FIT ##########
    def fit(self, X, y, weights, loss_func = cross_entropy, max_depth = 100, min_size =
 2, C = None):

        ## Add data
        self.X = X
        self.y = y
        self.N, self.D = self.X.shape
        dtypes = [np.array(list(self.X[:,d])).dtype for d in range(self.D)]
        self.dtypes = ['quant' if (dtype == float or dtype == int) else 'cat' for
dtype in dtypes]
        self.weights = weights

        ## Add model parameters
        self.loss_func = loss_func
        self.max_depth = max_depth
        self.min_size = min_size
        self.C = C

        ## Initialize nodes
        self.nodes_dict = {}
        self.current_ID = 0
        initial_node = Node(Xsub = X, ysub = y, observations = np.arange(self.N), ID =
self.current_ID, parent_ID = None)
        self.nodes_dict[self.current_ID] = initial_node
        self.current_ID += 1

        # Build
        self._build()

    ###### BUILD TREE ######
    def _build(self):

        eligible_buds = self.nodes_dict
        for layer in range(self.max_depth):

            ## Find eligible nodes for layer iteration
            eligible_buds = {ID:node for (ID, node) in self.nodes_dict.items() if
                                (node.leaf == True) &
                                (node.size >= self.min_size) &
                                (~ct.all_rows_equal(node.Xsub)) &
                                (len(np.unique(node.ysub)) > 1)}
            if len(eligible_buds) == 0:
                break

            ## split each eligible parent
            for ID, bud in eligible_buds.items():

                ## Find split
                self._find_split(bud)
                ## Make split
                if not self.splitter.no_split:
                    self._make_split()

    ###### FIND SPLIT ######
    def _find_split(self, bud):

        ## Instantiate splitter
        splitter = Splitter()
        splitter.bud_ID = bud.ID

        ## For each (eligible) predictor...
        if self.C is None:
            eligible_predictors = np.arange(self.D)
        else:
            eligible_predictors = np.random.choice(np.arange(self.D), self.C, replace =
False)
        for d in sorted(eligible_predictors):
            Xsub_d = bud.Xsub[:,d]
            dtype = self.dtypes[d]
            if len(np.unique(Xsub_d)) == 1:
                continue

            ## For each value...
            if dtype == 'quant':
                for t in np.unique(Xsub_d)[:-1]:
                    L_condition = Xsub_d <= t
                    ysub_L = bud.ysub[L_condition]
                    ysub_R = bud.ysub[~L_condition]
                    weights_L = self.weights[bud.observations][L_condition]
                    weights_R = self.weights[bud.observations][~L_condition]
                    loss = split_loss(ysub_L, ysub_R,
                                      weights_L, weights_R,
                                      loss = self.loss_func)
                    if loss < splitter.loss:
                        splitter._replace_split(loss, d, 'quant', t = t)
            else:
                for L_values in ct.possible_splits(np.unique(Xsub_d)):
                    L_condition = np.isin(Xsub_d, L_values)
                    ysub_L = bud.ysub[L_condition]
                    ysub_R = bud.ysub[~L_condition]
                    weights_L = self.weights[bud.observations][L_condition]
                    weights_R = self.weights[bud.observations][~L_condition]
                    loss = split_loss(ysub_L, ysub_R,
                                      weights_L, weights_R,
                                      loss = self.loss_func)
                    if loss < splitter.loss:
                        splitter._replace_split(loss, d, 'cat', L_values = L_values)

        ## Save splitter
        self.splitter = splitter

    ###### MAKE SPLIT ######
    def _make_split(self):

        ## Update parent node
        parent_node = self.nodes_dict[self.splitter.bud_ID]
        parent_node.leaf = False
        parent_node.child_L = self.current_ID
        parent_node.child_R = self.current_ID + 1
        parent_node.d = self.splitter.d
        parent_node.dtype = self.splitter.dtype
        parent_node.t = self.splitter.t
        parent_node.L_values = self.splitter.L_values

        ## Get X and y data for children
        if parent_node.dtype == 'quant':
            L_condition = parent_node.Xsub[:,parent_node.d] <= parent_node.t
        else:
            L_condition = np.isin(parent_node.Xsub[:,parent_node.d],
parent_node.L_values)
        Xchild_L = parent_node.Xsub[L_condition]
        ychild_L = parent_node.ysub[L_condition]
        child_observations_L = parent_node.observations[L_condition]
        Xchild_R = parent_node.Xsub[~L_condition]
        ychild_R = parent_node.ysub[~L_condition]
        child_observations_R = parent_node.observations[~L_condition]

        ## Create child nodes
        child_node_L = Node(Xchild_L, ychild_L, child_observations_L,
                            ID = self.current_ID, depth = parent_node.depth + 1,
                            parent_ID = parent_node.ID)
        child_node_R = Node(Xchild_R, ychild_R, child_observations_R,
                            ID = self.current_ID + 1, depth = parent_node.depth + 1,
                            parent_ID = parent_node.ID)
        self.nodes_dict[self.current_ID] = child_node_L
```

```
            self.nodes_dict[self.current_ID + 1] = child_node_R
            self.current_ID += 2


    #############################
    ####### 2. PREDICTING #######
    #############################

    ###### LEAF MODES ######
    def _get_leaf_modes(self):
        self.leaf_modes = {}
        for node_ID, node in self.nodes_dict.items():
            if node.leaf:
                values, counts = np.unique(node.ysub, return_counts=True)
                self.leaf_modes[node_ID] = values[np.argmax(counts)]

    ####### PREDICT ########
    def predict(self, X_test):

        # Calculate leaf modes
        self._get_leaf_modes()

        yhat = []
        for x in X_test:
            node = self.nodes_dict[0]
            while not node.leaf:
                if node.dtype == 'quant':
                    if x[node.d] <= node.t:
                        node = self.nodes_dict[node.child_L]
                    else:
                        node = self.nodes_dict[node.child_R]
                else:
                    if x[node.d] in node.L_values:
                        node = self.nodes_dict[node.child_L]
                    else:
                        node = self.nodes_dict[node.child_R]
            yhat.append(self.leaf_modes[node.ID])
        return np.array(yhat)
```

With the weighted decision tree constructed, we are ready to build our `AdaBoost` class. The class closely follows the algorithm introduced in the content section, which is copied below for convenience.

Discrete AdaBoost Algorithm

Define the target variable to be $y_n \in \{-1, +1\}$.

1. Initialize the weights with $w_n^1 = \frac{1}{N}$ for $n = 1, 2, \dots, N$.
2. For $t = 1, \dots, T$,
    - Build weak learner t using weights $\mathbf{w}^t$.
    - Calculate fitted values $f^t(\mathbf{x}_n) \in \{-1, +1\}$ for $n = 1, 2, \dots, N$. Let I_n^t equal 1 If $f^t(\mathbf{x}_n) \neq y_n$ and 0 otherwise. That is, I_n^t indicates whether learner t misclassifies observation n.
    - Calculate the weighted error for learner t:

$$\epsilon^t = \frac{\sum_{n=1}^N w_n^t I_n^t}{\sum_{n=1}^N w_n^t}.$$

    - Calculate the accuracy measure for learner t:

$$\alpha^t = \log\left(\frac{1-\epsilon^t}{\epsilon^t}\right).$$

    - Update the weighting with

$$w_n^{t+1} = w_n^t \exp(\alpha^t I_n^t),$$

    for $n = 1, 2, \dots, N$.
3. Calculate the overall fitted values with $\hat{y}_n = \text{sign}\left(\sum_{t=1}^T \alpha^t f^t(\mathbf{x}_n)\right)$.

```
class AdaBoost:

    def fit(self, X_train, y_train, T, stub_depth = 1):
        self.y_train = y_train
        self.X_train = X_train
        self.N, self.D = X_train.shape
        self.T = T
        self.stub_depth = stub_depth

        ## Instantiate stuff
        self.weights = np.repeat(1/self.N, self.N)
        self.trees = []
        self.alphas = []
        self.yhats = np.empty((self.N, self.T))

        for t in range(self.T):

            ## Calculate stuff
            self.T_t = DecisionTreeClassifier()
            self.T_t.fit(self.X_train, self.y_train, self.weights, max_depth =
self.stub_depth)
            self.yhat_t = self.T_t.predict(self.X_train)
            self.epsilon_t = sum(self.weights*(self.yhat_t !=
self.y_train))/sum(self.weights)
            self.alpha_t = np.log( (1-self.epsilon_t)/self.epsilon_t )
            self.weights = np.array([w*(1-self.epsilon_t)/self.epsilon_t if
self.yhat_t[i] != self.y_train[i]
                                    else w for i, w in enumerate(self.weights)])
            ## Append stuff
            self.trees.append(self.T_t)
            self.alphas.append(self.alpha_t)
            self.yhats[:,t] = self.yhat_t

        self.yhat = np.sign(np.dot(self.yhats, self.alphas))

    def predict(self, X_test):
        yhats = np.zeros(len(X_test))
        for t, tree in enumerate(self.trees):
            yhats_tree = tree.predict(X_test)
            yhats += yhats_tree*self.alphas[t]
        return np.sign(yhats)
```

The `AdaBoost` model is finally fit below. To train the model, we supply the training data as well as `T`—the number of weak learners—and `stub_depth`—the depth for each tree (our weak learner).

```
booster = AdaBoost()
booster.fit(X_train, y_train, T = 30, stub_depth = 3)
yhat = booster.predict(X_test)
np.mean(yhat == y_test)
```

```
0.9759036144578314
```

2. Regression with AdaBoost.R2

Next, let's implement *AdaBoost.R2*, a common boosting algorithm for regression tasks. We'll again use the tips dataset from `seaborn`, loaded in the hidden code cell below.

```
## Import packages
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets

## Load data
tips = sns.load_dataset('tips')
X = np.array(tips.drop(columns = 'tip'))
y = np.array(tips['tip'])

## Train-test split
np.random.seed(1)
test_frac = 0.25
test_size = int(len(y)*test_frac)
test_idxs = np.random.choice(np.arange(len(y)), test_size, replace = False)
X_train = np.delete(X, test_idxs, 0)
y_train = np.delete(y, test_idxs, 0)
X_test = X[test_idxs]
y_test = y[test_idxs]
```

Since our boosting class will use regression trees for its weak learners, let's also import the regression tree we constructed in Chapter 5.

```
## Import decision trees
import import_ipynb
import regression_tree as rt;
```

Recall that the final fitted values in *AdaBoost.R2* are based on a weighted median. Let's first make a helper function to return the weighted median.

```
def weighted_median(values, weights):

    sorted_indices = values.argsort()
    values = values[sorted_indices]
    weights = weights[sorted_indices]
    weights_cumulative_sum = weights.cumsum()
    median_weight = np.argmax(weights_cumulative_sum >= sum(weights)/2)
    return values[median_weight]
```

We can then fit the `AdaBoostR2` class. This again follows the algorithm closely, which is again copied below for convenience.

AdaBoost.R2 Algorithm

1. Initialize the weights with $w_n^1 = \frac{1}{N}$ for $n = 1, 2, \dots, N$.
2. For $t = 1, 2, \dots, T$ or while $\bar{L}^t$, as defined below, is less than or equal to 0.5,
   - Draw a sample of size N from the training data with replacement and with probability w_n^t for $n = 1, 2, \dots, N$.
   - Fit weak learner t to the resampled data and calculate the fitted values on the original dataset. Denote these fitted values with $f^t(\mathbf{x}_n)$ for $n = 1, 2, \dots, N$.
   - Calculate the observation error L_n^t for $n = 1, 2, \dots, N$:

$$\begin{aligned} D^t &= \max_n \{ |y_n - f^t(\mathbf{x}_n)| \} \\ L_n^t &= \frac{|y_n - f^t(\mathbf{x}_n)|}{D^t} \end{aligned}$$

   - Calculate the model error $\bar{L}^t$:

$$\bar{L}^t = \sum_{n=1}^{N} L_n^t w_n^t$$

   If $\bar{L}^t \geq 0.5$, end iteration and set T equal to $t - 1$.
   - Let $\beta^t = \frac{\bar{L}^t}{1-\bar{L}^t}$. The lower β^t, the greater our confidence in the model.
   - Let $Z^t = \sum_{n=1}^{N} w_n^t (\beta^t)^{1-L_n}$ and update the model weights with

$$w_n^{t+1} = \frac{w_n^t (\beta^t)^{1-L_n}}{Z^t},$$

   which increases the weight for observations with a greater error L_n^t.

3. Set the overall fitted value for observation n equal to the weighted median of $f^t(\mathbf{x}_n)$ for $t = 1, 2, \dots, T$ using weights $\log(1/\beta^t)$ for model t.

