1
Linear Regression: An Introduction

This chapter introduces *least squares linear regression*, one of the simplest and most popular model in data science. Several of you may have seen it in high school. In particular, we focus on understanding linear regression in the context of machine learning. Using linear regression as an example, we will introduce the terminologies and ideas (some of them were mentioned in the Preface) that are widely applicable to more complicated models in ML.

1.1 A Warm-up Example

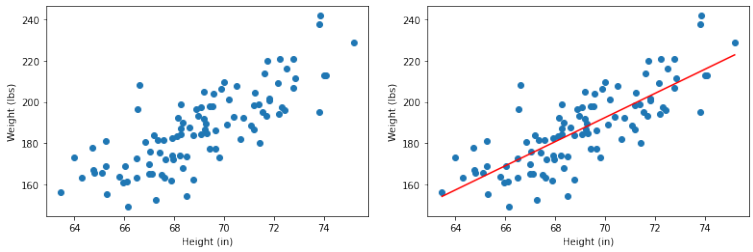

Figure 1.1: A dataset of heights and weights of some male adults. The figure on the right shows the least squares regression line that fits the data. Data from https://gist.github.com/nstokoe/7d4717e96c21b8ad04ec91f361b000cb

Suppose we have a dataset of heights and weights of some male individuals randomly chosen from the population. We wish to determine a relationship between heights and weights. One simple relationship would be a linear relationship; namely:

$$w = a_0 + a_1 h \tag{1.1}$$

where w is the weight, h is the height, and a_0, a_1 are constant coefficients. We can think of this as a *predictor* that maps height h to a predicted weight $a_0 + a_1 h$, and we want this value to be similar to the actual weight w. Obviously, a linear relationship won't describe the data exactly but we hope it is a reasonable fit to data. [1] In an ML setting, this relationship between h and w is called a *model* — a linear model to be more specific.

[1] Similar linear models are used in many disciplines. For instance, the famous Philips model in economics suggests a linear relationship between inflation rate and unemployment rate, at least when the inflation rate is high.

Based on the values of a_0 and a_1, there are infinitely many different choices of this linear model. Therefore, it is natural that we want to find the values of a_0, a_1 that yield the "best" model. In an ML setting, finding these optimal values of a_0, a_1 is known as *fitting* the model. One can posit different criteria for defining "goodness" of the model.

Here we use classic *least squares fit*, invented by Gauss. Given a dataset $\{(h_1, w_1), (h_2, w_2), \ldots, (h_n, w_n)\}$ of n pairs of heights and weights, the "goodness" of the model in (1.1) is

$$\frac{1}{n}\sum_{i=1}^{n}(w_i - a_0 - a_1 h_i)^2 \tag{1.2}$$

Notice that $w_i - a_0 - a_1 h_i$ is the difference between the actual weight w_i and the predicted weight $a_0 + a_1 h_i$. This difference is called the *residual* for the data point (h_i, w_i), and the full term in (1.2) is called the *average squared residuals*, or equivalently the *mean squared error* (MSE), of the dataset. The smaller the MSE, the closer the model's predictions are to actual weights, and the more accurate the model is. Therefore, the "best" model according to the least squares method would be the one defined by the values of a_0, a_1 that minimize (1.2). In an ML setting, a mathematical expression like (1.2) that captures the "badness" of the model is called a *loss function*. In general, we find the "best" model by minimizing the loss function.

Example 1.1.1. *If the data points (h, w) are given as $\{(65, 130), (58, 120), (73, 160)\}$, the least squares fit will find the values of a_0, a_1 that minimize*

$$\frac{1}{3}((130 - a_0 - 65a_1)^2 + (120 - a_0 - 58a_1)^2 + (160 - a_0 - 73a_1)^2)$$

which are $a_0 = -\frac{510}{13}$, $a_1 = \frac{35}{13}$.

Problem 1.1.2. *Between the two lines in Figure 1.2, which is more preferred by the least squares regression method?*

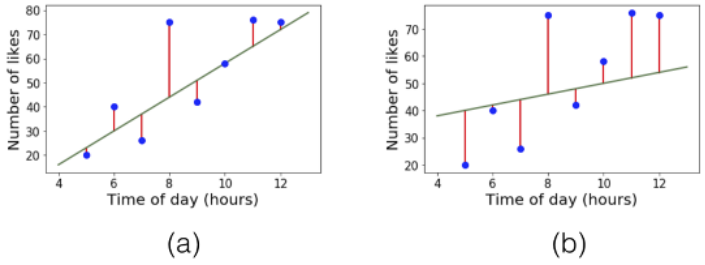

Figure 1.2: Two lines that describe the relationship of the same dataset.

Problem 1.1.3. *Using calculus, give an exact expression for a_0, a_1 that minimize (1.2). (Hint: (1.2) is quadratic in both a_0 and a_1. Fix the value of*

a_1 and minimize for a_0. Then minimize for a_1. Completing the square may be useful.) [2]

[2] A more general calculus based approach will be introduced in a later chapter.

1.1.1 Multivariate Linear Regression

One can generalize the above example to multi-variable settings. In general, we have k *predictor variables* and one *effect* variable. [3] The data points consist of $k+1$ coordinates, where the last coordinate is the value y of the effect variable and the first k coordinates contain values of the predictor variables $x_1, x_2, \ldots, x_k$. Then the relationship we are trying to fit has the form

[3] In the above example $k = 1$. The predictor variable was height and effect variable was weight.

$$y = a_0 + a_1 x_1 + a_2 x_2 + \cdots + a_k x_k \tag{1.3}$$

and the least squares fit method will find the values of $a_0, a_1, \cdots, a_k$ that minimize

$$\frac{1}{n} \sum_{i=1}^{n} (y^i - a_0 - a_1 x_1^i - a_2 x_2^i - \cdots - a_k x_k^i)^2 \tag{1.4}$$

where $(x_1^i, x_2^i, \ldots x_k^i, y^i)$ is the i-th data point.

We can simplify the notation by rewriting everything above in a vectorized notation. If we set $\vec{\mathbf{x}} = (1, x_1, x_2, \cdots, x_k)$ [4] and $\vec{\mathbf{a}} = (a_0, a_1, \cdots, a_k)$, then the relationship we are trying to fit has the form

[4] The 1 in the first coordinate is a dummy variable to naturally include the constant term into the vectorized notation.

$$y = \vec{\mathbf{a}} \cdot \vec{\mathbf{x}} \tag{1.5}$$

and the least squares fit method will find $\vec{\mathbf{a}} \in \mathbb{R}^{k+1}$ that minimize

$$\frac{1}{n} \sum_{i=1}^{n} (y^i - \vec{\mathbf{a}} \cdot \vec{\mathbf{x}}^i)^2 \tag{1.6}$$

where $(\vec{\mathbf{x}}^i, y^i)$ is the i-th data point. We discuss how to find the best values of $a_0, a_1, \ldots, a_k$ later in Chapter 3; for now just assume that the solution can be found.

1.1.2 Testing a model (Held-out Data)

A crucial step in machine learning is to *test* the trained/fitted model on *newly seen data*, or *held-out data*, that was not used during training. If we were to test the model in the above example, we would hold out a portion of the data points (say 20%) — *i.e.*, not use them during training — and check the average squared residual of the model on the held-out data points.

We can think of the average squared residual of the held-out data as an *estimate* of the average squared residual of the fitted model on the entire population of male adults. [5] The reason is that if the training data points were a random sample of the adult male population,

[5] If later in life you ever write up the results of a regression study, be sure to report the *RMSE error*, which is the square root of the average square residual on held-out data. Also report the R^2 value, which is closely related.

then so is the set of held-out data points. This is quite analogous to opinion polls, where the opinions of a few thousand randomly sampled individuals can be a reasonable estimate for the average opinion across the US. The math for such sampling estimates is covered in Chapter 18.