```
class AdaBoostR2:

    def fit(self, X_train, y_train, T = 100, stub_depth = 1, random_state = None):

        self.y_train = y_train
        self.X_train = X_train
        self.T = T
        self.stub_depth = stub_depth
        self.N, self.D = X_train.shape
        self.weights = np.repeat(1/self.N, self.N)
        np.random.seed(random_state)

        self.trees = []
        self.fitted_values = np.empty((self.N, self.T))
        self.betas = []
        for t in range(self.T):

            ## Draw sample, fit tree, get predictions
            bootstrap_indices = np.random.choice(np.arange(self.N), size = self.N,
replace = True, p = self.weights)
            bootstrap_X = self.X_train[bootstrap_indices]
            bootstrap_y = self.y_train[bootstrap_indices]
            tree = rt.DecisionTreeRegressor()
            tree.fit(bootstrap_X, bootstrap_y, max_depth = stub_depth)
            self.trees.append(tree)
            yhat = tree.predict(X_train)
            self.fitted_values[:,t] = yhat

            ## Calculate observation errors
            abs_errors_t = np.abs(self.y_train - yhat)
            D_t = np.max(abs_errors_t)
            L_ts = abs_errors_t/D_t

            ## Calculate model error (and possibly break)
            Lbar_t = np.sum(self.weights*L_ts)
            if Lbar_t >= 0.5:
                self.T = t - 1
                self.fitted_values = self.fitted_values[:,:t-1]
                self.trees = self.trees[:t-1]
                break

            ## Calculate and record beta
            beta_t = Lbar_t/(1 - Lbar_t)
            self.betas.append(beta_t)

            ## Reweight
            Z_t = np.sum(self.weights*beta_t**(1-L_ts))
            self.weights *= beta_t**(1-L_ts)/Z_t

        ## Get median
        self.model_weights = np.log(1/np.array(self.betas))
        self.y_train_hat = np.array([weighted_median(self.fitted_values[n],
self.model_weights) for n in range(self.N)])

    def predict(self, X_test):
        N_test = len(X_test)
        fitted_values = np.empty((N_test, self.T))
        for t, tree in enumerate(self.trees):
            fitted_values[:,t] = tree.predict(X_test)
        return np.array([weighted_median(fitted_values[n], self.model_weights) for n in
range(N_test)])
```

Again, we fit our booster by providing training data in addition to `T`—the number of weak learners—and `stub_depth`—the depth for our regression tree weak learners.

```
booster = AdaBoostR2()
booster.fit(X_train, y_train, T = 50, stub_depth = 4, random_state = 123)

fig, ax = plt.subplots(figsize = (7,5))
sns.scatterplot(y_test, booster.predict(X_test));
ax.set(xlabel = r'$y$', ylabel = r'$\hat{y}$', title = 'Fitted vs. Observed Values for
AdaBoostR2')
sns.despine()
```

../../_images/boosting_24_0.png

Implementation

This section demonstrates how to fit bagging, random forest, and boosting models using `scikit-learn`. We will again use the penguins dataset for classification and the tips dataset for regression.

```
## Import packages
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

1. Bagging and Random Forests

Recall that bagging and random forests can handle both classification and regression tasks. For this example we will do classification on the `penguins` dataset. Recall that `scikit-learn` trees do not currently support categorical predictors, so we must first convert those to dummy variables

```
## Load penguins data
penguins = sns.load_dataset('penguins')
penguins = penguins.dropna().reset_index(drop = True)
X = penguins.drop(columns = 'species')
y = penguins['species']

## Train-test split
np.random.seed(1)
test_frac = 0.25
test_size = int(len(y)*test_frac)
test_idxs = np.random.choice(np.arange(len(y)), test_size, replace = False)
X_train = X.drop(test_idxs)
y_train = y.drop(test_idxs)
X_test = X.loc[test_idxs]
y_test = y.loc[test_idxs]

## Get dummies
X_train = pd.get_dummies(X_train, drop_first = True)
X_test = pd.get_dummies(X_test, drop_first = True)
```

Bagging

A simple bagging classifier is fit below. The most important arguments are `n_estimators` and `base_estimator`, which determine the number and type of weak learners the bagging model should use. The default `base_estimator` is a decision tree, though this can be changed as in the second example below, which uses Naive Bayes estimators.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.naive_bayes import GaussianNB

## Decision Tree bagger
bagger1 = BaggingClassifier(n_estimators = 50, random_state = 123)
bagger1.fit(X_train, y_train)

## Naive Bayes bagger
bagger2 = BaggingClassifier(base_estimator = GaussianNB(), random_state = 123)
bagger2.fit(X_train, y_train)

## Evaluate
print(np.mean(bagger1.predict(X_test) == y_test))
print(np.mean(bagger2.predict(X_test) == y_test))
```

```
0.963855421686747
0.9156626506024096
```

Random Forests

An example of a random forest in `scikit-learn` is given below. The most important arguments to the random forest are the number of estimators (decision trees), `max_features` (the number of predictors to consider at each split), and any chosen parameters for the decision trees (such as the maximum depth). Guidelines for setting each of these parameters are given below.

- n_estimators: In general, the more base estimators the better, though there are diminishing marginal returns. While increasing the number of base estimators does not risk overfitting, it eventually provides no benefit.
- max_features: This argument is set by default to the square root of the number of total features (which is made explicit in the example below). If this value equals the number of total features, we are left with a bagging model. Lowering this value lowers the amount of correlation between trees but also prevents the base estimators from learning potentially valuable information.
- Decision tree parameters: These parameters are generally left untouched. This allows the individual decision trees to grow deep, increasing variance but decreasing bias. The variance is then decreased by the ensemble of individual trees.

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators = 100, max_features =
int(np.sqrt(X_test.shape[1])), random_state = 123)
rf.fit(X_train, y_train)
print(np.mean(rf.predict(X_test) == y_test))
```

```
0.9879518072289156
```

2. Boosting

Note

Note that the AdaBoostClassifier from scikit-learn uses a slightly different algorithm than the one introduced in the concept section though results should be similar. The AdaBoostRegressor class in scikit-learn uses the same algorithm we introduced: *AdaBoost.R2*

AdaBoost Classification

The AdaBoostClassifier in scikit-learn is actually able to handle multiclass target variables, but for consistency, let's use the same binary target we did in our AdaBoost construction: whether the penguin's species is *Adelie*.

```
## Make binary
y_train = (y_train == 'Adelie')
y_test = (y_test == 'Adelie')
```

We can then fit the classifier with the AdaBoostClassifier class as below. Again, we first convert categorical predictors to dummy variables. The classifier will by default use 50 decision trees, each with a max depth of 1, for the weak learners.

```
from sklearn.ensemble import AdaBoostClassifier

## Get dummies
X_train = pd.get_dummies(X_train, drop_first = True)
X_test = pd.get_dummies(X_test, drop_first = True)

## Build model
abc = AdaBoostClassifier(n_estimators = 50)
abc.fit(X_train, y_train)
y_test_hat = abc.predict(X_test)

## Evaluate
np.mean(y_test_hat == y_test)
```

```
0.9759036144578314
```

A different weak learner can easily be used in place of a decision tree. The below shows an example using logistic regression.

```
from sklearn.linear_model import LogisticRegression
abc = AdaBoostClassifier(base_estimator = LogisticRegression(max_iter = 1000))
abc.fit(X_train, y_train);
```

AdaBoost Regression

AdaBoost regression is implemented almost identically in `scikit-learn`. An example with the `tips` dataset is shown below.

```python
## Load penguins data
tips = sns.load_dataset('tips')
tips = tips.dropna().reset_index(drop = True)
X = tips.drop(columns = 'tip')
y = tips['tip']

## Train-test split
np.random.seed(1)
test_frac = 0.25
test_size = int(len(y)*test_frac)
test_idxs = np.random.choice(np.arange(len(y)), test_size, replace = False)
X_train = X.drop(test_idxs)
y_train = y.drop(test_idxs)
X_test = X.loc[test_idxs]
y_test = y.loc[test_idxs]
```

```python
from sklearn.ensemble import AdaBoostRegressor

## Get dummies
X_train = pd.get_dummies(X_train, drop_first = True)
X_test = pd.get_dummies(X_test, drop_first = True)

## Build model
abr = AdaBoostRegressor(n_estimators = 50)
abr.fit(X_train, y_train)
y_test_hat = abr.predict(X_test)

## Visualize predictions
fig, ax = plt.subplots(figsize = (7, 5))
sns.scatterplot(y_test, y_test_hat)
ax.set(xlabel = r'$y$', ylabel = r'$\hat{y}$', title = r'Test Sample $y$ vs.
$\hat{y}$')
sns.despine()
```

../../_images/code_24_0.png

Concept

The neural network is a highly powerful and versatile class of models that has become quite a hot topic in machine learning. While the neural network's ability to often outperform other popular model classes has earned it a reputation for being a near-magical black box algorithm, networks are not terribly complex or mysterious. Rather, by optimizing a highly-parametric and nonlinear structure, neural networks are flexible enough to model subtle relationships that other models may struggle to detect.

Note

Neural networks come in a variety of forms intended to accomplish a variety of tasks. Recurrent neural networks, for instance, are designed to model time series data, and convolutional neural networks are designed to model image data. In this chapter, we only cover feed-forward neural networks (FFNNs). FFNNs can be used for regression or classification tasks and serve as a natural introduction to other forms of neural networks.

This section is organized as follows.

1. Model Structure
    1. An Overview
    2. Communication between Layers
    3. Activation Functions
2. Optimization
    1. Back Propagation
    2. Calculating Gradients
    3. Combining Results with the Chain Rule
3. Combining Observations
    1. A New Representation
    2. Gradients

1. Model Structure

Throughout this chapter, suppose we have training data $\{\mathbf{x}_n, \mathbf{y}_n\}_{n=1}^N$ with $\mathbf{x}_n \in \mathbb{R}^{D_x}$—which does *not* include an intercept term—and $\mathbf{y}_n \in \mathbb{R}^{D_y}$ for $n = 1, 2, \dots, N$. In other words, for each observation we have D_x predictors and D_y target variables. In this chapter, these will primarily be referred to as the *input* and *output* variables, respectively. Note that unlike in previous chapters, we might now have a *vector* of target variables rather than a single value. If there is only one target variable per observation (i.e. $D_y = 1$), we will write it as y_n rather than $\mathbf{y}_n$.

1.1 An Overview

The diagram below is a helpful representation of a basic neural network. Neural networks operate in layers. The network starts of with an *input layer*, consisting of the vector of predictors for a single observation. This is shown by $x_0, \dots, x_3$ in the diagram (indicating that $D_x = 4$ here). The network then passes through one or more *hidden layers*. The first hidden layer is a function of the input layer and each following hidden layer is a function of the last. (We will discuss these functions in more detail later). The network below has two hidden layers. Finally, the network passes from the last hidden layer into an *output layer*, representing the target variable or variables. In the network below, the target variable is two-dimensional (i.e. $D_y = 2$), so the layer is represented by the values y_0 and y_1.

Note

Diagrams like the one below are commonly used to represent neural networks. Note that these diagrams show only a single observation at a time. For instance, $x_0, \dots x_3$ represent four predictors within one observation, rather than four different observations.

Each layer in a neural network consists of *neurons*, represented by the circles in the diagram above. Neurons are simply scalar values. In the *input layer*, each neuron represents a single predictor. In the above diagram, the input layer has four neurons, labeled x_0 through x_3, each representing a single predictor. The neurons in the input layer then determine the neurons in the first hidden layer, labeled $z_0^{(1)}$ through $z_2^{(1)}$. We will discuss *how* shortly, but for now simply note the lines running from the input layer's neurons to the first hidden layer's neurons in the diagram above. Once the neurons in the first hidden layer are set, they become predictors for the next layer, acting just as the input layer did. When the neurons in the final hidden layer are fixed, they act as predictors for the output layer.

One natural question is how many layers our neural network should contain. There is no single answer to this question, as the number of layers is chosen by the modeler. Any true neural network will have an input layer, an output layer, and at least one hidden layer. The network above has two hidden layers. Note that the superscript indicates the hidden layer number, e.g. $z_0^{(1)}$ through $z_2^{(1)}$ are in the first hidden layer and $z_0^{(2)}$ through $z_2^{(2)}$ are in the second hidden layer. We could also consider the input layer as an exogenous "hidden layer" and represent it with $z_0^{(0)}$ through $z_3^{(0)}$.

Another natural question is how many neurons each layer should contain. This is in part chosen by the modeler and in part predetermined. If our predictor vectors are of length D, the input layer must have D neurons. Similarly, the output layer must have as many neurons as there are target variables. If, for instance, our model attempts to predict a store's revenue and its costs (two targets) in a given month, our output layer must have two neurons. The sizes of the hidden layers, however, are chosen by the modeler. Too few neurons may cause underfitting by preventing the network from picking up on important patterns in the data while too many neurons may cause overfitting, allowing the network to fit parameters that match the training data exactly.

1.2 Communication between Layers

Let's now turn to the process through which one layer communicates with the next. In this section, let $\mathbf{z}^{(a)}$ and $\mathbf{z}^{(b)}$ represent the vector of neurons in any two consecutive layers. For instance, $\mathbf{z}^{(a)}$ might be an input layer and $\mathbf{z}^{(b)}$ the first hidden layer or $\mathbf{z}^{(a)}$ might be a hidden layer and $\mathbf{z}^{(b)}$ the following hidden layer. Suppose $\mathbf{z}^{(a)} \in \mathbb{R}^{D_a}$ and $\mathbf{z}^{(b)} \in \mathbb{R}^{D_b}$.

In a feed-forward neural network, each neuron in $\mathbf{z}^{(b)}$ is a function of every neuron in $\mathbf{z}^{(a)}$. This function occurs in two stages: first a linear mapping of $\mathbf{z}^{(a)}$ onto one dimension, then a nonlinear function called an *activation function*. Let's look at a single neuron within $\mathbf{z}^{(b)}$, $z_i^{(b)}$. The transformation from $\mathbf{z}^{(a)}$ to $z_i^{(b)}$ takes the form

$$
\begin{aligned}
h_i^{(b)} &= \mathbf{w}_i^\top \mathbf{z}^{(a)} + c_i \\
z_i^{(b)} &= f(h_i),
\end{aligned}
$$

where $\mathbf{w}_i \in \mathbb{R}^{D_a}$ is a vector of weights, c_i is a constant intercept term, and $f()$ is an activation function. Note that $\mathbf{w}_i$ and c_i are specific to the i^{th} neuron in $\mathbf{z}^{(b)}$ while $f()$ is typically common among all neurons in $\mathbf{z}^{(b)}$. We can also write the function relating the two layers in matrix form, as below.

$$
\begin{aligned}
\mathbf{h}^{(b)} &= \mathbf{W}\mathbf{z}^{(a)} + \mathbf{c} \\
\mathbf{z}^{(b)} &= f(\mathbf{h}^{(b)}),
\end{aligned}
$$

where $\mathbf{W} \in \mathbb{R}^{D_b \times D_a}$, $\mathbf{c} \in \mathbb{R}^{D_b}$ and $f()$ is applied element-wise.