1.1.3 *More about Linear Regression*

In the above example, we used the least squares method, which uses the average squared residual to assess the model. The least squares fit is very common but other notions of fit may also be used. For instance, instead of taking the sum of squares of residuals, one could consider the sum of absolute values, or expressions using logarithms, etc. We see some of these examples in Chapter 4.

It is important to note that the relationship learnt via regression — and machine learning in general — is (a) approximate and (b) only holds for the population that the data was drawn from. Therefore, we cannot use the relationship to predict the output of a data that is not from the same distribution. Additionally, if the distribution of the data is shifted, the relationship no longer holds. We will discuss more about this in depth in Chapter 2.

1.2 *Using Linear Regression for Sentiment Prediction*

1.2.1 *Introduction*

While you might have seen linear regression as early as in high school, you probably did not see this cool application. In *sentiment classification*, we are given a piece of text and have to label it with $+1$ if it expresses positive sentiment and -1 otherwise.

Example 1.2.1. *Consider the following dataset, collected by showing snippets of text to humans and asking them to label them as positive ($+1$) or negative (-1)*

The film's performances are thrilling.	$+1$
It's not a great monster movie.	-1
It is definitely worth seeing.	$+1$
Unflinchingly bleak and desperate.	-1

Table 1.1: Data from Stanford Sentiment Treebank (SST). https://nlp.stanford.edu/sentiment/treebank.html

How can we train a model to label text snippets with the correct sentiment value, given a dataset of training examples? Here is an idea to try to solve it using a linear regression model. We first enumerate all English words and assign a real-valued score to each word, where the score for the i-th word is denoted by w_i. These scores will

be the *parameters* of the model. The output of the model, given a training example, is defined as $\sum_{j \in S} w_j$ where S is the multiset of indices of words in the text. [6] Then the least squares method needs to solve the following optimization problem for a dataset of (text, sentiment) pairs

$$\text{minimize} \sum_i \left(y^i - \sum_{j \in S^i} w_j \right)^2 \tag{1.7}$$

where S^i is the multiset of words in the i-th piece of text. Each of the values $\left(y^i - \sum_{j \in S^i} w_j \right)^2$ is called a *least squares error* or more generally the *loss* for that particular training example. The full summation is called a *training loss* of the dataset.

[6] Unlike in a set, an element can appear multiple times in a multiset. For example, if the word *good* appears twice in a text, then S contains two copies of *good*.

Example 1.2.2. *Assume we are training a sentiment prediction model on a dataset. Table 1.1 shows some of the model parameter values. Then the output of the model on the sentence "I like this movie" from the training data will be* $0.15 + 0.55 + 0.03 - 0.07 = 0.66$. *The output for "I dislike this movie" from the training data will be* $0.15 - 0.74 + 0.03 - 0.07 = -0.63$

i	word	w_i
1	I	0.15
2	like	0.55
3	dislike	−0.74
4	this	0.03
5	movie	−0.07
6	a	0

Table 1.2: Some of the parameter values of a sentiment prediction model.

We can also cast this in the standard formulation of linear regression as follows. The *bag of words (BoW)* representation of a piece of text is a vector in $\mathbb{R}^N$ where N is the number of dictionary words. The i-th coordinate is the number of times the i-th word appears in the piece of text. This represents the text as a very long vector, one coordinate per one English word in the dictionary. The vector usually contains a lot of zeros, since most words probably do not appear in this piece of text. If we denote the BoW vector as $\vec{\mathbf{x}}$, the output of the model is seen to be

$$\sum_{j \in S} w_j = \sum_i w_i x_i$$

which shows that the linear model we have proposed for sentiment prediction is just a subcase of linear regression (see (1.5)).

Example 1.2.3. *Consider the same model in Example 1.2.2. The BoW representation for the sentence "I like this movie" is* $(1, 1, 0, 1, 1, 0 \cdots)$. *The BoW representation for the sentence "I dislike this movie" is* $(1, 0, 1, 1, 1, 0 \cdots)$.