> **Note**
>
> Note that we haven't yet discussed *how* $\mathbf{W}$, $\mathbf{c}$ or $f()$ are determined. For now, consider these all to be fixed and focus on the structure of a network. *How* we determine these values is discussed in the optimization section below.

Once $\mathbf{z}^{(b)}$ is fixed, we use the same process to create the next layer, $\mathbf{z}^{(c)}$. When discussing many layers at a time, it is helpful to add superscripts to $\mathbf{W}$, $\mathbf{c}$, and $f()$ to indicate the layer. We can write the transmission of $\mathbf{z}^{(a)}$ to $\mathbf{z}^{(b)}$ followed by $\mathbf{z}^{(b)}$ to $\mathbf{z}^{(c)}$ as

$$
\begin{aligned}
\mathbf{z}^{(b)} &= f^{(b)}\left(\mathbf{W}^{(b)}\mathbf{z}^{(a)} + \mathbf{c}^{(b)}\right) \\
\mathbf{z}^{(c)} &= f^{(c)}\left(\mathbf{W}^{(c)}\mathbf{z}^{(b)} + \mathbf{c}^{(c)}\right).
\end{aligned}
$$

A more mathematical representation of a neural network is given below. The network starts with a vector of predictors $\mathbf{x}$. This vector is then multiplied by $\mathbf{W}^{(1)}$ and added to $\mathbf{c}^{(1)}$, which sums to $\mathbf{h}^{(1)}$. We then apply an activation $f^{(1)}$ to $\mathbf{h}^{(1)}$, which results in our single hidden layer, $\mathbf{z}^{(1)}$. The same process is then applied to $\mathbf{z}^{(1)}$, which results in our output vector, $\mathbf{y}$.

1.3 Activation Functions

As we have seen, we create a neuron in one layer by taking a linear mapping of the neurons in the previous layer and then applying some *activation function*. What exactly is this activation function? An activation function is a (typically) nonlinear function that allows the network to learn complex relationships between the predictor(s) and the target variable(s).

Suppose, for instance, the relationship between a target variable y_n and a predictor x_n is given by

$$
y_n = |x_n| + \epsilon_n,
$$

where ϵ_n is a noise term. Despite its simplicity, this relationship cannot be accurately fit by a linear model.

Ideally, we would apply some function to the predictor and use a different model depending on the result of this function. In the case above, $x_n > 0$ would "activate" the model $y_n \approx x_n$, and $x_n \leq 0$ would "activate" the model $y_n \approx -x_n$. Hence the name "activation function".

There are many commonly used activation functions, and deciding which function to use is a major consideration in modeling a neural network. Here we will limit our discussion to two of the most common functions: the ReLU (Rectified Linear Unit) and sigmoid functions. The linear activation function (which is really the absence of an activation function) is also discussed.

ReLU

ReLU is a simple yet extremely common activation function. It is defined as

$$
f(x) = \max(x, 0).
$$

How can such a simple function benefit a neural network? ReLU acts like a switch, selectively turning channels on and off. Consider fitting a neural network to the dataset above generated with $y_n = |x_n| + \epsilon_n$. Let's use a very simple network represented by the diagram below. This network has one predictor, a single hidden layer with two neurons, and one output variable.

Now let's say we decide to use $f(\mathbf{x}) = \text{ReLU}(\mathbf{x})$ and we land on the following parameters:

$$\mathbf{W}^{(1)} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \; \mathbf{c}^{(1)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \; \mathbf{W}^{(2)} = \begin{pmatrix} 1 & 1 \end{pmatrix}, \; \mathbf{c}^{(2)} = 0.$$

This is equivalent to the following complete model

$$\begin{aligned} \mathbf{z}^{(1)} &= \text{ReLU}\left(\begin{pmatrix} 1 \\ -1 \end{pmatrix} x\right) \\ y &= \begin{pmatrix} 1 & 1 \end{pmatrix} \mathbf{z}^{(1)}. \end{aligned}$$

Will this model be able to fit our dataset? Suppose $x_n = c$ for some *positive* constant c. We will then get

$$\begin{aligned} \mathbf{z}^{(1)} &= \text{ReLU}\left(\begin{pmatrix} c \\ -c \end{pmatrix}\right) = \begin{pmatrix} c \\ 0 \end{pmatrix} \\ y &= \begin{pmatrix} 1 & 1 \end{pmatrix} \begin{pmatrix} c \\ 0 \end{pmatrix} = c. \end{aligned}$$

So we will predict $y_n = |x_n| = c$, a sensible result! Similarly, if $x_n = -c$, we would again obtain the valid prediction $y_n = |x_n| = c$. ReLU is able to achieve this result by activating a different channel depending on the value of x_n: if x_n is greater than 0, it activates $y_n = x_n$, and if x_n is less than 0, it activates $y_n = -x_n$.

As we will see in the next section, fitting a neural network consists of taking gradients of our activation functions. Fortunately ReLU has a straightforward derivative:

$$\frac{\partial}{\partial x}\text{ReLU}(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0. \end{cases}$$

Note that this derivative is not technically defined at 0. In practice, it is very unlikely that we will be applying an activation function to 0 *exactly*, though in that case the convention is to set its derivative equal to 0.

Sigmoid

A second common activation function is the *logistic sigmoid function*, often referred to as just *the sigmoid function*. This function was introduced in chapter 3 in the context of the logistic regression. The sigmoid function is defined as

$$\sigma(x) = \frac{1}{1 + \exp(-x)}.$$

Note that the sigmoid function takes any real value and returns a value between 0 and 1. As a result, the sigmoid function is commonly applied to the last hidden layer in a network in order to return a probability estimate in the output layer. This makes it common in classification problems.

As we saw in chapter 3, a convenient fact about the sigmoid function is that we can express its derivative in terms of itself.

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \frac{1}{1 + \exp(-x)} \cdot \frac{\exp(-x)}{1 + \exp(-x)} = \sigma(x)\left(1 - \sigma(x)\right).$$

The Linear Activation Function

Another possible activation function is the "linear" activation function, which is the same as skipping the activation function altogether. The linear activation function simply returns its input. It is defined with

$$f(x) = x,$$

and has derivative

$$\frac{\partial f(x)}{\partial x} = 1.$$

The linear activation function is often used before the last layer in a neural network for regression. Rather than constraining the fitted values to be in some range or setting half of them equal to 0, we want to leave them as they are.

2. Optimization

We have now seen that a neural network operates through a series of linear mappings and activation functions. The linear mapping for layer ℓ is determined by the parameters in $\mathbf{W}^{(\ell)}$ and $\mathbf{c}^{(\ell)}$, also called the *weights*. This section discusses the process through which the weights in a neural network are fit, called *back propagation*.

The rest of this page requires a good amount of matrix differentiation, which is introduced in the math appendix. Note that we use the "numerator layout," meaning for $\mathbf{y} \in \mathbb{R}^m$ and $\mathbf{x} \in \mathbb{R}^n$, we write $\partial \mathbf{y}/\partial \mathbf{x}$ as

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ & \dots & \\ \frac{\partial y_m}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}.$$

2.1 Back Propagation

Suppose we choose some loss function $\mathcal{L}$ for our network to minimize. Note that because our target variable is multi-dimensional, $\boldsymbol{\mathcal{L}}$ function will be a vector of losses (e.g. the loss for the first target, the loss for the second target, etc.). To find the network's optimal weights, we can conduct gradient descent, repeatedly taking the derivative of our loss function with respect to each weight and adjusting accordingly. As we will see, this involves finding the gradient of the network's final weights, then using the chain rule to find the gradient of the weights that came earlier. In this process, we move backward through the network, and hence the name "back propagation."

Consider conducting gradient descent for the network above. Write the loss function as $\mathcal{L}(\hat{\mathbf{y}})$, where $\hat{\mathbf{y}}$ is the network's output. Let's start by writing out the derivative of $\mathcal{L}$ with respect to $\mathbf{W}^{(L)}$, the final matrix of weights in our network. We can do this with the chain rule, as below.

$$\frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}^{(L)}} \cdot \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{W}^{(L)}}$$

The gradient of $\mathbf{c}^{(L)}$ is equivalent. The math behind these calculations is covered in the following section. Next, we want to find the gradient of $\mathbf{W}^{(L-1)}$, shown below.

$$\frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \mathbf{W}^{(L-1)}} = \frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}^{(L)}} \cdot \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{z}^{(L-1)}} \cdot \frac{\partial \mathbf{z}^{(L-1)}}{\partial \mathbf{h}^{(L-1)}} \cdot \frac{\partial \mathbf{h}^{(L-1)}}{\partial \mathbf{W}^{(L-1)}}$$

This expression is pretty ugly, but there is a shortcut. This gradient and the gradient of $\mathbf{W}^{(L)}$ share the first two terms, which represent the gradient of $\mathbf{h}^{(L)}$. To save time (both in writing out the gradients and in calculating them in practice), we can record this gradient, $\nabla \mathbf{h}^{(L)}$, and apply it where necessary. We can do the same with $\nabla \mathbf{h}^{(L-1)}$, which simplifies the gradient of $\mathbf{W}^{(L-2)}$:

$$\frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \mathbf{W}^{(L-2)}} = \nabla \mathbf{h}^{(L-1)} \cdot \frac{\partial \mathbf{h}^{(L-1)}}{\partial \mathbf{z}^{(L-2)}} \cdot \frac{\partial \mathbf{z}^{(L-2)}}{\partial \mathbf{h}^{(L-2)}} \cdot \frac{\partial \mathbf{h}^{(L-2)}}{\partial \mathbf{W}^{(L-2)}}.$$

We continue this same process until we reach the first set of weights.

We've now seen intuitively how to find the gradients for our network's many weights. To conduct back propagation, we simply use these gradients to run gradient descent. Next, let's see how to actually calculate these gradients.

2.2 Calculating Gradients

In this section we will derive the gradients used in back propagation. For each iteration in this process we need to know the derivative of our loss function with respect to each weight in the entire network. For the network shown above, this requires calculating the following gradients:

$$\frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \mathbf{W}^{(1)}}, \frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \mathbf{c}^{(1)}}, \dots, \frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \mathbf{W}^{(L)}}, \text{ and } \frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \mathbf{c}^{(L)}}.$$

Since we will find these with the chain rule, we will need to calculate other gradients along the way. All the necessary gradients are derived below. Note that the following sub-sections cover the stages within a single layer of a network in reverse order (as back propagation does).

Note

Note that the rest of this section considers only one observation at a time. The vector $\mathbf{y}$, for instance, refers to the output variables for a single observation, rather than a vector of 1-dimensional output variables for several observations. Similarly, $\partial \mathcal{L}(\hat{\mathbf{y}})/\partial \hat{\mathbf{y}}$ refers to the derivative of the loss with respect to a single observation's output. The final section discusses how to combine the derivatives from multiple observations.

For the following, let there be L layers in total. Also let layer ℓ have size D_ℓ, except the input and output layers which have sizes D_x and D_y, respectively.

2.2.1 Loss Functions and their Gradients

Back propagation begins where the network ends: the loss function $\mathcal{L}(\hat{\mathbf{y}})$. Let's start by introducing some common loss functions and their derivatives with respect to our predictions, $\hat{\mathbf{y}}$. Later, using the chain rule, we will use these derivatives to calculate the derivatives with respect to our network's weights.

A common loss function for quantitative output variables is the residual sum of squares. For a single observation, the loss is

$$\mathcal{L}_{RSS}(\hat{\mathbf{y}}) = (\mathbf{y} - \hat{\mathbf{y}})^2.$$

Note that we have a vector of losses because there are multiple output variables and we consider the loss for each variable independently. Now for the first step in back propagation, we calculate the derivative of this loss with respect to $\hat{\mathbf{y}}$, which is simply given by

$$\frac{\partial \mathcal{L}_{RSS}(\hat{\mathbf{y}})}{\partial \hat{\mathbf{y}}} = -2(\mathbf{y} - \hat{\mathbf{y}})^\top \in \mathbb{R}^{1 \times D_y}.$$

Since we are using the numerator layout convention, this derivative is a length-D_y row vector, or equivalently a 1 by D_y matrix.

For binary classification problems, a common loss function is the log loss or cross entropy, given by

$$\mathcal{L}_{Log}(\hat{\mathbf{y}}) = -\Big(\mathbf{y} \log \hat{\mathbf{y}} + (1 - \mathbf{y}) \log(1 - \hat{\mathbf{y}})\Big),$$

where the i^{th} entry in $\hat{\mathbf{y}}$ gives the estimated probability that the i^{th} output variable equals 1. The derivative of this loss function with respect to $\hat{\mathbf{y}}$ is given by

$$\frac{\partial \mathcal{L}_{Log}(\hat{\mathbf{y}})}{\partial \hat{\mathbf{y}}} = \left(-\frac{\mathbf{y}}{\hat{\mathbf{y}}} + \frac{1 - \mathbf{y}}{1 - \hat{\mathbf{y}}}\right)^\top \in \mathbb{R}^{1 \times D_y}.$$

Once we calculate $\partial \mathcal{L}(\hat{\mathbf{y}})/\partial \hat{\mathbf{y}}$, we can move further back into the network. Since $\hat{\mathbf{y}}$ is the result of an activation function, the next step in back propagation is to calculate the derivative of our activation functions.

2.2.2 Gradients of the Activation Functions

Recall that $\mathbf{z}^{(\ell)}$, the output layer of ℓ, is the result of an activation function applied to a linear mapping $\mathbf{h}^{(\ell)}$. This includes the output of the final layer, $\hat{\mathbf{y}}$, which we can also write as $\mathbf{z}^{(L)}$.

ReLU

Suppose we have $\mathbf{z}^{(\ell)} = f^{(\ell)}(\mathbf{h}^{(\ell)})$ where $f^{(\ell)}$ is the ReLU function. We are interested in $\partial \mathbf{z}^{(\ell)}/\partial \mathbf{h}^{(\ell)}$. For $i \neq j$, we have

$$\frac{\partial z_i^{(\ell)}}{\partial h_j^{(\ell)}} = 0,$$

since $z_i^{(\ell)}$ is not a function of $h_j^{(\ell)}$. Then using the ReLU derivative, we have

$$\frac{\partial z_i^{(\ell)}}{\partial h_i^{(\ell)}} = \begin{cases} 1, & h_i^{(\ell)} > 0 \\ 0, & h_i^{(\ell)} \leq 0. \end{cases}$$

We can then compactly write the entire derivative as

$$\frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{h}^{(\ell)}} = \text{diag}(\mathbf{h}^{(\ell)} > 0) \in \mathbb{R}^{D_\ell \times D_\ell}.$$

Sigmoid
Now suppose we have $\mathbf{z}^{(\ell)} = f^{(\ell)}(\mathbf{h}^{(\ell)})$ where $f^{(\ell)}$ is the sigmoid function. Again, the partial derivative $\partial z_i^{(\ell)} / \partial h_j^{(\ell)}$ is 0 for $i \neq j$. By the sigmoid derivative, we have

$$\frac{\partial z_i^{(\ell)}}{\partial h_i^{(\ell)}} = \sigma(h_i^{(\ell)})(1 - \sigma(h_i^{(\ell)})).$$

We can again write the entire result compactly as

$$\frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{h}^{(\ell)}} = \text{diag}\left(\sigma(\mathbf{h}^{(\ell)})(1 - \sigma(\mathbf{h}^{(\ell)})\right) \in \mathbb{R}^{D_\ell \times D_\ell}.$$

Linear
Finally, suppose we have $\mathbf{z}^{(\ell)} = f^{(\ell)}(\mathbf{h}^{(\ell)})$ where $f^{(\ell)}$ is the linear function. We then have

$$\frac{\partial z_i^{(\ell)}}{\partial h_j^{(\ell)}} = \begin{cases} 1, & i = j \\ 0, & i \neq j. \end{cases}$$

The entire gradient is then simply

$$\frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{h}^{(\ell)}} = I_{D_\ell} \in \mathbb{R}^{D_\ell \times D_\ell}.$$

2.2.3 Gradients of the Weights

We are now finally able to calculate the gradients of our weights. Specifically, we will calculate $\partial \mathbf{h}^{(\ell)} / \partial \mathbf{c}^{(\ell)}$ and $\partial \mathbf{h}^{(\ell)} / \partial \mathbf{W}^{(\ell)}$ which, when combined with our previous results through the chain rule, will allow us to obtain the derivative of the loss function with respect the layer ℓ weights.

Recall that we obtain $\mathbf{h}^{(\ell)}$ through

$$\mathbf{h}^{(\ell)} = \mathbf{W}^{(\ell)} \mathbf{z}^{(\ell-1)} + \mathbf{c}^{(\ell)},$$

giving us the simple derivative

$$\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{c}^{(\ell)}} = I_{D_\ell} \in \mathbb{R}^{D_\ell \times D_\ell}.$$

The derivative $\partial \mathbf{h}^{(\ell)} / \partial \mathbf{W}^{(\ell)}$ is more complicated. Since we are taking the derivative of a vector with respect to a matrix, our result will be a tensor. The shape of this tensor will be $D_\ell \times (D_\ell \times D_{\ell-1})$ since $\mathbf{h}^{(\ell)} \in R^{D_\ell}$ and $\mathbf{W}^{(\ell)} \in \mathbb{R}^{D_\ell \times D_{\ell-1}}$. The first element of this tensor is given by $\partial h_1^{(\ell)} / \partial \mathbf{W}^{(\ell)}$. Using the expression for $\mathbf{h}^{(\ell)}$ above, we see that this is a matrix with $(\mathbf{z}^{(\ell-1)})^\top$ in the first row and 0s everywhere else. More generally, the i^{th} entry in this derivative will have all 0s except $(\mathbf{z}^{(\ell-1)})^\top$ in its i^{th} row. This is represented below.

$$\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{W}^{(\ell)}} = \begin{bmatrix} \frac{\partial h_1^{(\ell)}}{\partial \mathbf{W}^{(\ell)}} \\ \\ \dots \\ \\ \frac{\partial h_{n_\ell}^{(\ell)}}{\partial \mathbf{W}^{(\ell)}} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} \mathbf{z}^{(\ell-1)})^\top \\ \dots \\ \mathbf{0}^\top \end{bmatrix} \\ \dots \\ \begin{bmatrix} \mathbf{0}^\top \\ \dots \\ (\mathbf{z}^{(\ell-1)})^\top \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{D_\ell \times (D_\ell \times D_{\ell-1})}.$$

2.2.4 One Last Gradient

We now have all the results necessary to calculate the derivative of the loss function with respect to the weights in the *final* layer. For instance, we can evaluate

$$\frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}^{(L)}} \cdot \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{W}^{(L)}}$$

using the results from sections 2.1, 2.2, and 2.3. However, to obtain the derivative of the loss function with respect to weights in the *previous* layers, we need one more derivative: the derivative of $\mathbf{h}^{(\ell)}$, the linear mapping in layer ℓ, with respect to $\mathbf{z}^{(\ell-1)}$, the output of the previous layer. Fortunately, this derivative is simple:

$$\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{z}^{(\ell-1)}} = \mathbf{W}^{(\ell)}.$$

Now that we have $\partial \mathbf{h}^{(\ell)} / \partial \mathbf{z}^{(\ell-1)}$, we reuse the results from sections 2.2 and 2.3 to calculate $\partial \mathbf{z}^{(\ell-1)} / \partial \mathbf{h}^{(\ell-1)}$ and $\partial \mathbf{h}^{(\ell-1)} / \partial \mathbf{W}^{(\ell-1)}$ (respectively); this gives us all the necessary results to compute the gradient of the weights in the previous layer. We then rinse, lather, and repeat with layer $\ell - 2$ through the first layer.

2.3 Combining Results with the Chain Rule

We've seen lots of individual derivatives. Ultimately, we really care about the derivatives of the loss function with respect to the network's weights. Let's review by calculating the derivatives of the loss function with respect to the weights in the final layer for the familiar network above. Suppose $f^{(2)}$ is the Sigmoid function and we use the log loss. For $\mathbf{W}^{(2)}$ we get the following.

$$\begin{aligned}
\frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \mathbf{W}^{(2)}} &= \frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}^{(2)}} \cdot \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{W}^{(2)}} \\
&= -\left(\frac{\mathbf{y}}{\hat{\mathbf{y}}} + \frac{1-\mathbf{y}}{1-\hat{\mathbf{y}}}\right)^\top \cdot \operatorname{diag}\left(\sigma(\mathbf{h}^{(2)})(1-\sigma(\mathbf{h}^{(2)}))\right) \cdot \mathbf{T} \\
&= -\begin{bmatrix} (\frac{y_1}{\hat{y}_1} + \frac{1-y_1}{1-\hat{y}_1}) \cdot \sigma(h_1^{(2)})(1-\sigma(h_1^{(2)})) \cdot \mathbf{z}^{(1)} \\ \dots \\ (\frac{y_{n_2}}{\hat{y}_{n_2}} + \frac{1-y_{n_2}}{1-\hat{y}_{n_2}}) \cdot \sigma(h_{n_2}^{(2)})(1-\sigma(h_{n_2}^{(2)})) \cdot \mathbf{z}^{(1)} \end{bmatrix} \in \mathbb{R}^{n_2 \times n_1},
\end{aligned}$$

where $\mathbf{T}$ is the tensor derivative discussed in section 2.2.3.

For $\mathbf{c}^{(2)}$, we get

$$\begin{aligned}
\frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \mathbf{c}^{(2)}} &= \frac{\partial \mathcal{L}(\hat{\mathbf{y}})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}^{(2)}} \cdot \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{c}^{(2)}} \\
&= -\left(\frac{\mathbf{y}}{\hat{\mathbf{y}}} + \frac{1-\mathbf{y}}{1-\hat{\mathbf{y}}}\right)^\top \cdot \operatorname{diag}\left(\sigma(\mathbf{h}^{(2)})(1-\sigma(\mathbf{h}^{(2)}))\right) \cdot I_{n_2} \\
&= -\begin{bmatrix} (\frac{y_1}{\hat{y}_1} + \frac{1-y_1}{1-\hat{y}_1}) \cdot \sigma(h_1^{(2)})(1-\sigma(h_1^{(2)})) \\ \dots \\ (\frac{y_{n_2}}{\hat{y}_{n_2}} + \frac{1-y_{n_2}}{1-\hat{y}_{n_2}}) \cdot \sigma(h_{n_2}^{(2)})(1-\sigma(h_{n_2}^{(2)})) \end{bmatrix} \in \mathbb{R}^{n_2}.
\end{aligned}$$

3. Combining Observations

So far, we've only considered the derivative of the loss function for a *single* observation. When training a network, we will of course want to consider the entire dataset. One way to do so is to simply add the derivatives of the loss function with respect to the weights across observations. Since the loss over the dataset is the sum of the individual observation losses and the derivative of a sum is the sum of the derivatives, we can simply add the results above. For instance, to find the derivative of the loss with respect to the final matrix of weights $\mathbf{W}^{(L)}$, we could loop through observations and sum the individual derivatives:

$$\frac{\partial \mathcal{L}(\{\hat{\mathbf{y}}_n\}_{n=1}^N)}{\partial \mathbf{W}^{(L)}} = \frac{\partial \sum_{n=1}^N \mathcal{L}(\hat{\mathbf{y}}_n)}{\partial \mathbf{W}^{(L)}} = \sum_{n=1}^N \frac{\partial \mathcal{L}(\hat{\mathbf{y}}_n)}{\partial \mathbf{W}^{(L)}}.$$

While straightforward, this approach is computationally inefficient. The rest of this section outlines a more complicated but *much* faster method.

3.1 A New Representation

So far, we've treated our predictors and outputs as vectors. The network starts with $\mathbf{x}$ and outputs $\mathbf{z}^{(1)}$. Then it predicts with $\mathbf{z}^{(1)}$ and outputs $\mathbf{z}^{(2)}$. It repeats this process until $\mathbf{z}^{(L-1)}$ outputs $\hat{\mathbf{y}}$. To incorporate multiple observations, we can turn these vectors into matrices. Again suppose our dataset consists of N observations with $\mathbf{x}_n \in \mathbb{R}^{D_x}$ and $\mathbf{y}_n \in \mathbb{R}^{D_y}$. We start with $\mathbf{X} \in \mathbb{R}^{N \times D_x}$, whose n^{th} row is $\mathbf{x}_n$. Note that in $\mathbf{X}$, $\mathbf{x}_n$ is a row vector; to keep consistent with our previous sections, we want it to be a column vector. So, we'll work with $\mathbf{X}^\top$ rather than $\mathbf{X}$.

Rather than feeding each observation through the network at once, we will feed all observations together and give each observation its own column. Each column in $\mathbf{X}^\top$ represents an observation's predictors. We then multiply this matrix by $\mathbf{W}^{(1)}$ and add $\mathbf{c}^{(1)}$ *element-wise* to get $\mathbf{H}^{(1)}$. Each column in $\mathbf{H}^{(1)}$ represents a vector of linear combinations of the corresponding column in $\mathbf{X}^\top$. We then pass $\mathbf{H}^{(1)}$ through an activation function to obtain $\mathbf{Z}^{(1)}$. Similarly, each column in $\mathbf{Z}^{(1)}$ represents the output vector for the corresponding observation in $\mathbf{X}^\top$. We then repeat, with $\mathbf{Z}^{(1)}$ acting as the matrix of predictors for the next layer. Ultimately, we will obtain a matrix $\hat{\mathbf{Y}}^\top \in \mathbb{R}^{D_y \times N}$ whose n^{th} column represents the vector of fitted values for the n^{th} observation.

3.2 Gradients

While this new representation is more efficient, it also makes the gradients more complicated since we are taking derivatives with respect to matrices rather than vectors. Ordinarily, the derivative of one matrix with respect to another would be a four-dimensional tensor. Luckily, there's a shortcut.