1.2.2 *Testing the Model*

Here we use the model from Example 1.2.2 to illustrate the training and testing process of a model. Assume that the following four sentences were a part of the training dataset.

I like this movie.	$+1$
I dislike this movie.	-1
I like this.	$+1$
I dislike this.	-1

Table 1.3: A portion of the training data for a sentiment prediction model.

Assuming that the model parameters are the same as reported in Table 1.2, we can calculate the training loss of the sentence "I like this movie" as $(+1 - 0.66)^2 \simeq 0.12$. Similarly, the squared residual for each of the four training sentences in Table 1.3 can be calculated as

I like this movie.	0.12
I dislike this movie.	0.14
I like this.	0.07
I dislike this.	0.19

Table 1.4: The squared residual for four training examples.

Now it is time to test the model. Assume that the sentence "I like a movie" is provided to the model as a test data. The *test loss* can be calculated in a way similar to the training loss as $(+1 - 0.63)^2 \simeq 0.14$. But to actually test if the model produces the correct sentiment label for this newly seen data, we now wish the model to output either $+1$ or -1, the only two labels that exist in the population. An easy fix is to change the output of the model at test time to be $\text{sign}(\sum_{j \in S} w_j)$. For this test data, the model will output $\text{sign}(0.63) = +1$.

On the Stanford Sentiment Treebank, this approach of training a least squares model yields a success rate of 78% [7]. By contrast, the state-of-the-art deep learning methods yield success rates exceeding 96%!

[7] To be more exact, this result is from a model called *ridge regression* model, which is linear regression model augmented by an ℓ_2 regularizer, which will be explained in Chapter 3

One thing to note is that while the training loss is calculated and explicitly used in the training process, the test loss is only a statistic that is generated after the training is over. It is a metric to assess if the model fitted on the training data also performs well for a more general data.

1.2.3 *Test Loss, Generalization, and Test accuracy*

As mentioned already, the goal of training a model is that it should make good predictions on new, previously-unseen data. Most models will exhibit a low training loss, but not all of them show a low test loss. This observation motivates the following definition:

$$\boxed{\text{Generalization Error} = |\text{training loss} - \text{test loss}|}$$

A trained model is said to *generalize well* if the generalization error is small. In our case, the loss is the average squared residual. Thus good generalization means that the average squared residual on test data points is similar to that on the training data.

Let us see what happens on our sentiment model when it is fitted and tested on the SST dataset.

Train MSE	0.0727
Test MSE	0.7523
Training accuracy	99.55%
Test accuracy	78.09%

Table 1.5: *Accuracy* refers to the classification accuracy when we make the model to output only ± 1 labels.

Example 1.2.4. *The **generalization error** above is the difference between MSE on test points and the MSE on training points, namely* $0.75 - 0.07 = 0.68$.

Let's try to understand the relationship between low test loss (the squared residual) and high test accuracy (for what fraction of test data points the sentiment was correct). Heuristically, the test loss (average squared residual) being 0.75 means that the the absolute value of the residual on a typical point is $\sqrt{0.75} \approx 0.87$. This means that for a data point with an actual positive sentiment (*i.e.*, label $+1$), the output of the model is roughly expected to lie in the interval $[1 - 0.87, 1 + 0.87]$, and similarly, for a data point with an actual negative sentiment (*i.e.*, label -1), the output of the model is roughly expected to lie in the interval $[-1 - 0.87, -1 + 0.87]$. Once we take the sign $\text{sign}(\sum_{j \in S} w_j)$ of the output of the model, the output is thus likely to be rounded off to the correct label. We also note that the training accuracy is almost 100%. This usually happens in settings where the number of parameters (*i.e.*, number of predictor variables) exceeds the number of training data points (or is close to it). The following problem explores this.