For each parameter θ in our network, we will find its gradient by asking "which parameters does θ affect in the next layer". Supposing the answer is some set $\{\psi_1, \psi_2, \ldots, \psi_n\}$, we will calculate the derivative of the loss function with respect to θ as

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{i=1}^{n} \frac{\partial L}{\partial \psi_i} \cdot \frac{\partial \psi_i}{\partial \theta}.$$

Recall that our loss function is a vector $\mathcal{L}$ of size D_y since we have D_y output variables. This loss vector is a *row-wise* function of the prediction matrix, $\hat{\mathbf{Y}}^\top$, meaning the d^{th} entry in $\mathcal{L}$ is a function of only the d^{th} row of $\hat{\mathbf{Y}}^\top$ (which represents the fitted values for the d^{th} output variable across observations). For the $(i, d)^{\text{th}}$ entry in $\hat{\mathbf{Y}}^\top$, then, we only need to consider the derivative of the d^{th} entry in $\mathcal{L}$—the derivative of any other entry in $\mathcal{L}$ with respect $\hat{\mathbf{Y}}^\top_{i,d}$ is 0. We can then use the following gradient in place of a four-dimensional tensor.

$$\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}^\top} = \begin{bmatrix} \frac{\partial \mathcal{L}_1}{\partial \hat{\mathbf{Y}}^\top_{1,1}} & \ldots & \frac{\partial \mathcal{L}_1}{\partial \hat{\mathbf{Y}}^\top_{1,N}} \\ & \ldots & \\ \frac{\partial \mathcal{L}_{D_y}}{\partial \hat{\mathbf{Y}}^\top_{D_y,1}} & \ldots & \frac{\partial \mathcal{L}_{D_y}}{\partial \hat{\mathbf{Y}}^\top_{D_y,N}} \end{bmatrix}$$

Next, we consider the derivative of $\hat{\mathbf{Y}}^\top$ with respect to $\mathbf{H}^{(L)}$. Note that $\hat{\mathbf{Y}}^\top$ is an *element-wise* function of $\mathbf{H}^{(L)}$. This means we only need to consider the gradient of each element in the former with respect to its corresponding element in the latter. This gives us

$$\frac{\partial \hat{\mathbf{Y}}^\top}{\partial \mathbf{H}^{(L)}} = \begin{bmatrix} \frac{\partial \hat{\mathbf{Y}}^\top_{1,1}}{\partial \mathbf{H}^{(L)}_{1,1}} & \ldots & \frac{\partial \hat{\mathbf{Y}}^\top_{1,N}}{\partial \mathbf{H}^{(L)}_{1,N}} \\ & \ldots & \\ \frac{\partial \hat{\mathbf{Y}}^\top_{D_y,1}}{\partial \mathbf{H}^{(1)}_{D_y,1}} & \ldots & \frac{\partial \hat{\mathbf{Y}}^\top_{D_y,N}}{\partial \mathbf{L}^{(L)}_{D_y,N}} \end{bmatrix}.$$

Now let's use the shortcut described above. Since each element in $\mathbf{H}^{(L)}$ only affects the corresponding element in $\hat{\mathbf{Y}}^\top$, we calculate $\partial \mathcal{L} / \partial \mathbf{H}^{(L)}$ by multiplying the two gradients above *element-wise*. I.e.,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{H}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}^\top} \circ \frac{\partial \hat{\mathbf{Y}}^\top}{\partial \mathbf{H}^{(L)}},$$

where $\circ$ is the element-wise multiplication operator, also known as the *Hadamard product*.

Next up is $\mathbf{c}^{(L)}$. Whereas each element in $\mathbf{H}^{(L)}$ affected only one element in $\hat{\mathbf{Y}}^\top$, each element in $\mathbf{c}^{(L)}$ affects N elements in $\mathbf{H}^{(L)}$—every element in its corresponding row. Consider the first entry in $\mathbf{c}^{(L)}$. Since this entry affects each entry in the first row of $\mathbf{H}^{(L)}$, the chain rule gives us

$$\frac{\partial \mathcal{L}}{\partial c_1^{(L)}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \mathbf{H}_{1,n}^{(L)}} \cdot \frac{\partial \mathbf{H}_{1,n}^{(L)}}{\partial c_1^{(L)}}.$$

Fortunately $\partial \mathbf{H}_{1,n}^{(L)}/\partial c_1^{(L)}$ is just 1 since $c_1^{(L)}$ is an intercept term. This implies that the derivative of the loss function with respect to $\mathbf{c}^{(1)}$ is just the row sum of $\partial \mathcal{L}/\partial \mathbf{H}^{(L)}$, or

$$\frac{\partial \mathcal{L}}{\partial c_i^{(L)}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \mathbf{H}_{i,n}^{(L)}}.$$

Next, we have $\mathbf{W}^{(L)}$. Using our shortcut, we ask “which values does the $(i, j)^{\text{th}}$ entry in $\mathbf{W}^{(L)}$ affect?” Since $\mathbf{H}^{(L)} = \mathbf{W}^{(L)}\mathbf{Z}^{(L-1)}$, we have that

$$\mathbf{H}_{i,n}^{(L)} = \mathbf{W}_{i,j}^{(L)}\mathbf{Z}_{j,n}^{(L-1)} \; \forall \; n \in \{1, \dots, N\}.$$

This tells us that $\mathbf{W}_{i,j}^{(L)}$ affects each entry in the i^{th} row of $\mathbf{H}^{(L)}$ and gives us the derivative $\partial \mathbf{H}_{i,n}^{(L)}/\partial \mathbf{W}_{i,j}^{(L)} = \mathbf{Z}_{j,n}^{(L-1)}$. Therefore,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{i,j}^{(L)}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \mathbf{H}_{i,n}^{(L)}} \cdot \frac{\partial \mathbf{H}_{i,n}^{(L)}}{\partial \mathbf{W}_{i,j}^{(L)}} = \sum_{n=1}^{N} (\nabla \mathbf{H}^{(L)})_{i,n} \cdot \mathbf{Z}_{j,n}^{(L-1)},$$

where $\nabla \mathbf{H}^{(L)}$ is the matrix representing $\partial \mathcal{L}/\partial \mathbf{H}^{(L)}$. This can be computed for each element in $\mathbf{W}^{(L)}$ using a tensor dot product, which will be covered in the construction section.

Finally, we have $\mathbf{Z}^{(L-1)}$. This case is symmetric to $\mathbf{W}^{(L)}$, and the same approach gives us the result

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}_{i,n}^{(L-1)}} = \sum_{r=1}^{R} \frac{\partial \mathcal{L}}{\partial \mathbf{H}_{r,n}^{(L)}} \cdot \frac{\partial \mathbf{H}_{r,n}^{(L)}}{\partial \mathbf{Z}_{i,n}^{(L-1)}} = \sum_{r=1}^{R} (\nabla \mathbf{H}^{(L)})_{r,n} \cdot \mathbf{W}_{r,i}^{(L)}.$$

Again, the derivative for all of $\mathbf{Z}^{(L-1)}$ can be calculated at once using a tensor dot product.

Construction

In this section, we construct two classes to implement a basic feed-forward neural network. For simplicity, both are limited to one hidden layer, though the number of neurons in the input, hidden, and output layers is flexible. The two differ in how they combine results across observations. The first loops through observations and adds the individual gradients while the second calculates the entire gradient across observatinos in one fell swoop.

Let’s start by importing `numpy`, some visualization packages, and two datasets: the Boston housing and breast cancer datasets from `scikit-learn`. We will use the former for regression and the latter for classification. We also split each dataset into a train and test set. This is done with the hidden code cell below

```python
## Import numpy and visualization packages
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets

## Import Boston and standardize
np.random.seed(123)
boston = datasets.load_boston()
X_boston = boston['data']
X_boston = (X_boston - X_boston.mean(0))/(X_boston.std(0))
y_boston = boston['target']

## Train-test split
np.random.seed(123)
test_frac = 0.25
test_size = int(len(y_boston)*test_frac)
test_idxs = np.random.choice(np.arange(len(y_boston)), test_size, replace = False)
X_boston_train = np.delete(X_boston, test_idxs, 0)
y_boston_train = np.delete(y_boston, test_idxs, 0)
X_boston_test = X_boston[test_idxs]
y_boston_test = y_boston[test_idxs]

## Import cancer and standardize
np.random.seed(123)
cancer = datasets.load_breast_cancer()
X_cancer = cancer['data']
X_cancer = (X_cancer - X_cancer.mean(0))/(X_cancer.std(0))
y_cancer = 1*(cancer['target'] == 1)

## Train-test split
np.random.seed(123)
test_frac = 0.25
test_size = int(len(y_cancer)*test_frac)
test_idxs = np.random.choice(np.arange(len(y_cancer)), test_size, replace = False)
X_cancer_train = np.delete(X_cancer, test_idxs, 0)
y_cancer_train = np.delete(y_cancer, test_idxs, 0)
X_cancer_test = X_cancer[test_idxs]
y_cancer_test = y_cancer[test_idxs]
```

Before constructing classes for our network, let's build our activation functions. Below we implement the ReLU function, sigmoid function, and the linear function (which simply returns its input). Let's also combine these functions into a dictionary so we can identify them with a string argument.

```python
## Activation Functions
def ReLU(h):
    return np.maximum(h, 0)

def sigmoid(h):
    return 1/(1 + np.exp(-h))

def linear(h):
    return h

activation_function_dict = {'ReLU':ReLU, 'sigmoid':sigmoid, 'linear':linear}
```

1. The Loop Approach

Next, we construct a class for fitting feed-forward networks by looping through observations. This class conducts gradient descent by calculating the gradients based on one observation at a time, looping through all observations, and summing the gradients before adjusting the weights.

Once instantiated, we fit a network with the `fit()` method. This method requires training data, the number of nodes for the hidden layer, an activation function for the first and second layers' outputs, a loss function, and some parameters for gradient descent. After storing those values, the method randomly instantiates the network's weights: `W1`, `c1`, `W2`, and `c2`. It then passes the data through this network to instantiate the output values: `h1`, `z1`, `h2`, and `yhat` (equivalent to `z2`).

We then begin conducting gradient descent. Within each iteration of the gradient descent process, we also iterate through the observations. For each observation, we calculate the derivative of the loss for that observation with respect to the network's weights. We then sum these individual derivatives and adjust the weights accordingly, as is typical in gradient descent. The derivatives we calculate are covered in the concept section.

Once the network is fit, we can form predictions with the `predict()` method. This simply consists of running test observations through the network and returning their outputs.

```
class FeedForwardNeuralNetwork:

    def fit(self, X, y, n_hidden, f1 = 'ReLU', f2 = 'linear', loss = 'RSS', lr = 1e-5,
n_iter = 1e3, seed = None):

        ## Store Information
        self.X = X
        self.y = y.reshape(len(y), -1)
        self.N = len(X)
        self.D_X = self.X.shape[1]
        self.D_y = self.y.shape[1]
        self.D_h = n_hidden
        self.f1, self.f2 = f1, f2
        self.loss = loss
        self.lr = lr
        self.n_iter = int(n_iter)
        self.seed = seed

        ## Instantiate Weights
        np.random.seed(self.seed)
        self.W1 = np.random.randn(self.D_h, self.D_X)/5
        self.c1 = np.random.randn(self.D_h, 1)/5
        self.W2 = np.random.randn(self.D_y, self.D_h)/5
        self.c2 = np.random.randn(self.D_y, 1)/5

        ## Instantiate Outputs
        self.h1 = np.dot(self.W1, self.X.T) + self.c1
        self.z1 = activation_function_dict[f1](self.h1)
        self.h2 = np.dot(self.W2, self.z1) + self.c2
        self.yhat = activation_function_dict[f2](self.h2)

        ## Fit Weights
        for iteration in range(self.n_iter):

            dL_dW2 = 0
            dL_dc2 = 0
            dL_dW1 = 0
            dL_dc1 = 0

            for n in range(self.N):

                # dL_dyhat
                if loss == 'RSS':
                    dL_dyhat = -2*(self.y[n] - self.yhat[:,n]).T # (1, D_y)
                elif loss == 'log':
                    dL_dyhat = (-(self.y[n]/self.yhat[:,n]) + (1-self.y[n])/(1-
self.yhat[:,n])).T # (1, D_y)


                ## LAYER 2 ##
                # dyhat_dh2
                if f2 == 'linear':
                    dyhat_dh2 = np.eye(self.D_y) # (D_y, D_y)
                elif f2 == 'sigmoid':
                    dyhat_dh2 = np.diag(sigmoid(self.h2[:,n])*(1-
sigmoid(self.h2[:,n]))) # (D_y, D_y)

                # dh2_dc2
                dh2_dc2 = np.eye(self.D_y) # (D_y, D_y)

                # dh2_dW2
                dh2_dW2 = np.zeros((self.D_y, self.D_y, self.D_h)) # (D_y, (D_y, D_h))
                for i in range(self.D_y):
                    dh2_dW2[i] = self.z1[:,n]

                # dh2_dz1
                dh2_dz1 = self.W2 # (D_y, D_h)


                ## LAYER 1 ##
                # dz1_dh1
                if f1 == 'ReLU':
                    dz1_dh1 = 1*np.diag(self.h1[:,n] > 0) # (D_h, D_h)
                elif f1 == 'linear':
                    dz1_dh1 = np.eye(self.D_h) # (D_h, D_h)


                # dh1_dc1
                dh1_dc1 = np.eye(self.D_h) # (D_h, D_h)

                # dh1_dW1
                dh1_dW1 = np.zeros((self.D_h, self.D_h, self.D_X)) # (D_h, (D_h, D_X))
                for i in range(self.D_h):
                    dh1_dW1[i] = self.X[n]


                ## DERIVATIVES W.R.T. LOSS ##
                dL_dh2 = dL_dyhat @ dyhat_dh2
                dL_dW2 += dL_dh2 @ dh2_dW2
                dL_dc2 += dL_dh2 @ dh2_dc2
                dL_dh1 = dL_dh2 @ dh2_dz1 @ dz1_dh1
                dL_dW1 += dL_dh1 @ dh1_dW1
                dL_dc1 += dL_dh1 @ dh1_dc1

            ## Update Weights
            self.W1 -= self.lr * dL_dW1
            self.c1 -= self.lr * dL_dc1.reshape(-1, 1)
            self.W2 -= self.lr * dL_dW2
            self.c2 -= self.lr * dL_dc2.reshape(-1, 1)

            ## Update Outputs
            self.h1 = np.dot(self.W1, self.X.T) + self.c1
            self.z1 = activation_function_dict[f1](self.h1)
            self.h2 = np.dot(self.W2, self.z1) + self.c2
            self.yhat = activation_function_dict[f2](self.h2)

    def predict(self, X_test):
        self.h1 = np.dot(self.W1, X_test.T) + self.c1
        self.z1 = activation_function_dict[self.f1](self.h1)
        self.h2 = np.dot(self.W2, self.z1) + self.c2
        self.yhat = activation_function_dict[self.f2](self.h2)
        return self.yhat
```

Let's try building a network with this class using the `boston` housing data. This network contains 8 neurons in its hidden layer and uses the ReLU and linear activation functions after the first and second layers, respectively.

```
ffnn = FeedForwardNeuralNetwork()
ffnn.fit(X_boston_train, y_boston_train, n_hidden = 8)
y_boston_test_hat = ffnn.predict(X_boston_test)

fig, ax = plt.subplots()
sns.scatterplot(y_boston_test, y_boston_test_hat[0])
ax.set(xlabel = r'$y$', ylabel = r'$\hat{y}$', title = r'$y$ vs. $\hat{y}$')
sns.despine()
```

../../_images/construction_9_0.png

We can also build a network for binary classification. The model below attempts to predict whether an individual's cancer is malignant or benign. We use the log loss, the sigmoid activation function after the second layer, and the ReLU function after the first.

```
ffnn = FeedForwardNeuralNetwork()
ffnn.fit(X_cancer_train, y_cancer_train, n_hidden = 8,
         loss = 'log', f2 = 'sigmoid', seed = 123, lr = 1e-4)
y_cancer_test_hat = ffnn.predict(X_cancer_test)
np.mean(y_cancer_test_hat.round() == y_cancer_test)
```

```
0.9929577464788732
```

2. The Matrix Approach

Below is a second class for fitting neural networks that runs *much* faster by simultaneously calculating the gradients across observations. The math behind these calculations is outlined in the concept section. This class's fitting algorithm is identical to that of the one above with one big exception: we don't have to iterate over observations.

Most of the following gradient calculations are straightforward. A few require a tensor dot product, which is easily done using numpy. Consider the following gradient:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}_{i,j}} = \sum_{n=1}^{N} (\nabla \mathbf{H}^{(L)})_{i,n} \cdot \mathbf{Z}^{(L-1)}_{j,n}.$$

In words, $\partial \mathcal{L}/\partial \mathbf{W}^{(L)}$ is a matrix whose $(i, j)^{\text{th}}$ entry equals the sum across the i^{th} row of $\nabla \mathbf{H}^{(L)}$ multiplied element-wise with the j^{th} row of $\mathbf{Z}^{(L-1)}$.

This calculation can be accomplished with `np.tensordot(A, B, (1,1))`, where `A` is $\nabla \mathbf{H}^{(L)}$ and `B` is $\mathbf{Z}^{(L-1)}$. `np.tensordot()` sums the element-wise product of the entries in `A` and the entries in `B` along a specified index. Here we specify the index with `(1,1)`, saying we want to sum across the columns for each.

Similarly, we will use the following gradient:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}_{i,n}^{(L-1)}} = \sum_{d=1}^{D_y} (\nabla \mathbf{H}^{(L)})_{d,n} \cdot \mathbf{W}_{d,i}^{(L)}.$$

Letting `C` represent $\mathbf{W}^{(L)}$, we can calculate this gradient in numpy with `np.tensordot(C, A, (0,0))`.

```python
class FeedForwardNeuralNetwork:


    def fit(self, X, Y, n_hidden, f1 = 'ReLU', f2 = 'linear', loss = 'RSS', lr = 1e-5,
n_iter = 5e3, seed = None):

        ## Store Information
        self.X = X
        self.Y = Y.reshape(len(Y), -1)
        self.N = len(X)
        self.D_X = self.X.shape[1]
        self.D_Y = self.Y.shape[1]
        self.Xt = self.X.T
        self.Yt = self.Y.T
        self.D_h = n_hidden
        self.f1, self.f2 = f1, f2
        self.loss = loss
        self.lr = lr
        self.n_iter = int(n_iter)
        self.seed = seed

        ## Instantiate Weights
        np.random.seed(self.seed)
        self.W1 = np.random.randn(self.D_h, self.D_X)/5
        self.c1 = np.random.randn(self.D_h, 1)/5
        self.W2 = np.random.randn(self.D_Y, self.D_h)/5
        self.c2 = np.random.randn(self.D_Y, 1)/5

        ## Instantiate Outputs
        self.H1 = (self.W1 @ self.Xt) + self.c1
        self.Z1 = activation_function_dict[self.f1](self.H1)
        self.H2 = (self.W2 @ self.Z1) + self.c2
        self.Yhatt = activation_function_dict[self.f2](self.H2)

        ## Fit Weights
        for iteration in range(self.n_iter):

            # Yhat #
            if self.loss == 'RSS':
                self.dL_dYhatt = -(self.Yt - self.Yhatt) # (D_Y x N)
            elif self.loss == 'log':
                self.dL_dYhatt = (-(self.Yt/self.Yhatt) + (1-self.Yt)/(1-self.Yhatt)) #
(D_y x N)

            # H2 #
            if self.f2 == 'linear':
                self.dYhatt_dH2 = np.ones((self.D_Y, self.N))
            elif self.f2 == 'sigmoid':
                self.dYhatt_dH2 = sigmoid(self.H2) * (1- sigmoid(self.H2))
            self.dL_dH2 = self.dL_dYhatt * self.dYhatt_dH2 # (D_Y x N)

            # c2 #
            self.dL_dc2 = np.sum(self.dL_dH2, 1) # (D_y)

            # W2 #
            self.dL_dW2 = np.tensordot(self.dL_dH2, self.Z1, (1,1)) # (D_Y x D_h)

            # Z1 #
            self.dL_dZ1 = np.tensordot(self.W2, self.dL_dH2, (0, 0)) # (D_h x N)

            # H1 #
            if self.f1 == 'ReLU':
                self.dL_dH1 = self.dL_dZ1 * np.maximum(self.H1, 0) # (D_h x N)
            elif self.f1 == 'linear':
                self.dL_dH1 = self.dL_dZ1 # (D_h x N)

            # c1 #
            self.dL_dc1 = np.sum(self.dL_dH1, 1) # (D_h)

            # W1 #
            self.dL_dW1 = np.tensordot(self.dL_dH1, self.Xt, (1,1)) # (D_h, D_X)

            ## Update Weights
            self.W1 -= self.lr * self.dL_dW1
            self.c1 -= self.lr * self.dL_dc1.reshape(-1, 1)
            self.W2 -= self.lr * self.dL_dW2
            self.c2 -= self.lr * self.dL_dc2.reshape(-1, 1)

            ## Update Outputs
            self.H1 = (self.W1 @ self.Xt) + self.c1
            self.Z1 = activation_function_dict[self.f1](self.H1)
            self.H2 = (self.W2 @ self.Z1) + self.c2
            self.Yhatt = activation_function_dict[self.f2](self.H2)

    def predict(self, X_test):
        X_testt = X_test.T
        self.h1 = (self.W1 @ X_testt) + self.c1
        self.z1 = activation_function_dict[self.f1](self.h1)
        self.h2 = (self.W2 @ self.z1) + self.c2
        self.Yhatt = activation_function_dict[self.f2](self.h2)
        return self.Yhatt
```

We fit networks of this class in the same way as before. Examples of regression with the `boston` housing data and classification with the `breast_cancer` data are shown below.

```
ffnn = FeedForwardNeuralNetwork()
ffnn.fit(X_boston_train, y_boston_train, n_hidden = 8)
y_boston_test_hat = ffnn.predict(X_boston_test)

fig, ax = plt.subplots()
sns.scatterplot(y_boston_test, y_boston_test_hat[0])
ax.set(xlabel = r'$y$', ylabel = r'$\hat{y}$', title = r'$y$ vs. $\hat{y}$')
sns.despine()
```

../../_images/construction_16_01.png

```
ffnn = FeedForwardNeuralNetwork()
ffnn.fit(X_cancer_train, y_cancer_train, n_hidden = 8,
         loss = 'log', f2 = 'sigmoid', seed = 123, lr = 1e-4)
y_cancer_test_hat = ffnn.predict(X_cancer_test)
np.mean(y_cancer_test_hat.round() == y_cancer_test)
```

```
0.9929577464788732
```

Implementation

Several Python libraries allow for easy and efficient implementation of neural networks. Here, we'll show examples with the very popular `tf.keras` submodule. This submodule integrates Keras, a user-friendly high-level API, into Tensorflow, a lower-level backend. Let's start by loading Tensorflow, our visualization packages, and the Boston housing dataset from `scikit-learn`.

```
import tensorflow as tf
from sklearn import datasets
import matplotlib.pyplot as plt
import seaborn as sns

boston = datasets.load_boston()
X_boston = boston['data']
y_boston = boston['target']
```

Neural networks in Keras can be fit through one of two APIs: the *sequential* or the *functional* API. For the type of models discussed in this chapter, either approach works.

1. The Sequential API

Fitting a network with the Keras sequential API can be broken down into four steps:

1. Instantiate model
2. Add layers
3. Compile model (and summarize)
4. Fit model

An example of the code for these four steps is shown below. We first instantiate the network using `tf.keras.models.Sequential()`.

Next, we add layers to the network. Specifically, we have to add any hidden layers we like followed by a single output layer. The type of networks covered in this chapter use only `Dense` layers. A "dense" layer is one in which each neuron is a function of all the other neurons in the previous layer. We identify the number of neurons in the layer with the `units` argument and the activation function applied to the layer with the `activation` argument. For the first layer only, we must also identify the `input_shape`, or the number of neurons in the input layer. If our predictors are of length `D`, the input shape will be `(D, )` (which is the shape of a single observation, as we can see with `X[0].shape`).

The next step is to compile the model. Compiling determines the configuration of the model; we specify the optimizer and loss function to be used as well as any metrics we would like to monitor. After compiling, we can also preview our model with `model.summary()`.

Finally, we fit the model. Here is where we actually provide our training data. Two other important arguments are `epochs` and `batch_size`. Models in Keras are fit with *mini-batch gradient descent*, in which samples of the training data are looped through and individually used to calculate and update gradients. `batch_size` determines the size of these samples, and `epochs` determines how many times the gradient is calculated for each sample.

```
## 1. Instantiate
model = tf.keras.models.Sequential(name = 'Sequential_Model')

## 2. Add Layers
model.add(tf.keras.layers.Dense(units = 8,
                                activation = 'relu',
                                input_shape = (X_boston.shape[1], ),
                                name = 'hidden'))
model.add(tf.keras.layers.Dense(units = 1,
                                activation = 'linear',
                                name = 'output'))

## 3. Compile (and summarize)
model.compile(optimizer = 'adam', loss = 'mse')
print(model.summary())

## 4. Fit
model.fit(X_boston, y_boston, epochs = 100, batch_size = 1, validation_split=0.2,
verbose = 0);
```

```
Model: "Sequential_Model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden (Dense)               (None, 8)                 112
_________________________________________________________________
output (Dense)               (None, 1)                 9
=================================================================
Total params: 121
Trainable params: 121
Non-trainable params: 0
_________________________________________________________________
None
```

Predictions with the model built above are shown below.

```
# Create Predictions
yhat_boston = model.predict(X_boston)[:,0]

# Plot
fig, ax = plt.subplots()
sns.scatterplot(y_boston, yhat_boston)
ax.set(xlabel = r"$y$", ylabel = r"$\hat{y}$", title = r"$y$ vs. $\hat{y}$")
sns.despine()
```

../../_images/code_8_01.png

2. The Functional API

Fitting models with the Functional API can again be broken into four steps, listed below.

1. Define layers
2. Define model
3. Compile model (and summarize)
4. Fit model

While the sequential approach first defines the model and then adds layers, the functional approach does the opposite. We start by adding an input layer using `tf.keras.Input()`. Next, we add one or more hidden layers using `tf.keras.layers.Dense()`. Note that in this approach, we link layers directly. For instance, we indicate that the `hidden` layer below follows the `inputs` layer by adding `(inputs)` to the end of its definition.

After creating the layers, we can define our model. We do this by using `tf.keras.Model()` and identifying the input and output layers. Finally, we compile and fit our model as in the sequential API.

```python
## 1. Define layers
inputs = tf.keras.Input(shape = (X_boston.shape[1],), name = "input")
hidden = tf.keras.layers.Dense(8, activation = "relu", name = "first_hidden")(inputs)
outputs = tf.keras.layers.Dense(1, activation = "linear", name = "output")(hidden)

## 2. Model
model = tf.keras.Model(inputs = inputs, outputs = outputs, name = "Functional_Model")

## 3. Compile (and summarize)
model.compile(optimizer = "adam", loss = "mse")
print(model.summary())

## 4. Fit
model.fit(X_boston, y_boston, epochs = 100, batch_size = 1, validation_split=0.2,
verbose = 0);
```

```
Model: "Functional_Model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input (InputLayer)           [(None, 13)]              0
_________________________________________________________________
first_hidden (Dense)         (None, 8)                 112
_________________________________________________________________
output (Dense)               (None, 1)                 9
=================================================================
Total params: 121
Trainable params: 121
Non-trainable params: 0
_________________________________________________________________
None
```

Predictions formed with this model are shown below.

```python
# Create Predictions
yhat_boston = model.predict(X_boston)[:,0]

# Plot
fig, ax = plt.subplots()
sns.scatterplot(y_boston, yhat_boston)
ax.set(xlabel = r"$y$", ylabel = r"$\hat{y}$", title = r"$y$ vs. $\hat{y}$")
sns.despine()
```

../../_images/code_13_0.png

Math

For a book on mathematical derivations, this text assumes knowledge of relatively few mathematical methods. Most of the mathematical background required is summarized in the three following sections on calculus, matrices, and matrix calculus.