Problem 1.2.5. *An expert on TV claims to have a formula to predict the outcome of presidential elections. It uses* 31 *measurements of various economic and societal quantities (inflation, divorce rate, etc). The formula correctly predicts the winner of all elections* 1928-2020. *Should you believe the formula's prediction for the* 2024 *election? (Hint: Under fairly general conditions,* $T + 1$ *completely nonsense variables — i.e., having nothing to do with presidential politics — can be used to perfectly fit (via linear regression) the outcomes for* T *past presidential elections.* [8]*)*

[8] If a model does not generalize well, then it is said to *overfit* the training data.

1.2.4 Interpreting the Model

In many settings (*e.g.*, medicine), an important purpose of regression modeling is to understand the data or the phenomenon a bit

better. In this case, the phenomenon is "sentiment" and we are naturally curious about what positive or negative sentiment amounts to. Specifically, what caused the model's output to be $+1$ or -1 given a specific sentence?

Figure 1.3 shows a histogram of the values of w_i, the parameters of a sentiment prediction model that was trained on the Stanford Sentiment Treebank. Positive values of w_i imply that the words carry a positive sentiment, while negative values of w_i imply that the words carry a negative sentiment. Also, the greater the absolute value of w_i is, the stronger the sentiment. Notice that most words have a value of w_i close to zero, meaning the model views most words as neutral. The model "pays attention" to only a tiny set of words.

Words with high positive w_i values (*i.e.*, positive words) include *enjoyable*, *fun*, and *remarkable*. Words with high negative values (*i.e.*, negative words) include *suffers*, *dull*, and *worst*. Words with w_i values close to 0 (*i.e.*, neutral words) include *duty* and *desire*.

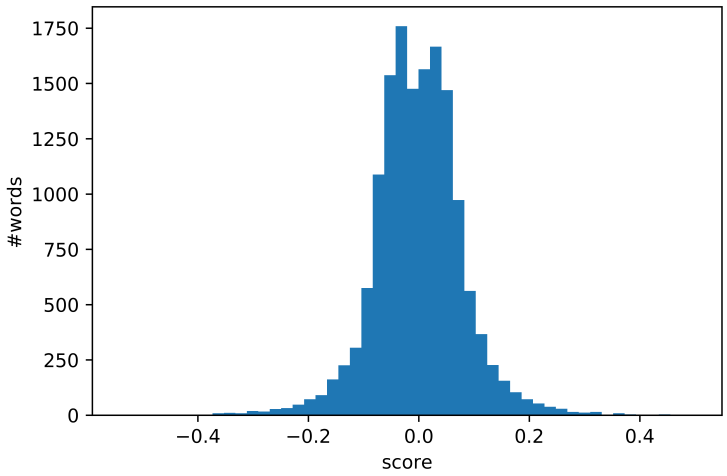

Figure 1.3: A histogram of the learned parameters w_i of a sentiment prediction model trained on the Stanford Sentiment Treebank.

1.3 *Importance of Featurization*

In the sentiment model, we chose a particular method to represent a piece of text with a vector. The coordinates of this vector are often referred to as *features* and this process of converting data into vectors is called *featurization*. One can conceive of other choices for featurizing text. For example, *bigram* featurization consists of the following: the coordinates of the vector correspond to *pairs* of words and the coordinate contains the number of times this pair of words appeared consecutively in the piece of text. In contrast, the choice of featurization from the earlier example matches each coordinate with a single word, and is called a *unigram* featuraization.

Bigram features allow the model to access information about phrases that were present in the text. For instance, in isolation "pretty" is a positive word and "bad" is a negative word. If they both occur in text one would imagine that they cancel each other out as far as overall sentiment is concerned. But the phrase "pretty bad" is more negative than "bad." Thus bigram features can improve the model's ability to capture sentiment.

The required number of dimension for bigram representations can get rather large. If the number of words is N, then the number of coordinates is N^2. Realize that the number of model parameters in linear regressions is the same as the number of coordinates. Thus if N is $30,000$ then the number of coordinates in bigram feature vector (and hence the number of model parameters) is close to a billion, which is a rather large number. In practice one might throw away information for all pairs except say the $10,000$ most common ones in the dataset. Usually models that incorporate bigram features do better than unigram-only models.

If one is trying to do studies of medical treatment with regression, there can be many potential featurizations of patient data. Doctors' annotations, test results, X-ray scans, etc. all have to be converted somehow into real-valued features, and the experimenter uses their prior knowledge and intuitions while featurizing the data.

Example 1.3.1. *Patients' raw data might include height and weight. If we use linear regression, the effect variable can only depend upon a linear combination of height and weight. But it is known that several health outcomes are better modeled using* Body Mass Index*, defined as* $weight/(height^2)$*. Thus the experimenter may include a separate coordinate for BMI, even though it duplicates information already present in the other coordinates.*

Example 1.3.2. *In Example 1.3.1, the weight in pounds may span a range of* $[90, 350]$*, whereas cholesterol ratio may span a range of* $[1, 10]$*. It is often a good idea to* normalize *the coordinate, which means to replace* x *with* $(x - \mu)/\sigma$ *where* μ *is the mean of the coordinate values in the dataset and* σ *is the standard deviation.*

Thus the same raw dataset can have multiple featurizations, with different number of coordinates. Problem 1.2.5 may make us wary of using featurizations with too many coordinates. We will learn a technique called *regularization* in Chapter 3, which helps mitigate the issue identified in Problem 1.2.5.

1.4 *Linear Regression in Python Programming*

In this section, we briefly discuss how to write the Python code to perform linear regression (*e.g.*, sentiment prediction). Python is often the language of choice for many machine learning applications due to its relative ease of use and the large variety of external packages available to automate the process. Here, we introduce a few of these packages:

- *numpy*: This package is ubiquitous throughout the machine learning community. It provides access to specialized array data structures which are implemented in highly optimized C code. Linear algebra computations and array restructuring operations are significantly faster with *numpy* compared to using Python directly. [9]
- *matplotlib*: This package enables Python programmers to create high quality plots and graphs. Visualizations are highly configurable and interoperable with several other Python packages. [10]
- *sklearn*: This package provides a potpourri of machine learning and data science models through an easy to use object-oriented API. In addition to linear regression, *sklearn* makes it possible to implement SVMs, clustering, neural networks, and much more; you will learn about a some of these models later in the course. [11]

[9] Documentation is available at https://numpy.org/

[10] Documentation is available at https://matplotlib.org/

[11] Documentation is available at https://scikit-learn.org/stable/index.html

Throughout this course, you will be asked to make use of functions defined in some of these external packages. You may not always be familiar with the usage of these functions. It is important to check the official documentation to learn about the usage and the signature of the functions.

The code snippet below uses the three aforementioned packages to perform linear regression on any given dataset.

```
# import necessary packages
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as mse
import matplotlib.pyplot as plt

# prepare train, test data
X = ...
y = ...
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# perform linear regression on train data
linreg = LinearRegression().fit(X_train, y_train)
pred_train = linreg.predict(X_train)
pred_test = linreg.predict(X_test)

# print train results
print('Train MSE: ', '{0:.4f}'.format(mse(y_train, pred_train)))
```

```
print('Test MSE: ', '{0:.4f}'.format(mse(y_test, pred_test)))
print('Train Acc: ', '{0:.2f}'.format(100*(np.sign(pred_train)==y_train).
                                      mean()))
print('Test Acc: ', '{0:.2f}'.format(100*(np.sign(pred_test)==y_test).mean()
                                      ))

# plot gold (actual) vs predicted value
plt.scatter(y_test, pred_test, c="red")
plt.xlabel("actual y value (y)")
plt.ylabel("predicted y value (y hat)")
plt.title("y vs y hat")
```

For readers who are not familiar with Python, we discuss some key details. In the first section of the code, we import the relevant packages

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as mse
import matplotlib.pyplot as plt
```

As seen in this example, there are two ways to load a package. The first option is to import the full package with the *import* keyword

```
import numpy as np
```

Notice that the we can assign the imported package a customized name with the *as* keyword. In this case, we decided refer to the package *numpy* with the name *np* throughout the rest of the code. This is indeed the case when we call

```
np.sign()
```

Here we refer to the method *sign()* of the *numpy* package with the customized name *np*. Alternatively, we can selectively import particular methods or classes with the *from* keyword

```
from sklearn.model_selection import train_test_split
```

The next part of the code is preparing the train, test data.