Calculus

The most important mathematical prerequisite for this book is calculus. Almost all of the methods covered involve minimizing a loss function or maximizing a likelihood function, done by taking the function's derivative with respect to one or more parameters and setting it equal to 0.

Let's start by reviewing some of the most common derivatives used in this book:

$$
\begin{aligned}
f(x) &= x^a \rightarrow f'(x) = ax^{a-1} \\
f(x) &= \exp(x) \rightarrow f'(x) = \exp(x) \\
f(x) &= \log(x) \rightarrow f'(x) = \frac{1}{x} \\
f(x) &= |x| \rightarrow f'(x) = \begin{cases} 1, & x > 0 \\ -1, & x < 0, \end{cases}
\end{aligned}
$$

We will also often use the sum, product, and quotient rules:

$$
\begin{aligned}
f(x) &= g(x) + h(x) \rightarrow f'(x) = g'(x) + h'(x) \\
f(x) &= g(x) \cdot h(x) \rightarrow f'(x) = g'(x)h(x) + g(x)h'(x) \\
f(x) &= g(x)/h(x) \rightarrow f'(x) = \frac{h(x)g'(x) + g(x)h'(x)}{h(x)^2}.
\end{aligned}
$$

Finally, we will heavily rely on the chain rule:

$$
f(x) = g(h(x)) \rightarrow f'(x) = g'(h(x))h'(x).
$$

Matrices

While little linear algebra is used in this book, matrix and vector representations of data are very common. The most important matrix and vector operations are reviewed below.

Let $\mathbf{u}$ and $\mathbf{v}$ be two column vectors of length D. The **dot product** of $\mathbf{u}$ and $\mathbf{v}$ is a scalar value given by

$$
\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^\top \mathbf{v} = \sum_{d=1}^{D} u_d v_d = u_1 v_1 + u_2 v_2 + \cdots + u_D v_D.
$$

If $\mathbf{v}$ is a vector of features (with a leading 1 appended for the intercept term) and $\mathbf{u}$ is a vector of weights, this dot product is also referred to as a *linear combination* of the predictors in $\mathbf{v}$.

The **L1 norm** and **L2 norm** measure a vector's magnitude. For a vector $\mathbf{u}$, these are given respectively by

$$
\begin{aligned}
||\mathbf{u}||_1 &= \sum_{d=1}^{D} |u_d| \\
||\mathbf{u}||_2 &= \sqrt{\sum_{d=1}^{D} u_d^2}.
\end{aligned}
$$

Let $\mathbf{A}$ be a $(N \times D)$ matrix defined as

$$
\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1D} \\ A_{21} & A_{22} & \dots & A_{2D} \\ & & \dots & \\ A_{N1} & A_{N2} & \dots & A_{ND} \end{pmatrix}
$$

The transpose of $\mathbf{A}$ is a $(D \times N)$ matrix given by

$$
\mathbf{A}^T = \begin{pmatrix} A_{11} & A_{21} & \dots & A_{N1} \\ A_{12} & A_{22} & \dots & A_{N2} \\ & & \dots & \\ A_{1D} & A_{2D} & \dots & A_{ND} \end{pmatrix}
$$

If $\mathbf{A}$ is a square $(N \times N)$ matrix, its inverse, given by $\mathbf{A}^{-1}$, is the matrix such that

$$
\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = I_N.
$$

Matrix Calculus

Dealing with multiple parameters, multiple observations, and sometimes multiple loss functions, we will often have to take multiple derivatives at once in this book. This is done with matrix calculus.

In this book, we will use the numerator layout convention for matrix derivatives. This is most easily shown with examples. First, let a be a scalar and $\mathbf{u}$ be a vector of length I. The derivative of a with respect to $\mathbf{u}$ is given by

$$
\frac{\partial a}{\partial \mathbf{u}} = \begin{pmatrix} \frac{\partial a}{\partial u_1} & \cdot\cdot & \frac{\partial a}{\partial u_I} \end{pmatrix} \in \mathbb{R}^I,
$$

and the derivative of $\mathbf{u}$ with respect to a is given by

$$
\frac{\partial \mathbf{u}}{\partial a} = \begin{pmatrix} \frac{\partial u_1}{\partial a} \\ \dots \\ \frac{\partial u_I}{\partial a} \end{pmatrix} \mathbb{R}^I.
$$

Note that in either case, the first dimension of the derivative is determined by what's in the numerator. Similarly, letting $\mathbf{v}$ be a vector of length J, the derivative of $\mathbf{u}$ with respect to $\mathbf{v}$ is given with

$$\frac{\partial \mathbf{u}}{\partial \mathbf{v}} = \begin{pmatrix} \frac{\partial u_1}{\partial v_1} & \dots & \frac{\partial u_1}{\partial v_J} \\ & \dots & \\ \frac{\partial u_I}{\partial v_1} & \dots & \frac{\partial u_I}{\partial v_J} \end{pmatrix} \in \mathbb{R}^{I \times J}.$$

We will also have to take derivatives of or with respect to matrices. Let $\mathbf{X}$ be a $(N \times D)$ matrix. The derivative of $\mathbf{X}$ with respect to a constant a is given by

$$\frac{\partial \mathbf{X}}{\partial a} = \begin{pmatrix} \frac{\partial X_{11}}{\partial a} & \dots & \frac{\partial X_{1D}}{\partial a} \\ & \dots & \\ \frac{\partial X_{N1}}{\partial a} & \dots & \frac{\partial X_{ND}}{\partial a} \end{pmatrix} \in \mathbb{R}^{N \times D},$$

and conversely the derivative of a with respect to $\mathbf{X}$ is given by

$$\frac{\partial a}{\partial \mathbf{X}} = \begin{pmatrix} \frac{\partial a}{\partial X_{11}} & \dots & \frac{\partial a}{\partial X_{1D}} \\ & \dots & \\ \frac{\partial a}{\partial X_{N1}} & \dots & \frac{\partial a}{\partial X_{ND}} \end{pmatrix} \in \mathbb{R}^{N \times D}.$$

Finally, we will occasionally need to take derivatives of vectors with respect to matrices or vice versa. This results in a *tensor* of 3 or more dimensions. Two examples are given below. First, the derivative of $\mathbf{u} \in \mathbb{R}^I$ with respect to $\mathbf{X} \in \mathbb{R}^{N \times D}$ is given by

$$\frac{\partial \mathbf{u}}{\partial \mathbf{X}} = \left(\begin{pmatrix} \frac{\partial u_1}{\partial X_{11}} & \dots & \frac{\partial u_1}{\partial X_{1D}} \\ & \dots & \\ \frac{\partial u_1}{\partial X_{N1}} & \dots & \frac{\partial u_1}{\partial X_{ND}} \end{pmatrix} \quad \dots \quad \begin{pmatrix} \frac{\partial u_I}{\partial X_{11}} & \dots & \frac{\partial u_I}{\partial X_{1D}} \\ & \dots & \\ \frac{\partial u_I}{\partial X_{N1}} & \dots & \frac{\partial u_I}{\partial X_{ND}} \end{pmatrix} \right) \in \mathbb{R}^{I \times N \times D},$$

and the derivative of $\mathbf{X}$ with respect to $\mathbf{u}$ is given by

$$\frac{\partial \mathbf{X}}{\partial \mathbf{u}} = \begin{pmatrix} \left(\frac{\partial X_{11}}{\partial u_1} \quad \dots \quad \frac{\partial X_{11}}{\partial u_I} \right) & \dots & \left(\frac{\partial X_{1D}}{\partial u_1} \quad \dots \quad \frac{\partial X_{1D}}{\partial u_I} \right) \\ & \dots & \\ \left(\frac{\partial X_{N1}}{\partial u_1} \quad \dots \quad \frac{\partial X_{N1}}{\partial u_I} \right) & \dots & \left(\frac{\partial X_{ND}}{\partial u_1} \quad \dots \quad \frac{\partial X_{ND}}{\partial u_I} \right) \end{pmatrix} \in \mathbb{R}^{N \times D \times I}.$$

Notice again that what we are taking the derivative *of* determines the first dimension(s) of the derivative and what we are taking the derivative with respect *to* determines the last.

Probability

Many machine learning methods are rooted in probability theory. Probabilistic methods in this book include linear regression, Bayesian regression, and generative classifiers. This section covers the probability theory needed to understand those methods.

1. Random Variables and Distributions

Random Variables

A **random variable** is a variable whose value is randomly determined. The set of possible values a random variable can take on is called the variable's **support**. An example of a random variable is the value on a die roll. This variable's support is $\{1, 2, 3, 4, 5, 6\}$. Random variables will be represented with uppercase letters and values in their support with lowercase letters. For instance $X = x$ implies that a random variable X happened to take on value x. Letting X be the value of a die roll, $X = 4$ indicates that the die landed on 4.

Density Functions

The likelihood that a random variable takes on a given value is determined through its density function. For a discrete random variable (one that can take on a finite set of values), this density function is called the **probability mass function (PMF)**. The PMF of a random variable X gives the probability that X will equal some value x. We write it as $f_X(x)$ or just $f(x)$, and it is defined as

$$f(x) = P(X = x).$$

For a continuous random variable (one that can take on infinitely many values), the density function is called the **probability density function (PDF)**. The PDF $f_X(x)$ of a continuous random variable X does not give $P(X = x)$ but it does determine the probability that X lands in a certain range. Specifically,

$$P(a \leq X \leq b) = \int_{x=a}^{b} f(x)dx.$$

That is, integrating $f(x)$ over a certain range gives the probability of X being in that range. While $f(x)$ does not give the probability that X will equal a certain value, it does indicate the relative likelihood that it will be *around* that value. E.g. if $f(a) > f(b)$, we can say X is more likely to be in an arbitrarily small area around the value a than around the value b.

Distributions

A random variable's **distribution** is determined by its density function. Variables with the same density function are said to follow the same distributions. Certain families of distributions are very common in probability and machine learning. Two examples are given below.

The **Bernoulli** distribution is the most simple probability distribution and it describes the likelihood of the outcomes of a binary event. Let X be a random variable that equals 1 (representing "success") with probability p and 0 (representing "failure") with probability $1 - p$. Then, X is said to follow the Bernoulli distribution with probability parameter p, written $X \sim \text{Bern}(p)$, and its PMF is given by

$$f_X(x) = p^x(1-p)^{(1-x)}.$$

We can check to see that for any valid value x in the support of X—i.e., 1 or 0—, $f(x)$ gives $P(X = x)$.

The **Normal** distribution is extremely common and will be used throughout this book. A random variable X follows the Normal distribution with mean parameter $\mu \in \mathbb{R}$ and variance parameter $\sigma^2 > 0$, written $X \sim \mathcal{N}(\mu, \sigma^2)$, if its PDF is defined as

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

The shape of the Normal random variable's density function gives this distribution the name "the bell curve", as shown below. Values closest to μ are most likely and the density is symmetric around μ.

normal

Independence

So far we've discussed the density of individual random variables. The picture can get much more complicated when we want to study the behavior of multiple random variables simultaneously. The assumption of independence simplifies things greatly. Let's start by defining independence in the discrete case.

Two discrete random variables X and Y are **independent** if and only if

$$P(X = x, Y = y) = P(X = x)P(Y = y),$$

for all x and y. This says that if X and Y are independent, the probability that $X = x$ and $Y = y$ simultaneously is just the product of the probabilities that $X = x$ and $Y = y$ individually.

To generalize this definition to continuous random variables, let's first introduce *joint density function*. Quite simply, the joint density of two random variables X and Y, written $f_{X,Y}(x, y)$ gives the probability density of X and Y evaluated simultaneously at x and y, respectively. We can then say that X and Y are independent if and only if

$$f_{X,Y}(x, y) = f_X(x)f_Y(y),$$

for all x and y.

2. Maximum Likelihood Estimation

Maximum likelihood estimation is used to understand the parameters of a distribution that gave rise to observed data. In order to model a data generating process, we often assume it comes from some family of distributions, such as the Bernoulli or Normal distributions. These distributions are indexed by certain parameters (p for the Bernoulli and μ and σ^2 for the Normal)—maximum likelihood estimation evaluates which parameters would be most consistent with the data we observed.

Specifically, maximum likelihood estimation finds the values of unknown parameters that maximize the probability of observing the data we did. Basic maximum likelihood estimation can be broken into three steps:

1. Find the joint density of the observed data, also called the *likelihood*
2. Take the log of the likelihood, giving the *log-likelihood*.
3. Find the value of the parameter that maximizes the log-likelihood (and therefore the likelihood as well) by setting its derivative equal to 0.

Finding the value of the parameter to maximize the log-likelihood rather than the likelihood makes the math easier and gives us the same solution.

Let's go through an example. Suppose we are interested in calculating the average weight of a Chihuahua. We assume the weight of any given Chihuahua is *independently* distributed Normally with $\sigma^2 = 1$ but an unknown mean μ. So, we gather 10 Chihuahuas and weigh them. Denote the j^{th} Chihuahua weight with $W_j \sim \mathcal{N}(\mu, 1)$. For step 1, let's calculate the probability density of our data (i.e., the 10 Chihuahua weights). Since the weights are assumed to be independent, the densities multiply. Letting $L(\mu)$ be the likelihood of μ, we have

$$\begin{aligned} L(\mu) &= f_{W_1,\dots,W_{10}}(w_1, \dots, w_{10}) \\ &= f_{W_1}(w_1) \cdot \ldots \cdot f_{W_{10}}(w_{10}) \\ &= \prod_{j=1}^{10} \frac{1}{\sqrt{2\pi \cdot 1}} \exp\left(-\frac{(w_j - \mu)^2}{2}\right) \\ &\propto \exp\left(-\sum_{j=1}^{10} \frac{(w_j - \mu)^2}{2}\right). \end{aligned}$$

Note that we can work up to a constant of proportionality since the value of μ that maximizes $L(\mu)$ will also maximize anything proportional to $L(\mu)$. For step 2, take the log:

$$\log L(\mu) = -\sum_{j=1}^{10} \frac{(w_j - \mu)^2}{2} + c,$$

where c is some constant. For step 3, take the derivative:

$$\frac{\partial}{\partial \mu} \log L(\mu) = -\sum_{j=1}^{10} (w_j - \mu).$$

Setting this equal to 0, we find that the (log) likelihood is maximized with

$$\hat{\mu} = \frac{1}{10} \sum_{j=1}^{10} w_j = \bar{w}.$$

We put a hat over μ to indicate that it is our *estimate* of the true μ. Note the sensible result—we estimate the true mean of the Chihuahua weight distribution to be the sample mean of our observed data.

3. Conditional Probability

Probabilistic machine learning methods typically consider the distribution of a target variable conditional on the value of one or more predictor variables. To understand these methods, let's introduce some of the basic principles of conditional probability.

Consider two events, A and B. The **conditional probability** of A given B is the probability that A occurs given B occurs, written $P(A|B)$. Closely related is the **joint probability** of A and B, or the probability that both A and B occur, written $P(A, B)$. We navigate between the conditional and joint probability with the following

$$P(A, B) = P(A|B)P(B).$$

The above equation leads to an extremely important principle in conditional probability: Bayes' rule. **Bayes' rule** states that

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

Both of the above expressions work for random variables as well as events. For any two discrete random variables, X and Y

$$\begin{aligned} P(X = x, Y = y) &= P(X = x|Y = y)P(Y = y) \\ P(X = x|Y = y) &= \frac{P(Y = y|X = x)P(X = x)}{P(Y = y)}. \end{aligned}$$

The same is true for continuous random variables, replacing the PMFs with PDFs.

Common Methods

This section will review two methods that are used to fit a variety of machine learning models: *gradient descent* and *cross validation*. These methods will be used repeatedly throughout this book.

1. Gradient Descent

Almost all the models discussed in this book aim to find a set of parameters that minimize a chosen loss function. Sometimes we can find the optimal parameters by taking the derivative of the loss function, setting it equal to 0, and solving. In situations for which no closed-form solution is available, however, we might turn to gradient descent. **Gradient descent** is an iterative approach to approximating the parameters that minimize a differentiable loss function.

The Set-Up

Let's first introduce a typical set-up for gradient descent. Suppose we have N observations where each observation has predictors $\mathbf{x}_n$ and target variable y_n. We decide to approximate y_n with $\hat{y}_n = f(\mathbf{x}_n, \hat{\boldsymbol{\beta}})$, where $f()$ is some differentiable function and $\hat{\boldsymbol{\beta}}$ is a set of parameter estimates. Next, we introduce a differentiable loss function $\mathcal{L}$. For simplicity, let's assume we can write the model's entire loss as the sum of the individual losses across observations. That is,

$$\mathcal{L} = \sum_{n=1}^{N} g(y_n, \hat{y}_n),$$

where $g()$ is some differentiable function representing an observation's individual loss.

To fit this generic model, we want to find the values of $\hat{\boldsymbol{\beta}}$ that minimize $\mathcal{L}$. We will likely start with the following derivative:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{\beta}}} &= \sum_{n=1}^{N} \frac{\partial g(y_n, \hat{y}_n)}{\partial \hat{\boldsymbol{\beta}}} \\ &= \sum_{n=1}^{N} \frac{\partial g(y_n, \hat{y}_n)}{\partial \hat{y}_n} \cdot \frac{\partial \hat{y}_n}{\partial \hat{\boldsymbol{\beta}}}. \end{aligned}$$

Ideally, we can set the above derivative equal to 0 and solve for $\hat{\boldsymbol{\beta}}$, giving our optimal solution. If this isn't possible, we can iteratively search for the values of $\hat{\boldsymbol{\beta}}$ that minimize $\mathcal{L}$. This is the process of gradient descent.

An Intuitive Introduction

gd

To understand this process intuitively, consider the image above showing a model's loss as a function of one parameter, β. We start our search for the optimal β by randomly picking a value. Suppose we start with β at point A. From point A we ask "would the loss function decrease if I increased or decreased β". To answer this question, we calculate the derivative of $\mathcal{L}$ with respect to β evaluated at $\beta = A$. Since this derivative is negative, we know that increasing β some small amount will decrease the loss.

Now we know we want to increase β, but how much? Intuitively, the more negative the derivative, the more the loss will decrease with an increase in β. So, let's increase β by an amount proportional to the negative of the derivative. Letting δ be the derivative and η be a small constant learning rate, we might increase β with

$$\beta \leftarrow \beta - \eta\delta.$$

The more negative δ is, the more we increase β.

Now suppose we make the increase and wind up with $\beta = B$. Calculating the derivative again, we get a slightly positive number. This tells us that we went too far: increasing β will increase $\mathcal{L}$. However, since the derivative is only *slightly* positive, we want to only make a slight correction. Let's again use the same adjustment, $\beta \leftarrow \beta - \eta\delta$. Since δ is now slightly positive, β will now decrease slightly. We will repeat this same process a fixed number of times or until β barely changes. And that is gradient descent!

The Steps

We can describe gradient descent more concretely with the following steps. Note here that $\hat{\boldsymbol{\beta}}$ can be a vector, rather than just a single parameter.

1. Choose a small learning rate η
2. Randomly instantiate $\hat{\boldsymbol{\beta}}$
3. For a fixed number of iterations or until some stopping rule is reached:
    1. Calculate $\boldsymbol{\delta} = \partial\mathcal{L}/\partial\hat{\boldsymbol{\beta}}$
    2. Adjust $\hat{\boldsymbol{\beta}}$ with

$$\hat{\boldsymbol{\beta}} \leftarrow \hat{\boldsymbol{\beta}} - \eta\boldsymbol{\delta}.$$

A potential stopping rule might be a minimum change in the magnitude of $\hat{\boldsymbol{\beta}}$ or a minimum decrease in the loss function $\mathcal{L}$.

An Example

As a simple example of gradient descent in action, let's derive the ordinary least squares (OLS) regression estimates. (This problem does have a closed-form solution, but we'll use gradient descent to demonstrate the approach). As discussed in Chapter 1, linear regression models $\hat{y}_n$ with

$$\hat{y}_n = \mathbf{x}_n^\top\hat{\boldsymbol{\beta}},$$

where $\mathbf{x}_n$ is a vector of predictors appended with a leading 1 and $\hat{\boldsymbol{\beta}}$ is a vector of coefficients. The OLS loss function is defined with

$$\mathcal{L}(\hat{\boldsymbol{\beta}}) = \frac{1}{2}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2 = \frac{1}{2}\sum_{n=1}^{N}(y_n - \mathbf{x}_n^\top\hat{\boldsymbol{\beta}})^2.$$

After choosing η and randomly instantiating $\hat{\boldsymbol{\beta}}$, we iteratively calculate the loss function's gradient:

$$\boldsymbol{\delta} = \frac{\partial\mathcal{L}(\hat{\boldsymbol{\beta}})}{\partial\hat{\boldsymbol{\beta}}} = -\sum_{n=1}^{N}(y_n - \mathbf{x}_n^\top\hat{\boldsymbol{\beta}}) \cdot \boldsymbol{\phi}_n^\top,$$

and adjust with

$$\hat{\boldsymbol{\beta}} \leftarrow \hat{\boldsymbol{\beta}} - \eta\boldsymbol{\delta}.$$

This is accomplished with the following code. Note that we can also calculate $\boldsymbol{\delta} = -\mathbf{X}^\top(\mathbf{y} - \hat{\mathbf{y}})$, where $\mathbf{X}$ is the feature matrix, $\mathbf{y}$ is the vector of targets, and $\hat{\mathbf{y}}$ is the vector of fitted values.

```python
import numpy as np

def OLS_GD(X, y, eta = 1e-3, n_iter = 1e4, add_intercept = True):

  ## Add Intercept
  if add_intercept:
    ones = np.ones(X.shape[0]).reshape(-1, 1)
    X = np.concatenate((ones, X), 1)

  ## Instantiate
  beta_hat = np.random.randn(X.shape[1])

  ## Iterate
  for i in range(int(n_iter)):

    ## Calculate Derivative
    yhat = X @ beta_hat
    delta = -X.T @ (y - yhat)
    beta_hat -= delta*eta
```

2. Cross Validation

Several of the models covered in this book require *hyperparameters* to be chosen exogenously (i.e. before the model is fit). The value of these hyperparameters affects the quality of the model's fit. So how can we choose these values without fitting a model? The most common answer is cross validation.

Suppose we are deciding between several values of a hyperparameter, resulting in multiple competing models. One way to choose our model would be to split our data into a *training* set and a *validation* set, build each model on the training set, and see which performs better on the validation set. By splitting the data into training and validation, we avoid evaluating a model based on its in-sample performance.

The obvious problem with this set-up is that we are comparing the performance of models on just *one* dataset. Instead, we might choose between competing models with **K-fold cross validation**, outlined below.

1. Split the original dataset into K *folds* or subsets.
2. For $k = 1, \dots, K$, treat fold k as the validation set. Train each competing model on the data from the other $K - 1$ folds and evaluate it on the data from the k^{th}.
3. Select the model with the best average validation performance.

As an example, let's use cross validation to choose a penalty value for a Ridge regression model, discussed in chapter 2. This model constrains the magnitude of the regression coefficients; the higher the penalty term, the more the coefficients are constrained.

The example below uses the `Ridge` class from `scikit-learn`, which defines the penalty term with the `alpha` argument. We will use the Boston housing dataset.

```
## Import packages
import numpy as np
from sklearn.linear_model import Ridge
from sklearn.datasets import load_boston

## Import data
boston = load_boston()
X = boston['data']
y = boston['target']
N = X.shape[0]

## Choose alphas to consider
potential_alphas = [0, 1, 10]
error_by_alpha = np.zeros(len(potential_alphas))

## Choose the folds
K = 5
indices = np.arange(N)
np.random.shuffle(indices)
folds = np.array_split(indices, K)

## Iterate through folds
for k in range(K):

  ## Split Train and Validation
    X_train = np.delete(X, folds[k], 0)
    y_train = np.delete(y, folds[k], 0)
    X_val = X[folds[k]]
    y_val = y[folds[k]]

  ## Iterate through Alphas
    for i in range(len(potential_alphas)):

        ## Train on Training Set
        model = Ridge(alpha = potential_alphas[i])
        model.fit(X_train, y_train)

        ## Calculate and Append Error
        error = np.sum( (y_val - model.predict(X_val))**2 )
        error_by_alpha[i] += error

error_by_alpha /= N
```

We can then check `error_by_alpha` and choose the `alpha` corresponding to the lowest average error!

Datasets

The examples in this book use several datasets that are available either through `scikit-learn` or `seaboarn`. Those datasets are described briefly below.

Boston Housing

The Boston housing dataset contains information on 506 neighborhoods in Boston, Massachusetts. The target variable is the median value of owner-occupied homes (which appears to be censored at $50,000). This variable is approximately continuous, and so we will use this dataset for regression tasks. The predictors are all numeric and include details such as racial demographics and crime rates. It is available through `sklearn.datasets`.

Breast Cancer

The breast cancer dataset contains measurements of cells from 569 breast cancer patients. The target variable is whether the cancer is malignant or benign, so we will use it for binary classification tasks. The predictors are all quantitative and include information such as the perimeter or concavity of the measured cells. It is available through `sklearn.datasets`.

Penguins

The penguins dataset contains measurements from 344 penguins of three different species: *Adelie*, *Gentoo*, and *Chinstrap*. The target variable is the penguin's species. The predictors are both quantitative and categorical, and include information from the penguin's flipper size to the island on which it was found. Since this dataset includes categorical predictors, we will use it for tree-based models (though one could use it for quantitative models by creating dummy variables). It is available through `seaborn.load_dataset()`

Tips

The tips dataset contains 244 observations from a food server in 1990. The target variable is the amount of tips in dollars that the server received per meal. The predictors are both quantitative and categorical: the total bill, the size of the party, the day of the week, etc. Since the dataset includes categorical predictors and a quantitative target variable, we will use it for tree-based regression tasks. It is available through `seaborn.load_dataset()`.

Wine

The wine dataset contains data from chemical analysis on 178 wines of three classes. The target variable is the wine class, and so we will use it for classification tasks. The predictors are all numeric and detail each wine's chemical makeup. It is available through `sklearn.datasets`.