```
X = ...
y = ...
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

X will have to be an array of arrays, and y will have to be an array of values, with the same length as X. These arrays can be defined directly by specifying each of their entries, or they could be read from some external data (most commonly a csv file). Here, we present an example dataset where $\vec{\mathbf{x}} \in \mathbb{R}^2$:

```
X = [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9]]
y = [1, 1, 1, -1, -1]
```

Then we call the *train_test_split()* method to split the dataset into data for model training and testing. Alternatively, we can split the

dataset by manually slicing the data arrays. [12] In general, slicing a Python array involves the *:* operator along with start and end indices. For instance, consider an arbitrary array *a*. Then, the output of *a[i:j]* will be a subarray of *a* from the index *i* (inclusive) to the index *j* (exclusive). In the following code sample, we slice the data by specifying the number of training data points

[12] In Python, the term *slicing* refers to the process of creating a subarray of an array.

```
train_size = ...
X_train = X[:train_size]
X_test = X[train_size:]
y_train = y[:train_size]
y_test = y[train_size:]
```

Note that we have omitted some of the bounding indices. If the start index is omitted, Python assumes it to be 0 (so that the subarray is from the start of the array); for example, *X[:train_size]* is the first *train_size* entries of *X*. If the end index is omitted, Python assumes it to be *n*, the length of the array (so that the subarray ends at the end of the array); for instance, *X[train_size:]* is the remaining entries of *X*, once we remove the first *train_size* entries. Another way to slice the arrays is by specifying the number of test data points

```
test_size = ...
X_train = X[:-test_size]
X_test = X[-test_size:]
y_train = y[:-test_size]
y_test = y[-test_size:]
```

Here, notice that the index *-test_size* is a negative number. In this case, Python interprets this as *n - test_size*, where *n* is the size of the array. In other words, it is the index of the *test_size*-th element from the back of the array.

The third part of the code is fitting the linear regression model.

```
linreg = LinearRegression().fit(X_train, y_train)
pred_train = linreg.predict(X_train)
pred_test = linreg.predict(X_test)
```

The first line will generate the least squares fit model based on the train data. Then we can have the model make predictions on the train, test data.

Next, we print out the mean squared loss and the accuracy for the train, test data.

```
print('Train MSE: ', '{0:.4f}'.format(mse(y_train, pred_train)))
print('Test MSE: ', '{0:.4f}'.format(mse(y_test, pred_test)))
print('Train Acc: ', '{0:.2f}'.format(100*(np.sign(pred_train)==y_train).
                                      mean()))
print('Test Acc: ', '{0:.2f}'.format(100*(np.sign(pred_test)==y_test).mean()
                                      ))
```

Notice that we use the *mse()* method that we imported from the *sklearn* package. Also notice that when computing the accuracy, we

changed the output of the model to be the sign of the predicted values, so that we can compare them with the gold values. In many cases, there are packages that perform these elementary operations for machine learning.

Finally, we plot the actual and predicted values using the *matplotlib* package.

```
plt.scatter(y_test, pred_test, c="red")
plt.xlabel("actual y value (y)")
plt.ylabel("predicted y value (y hat)")
plt.title("y vs y hat")
```

The first line draws a scatter plot with the *y_test* in the x-axis and *pred_test* in the y-axis. Notice that you can specify the color of the data points by specifying the value of the parameter c. In general, *parameters* are optional values you can provide to Python functions. If the values to parameters are omitted, the functions will use their default values. The second and third lines specify the labels that will be written next to the axes. The final line specifies the title of the plot.