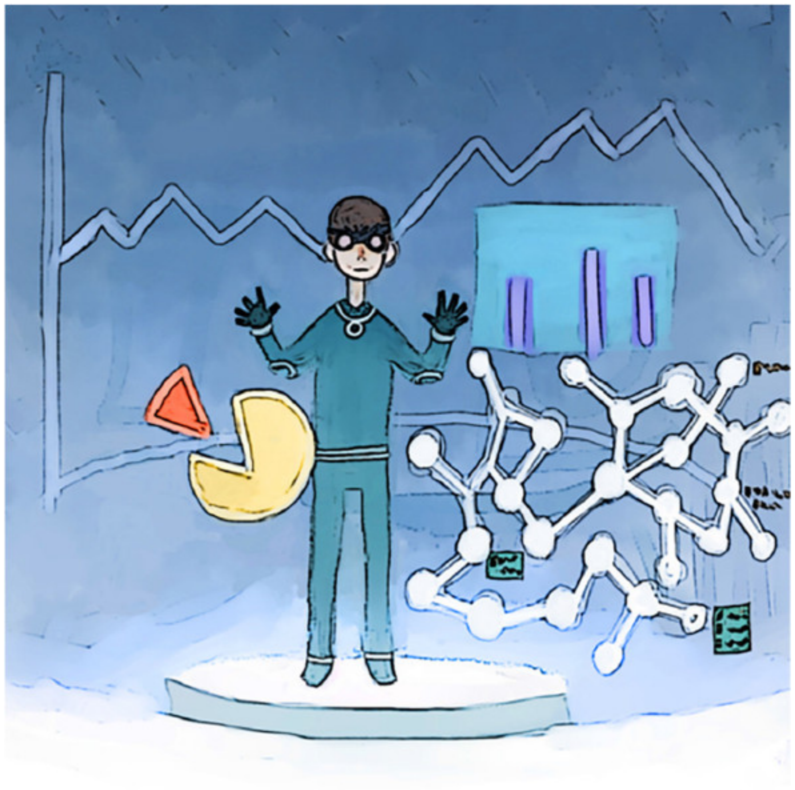
PREV
Part 2. Core learning algorithms

NEXT
Chapter 4. A gentle introduction to classification

Chapter 3. Linear regression and beyond



This chapter covers

- Fitting a line to data points
- Fitting arbitrary curves to data points
- Testing performance of regression algorithms
- Applying regression to real-world data

Remember science courses back in high school? It might have been a while ago, or who knows—maybe you're in high school now, starting your journey in machine learning early. Either way, whether you took biology, chemistry, or physics, a common technique to analyze data is to plot how changing one variable affects another.

Imagine plotting the correlation between rainfall frequency and agriculture production. You may observe that an increase in rainfall produces an increase in agriculture production rate. Fitting a line to these data points enables you to make predictions about the production rate under different rain conditions. If you discover the underlying function from a few data points, then that learned function empowers you to make predictions about the values of unseen data.

Regression is a study of how to best fit a curve to summarize your data. It's one of the most powerful and well-studied types of supervised-learning algorithms. In regression, we try to understand the data points by discovering the curve that might have generated them. In doing so, we seek an explanation for why the given data is scattered the way it is. The best-fit curve gives us a model for explaining how the dataset might have been produced.

This chapter shows you how to formulate a real-world problem to use regression. As you'll see, TensorFlow is just the right tool that delivers some of the most powerful predictors.

3.1. FORMAL NOTATION

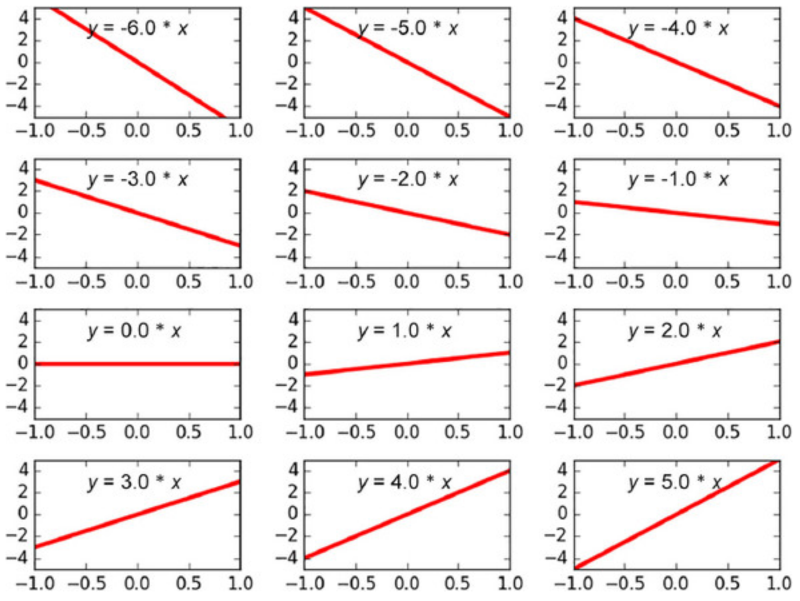
If you have a hammer, every problem looks like a nail. This chapter demonstrates the first major machine-learning tool, regression, and formally defines it by using precise mathematical symbols. Learning regression first is a great idea, because many of the skills you'll develop will carry over to other types of problems in future chapters. By the end of this chapter, regression will become the "hammer" in your box of machine-learning tools.

Let's say you have data about how much money people spent on bottles of beer. Alice spent \$4 on 2 bottles, Bob spent \$6 on 3 bottles, and Clair spent \$8 on 4 bottles. You want to find an equation that describes how the number of bottles affects the total cost. For example, if the linear equation $y = 2x$ describes the cost of buying a particular number of bottles, then you can find out how much each bottle of beer costs.

When a line appears to fit some data points well, you might claim that your linear model performs well. But you could have tried out many possible slopes instead of choosing the value 2. The choice of slope is the *parameter*, and the equation containing the parameter is the *model*. Speaking in machine-learning terms, the equation of the best-fit curve comes from learning the parameters of a model.

As another example, the equation $y = 3x$ is also a line, except with a steeper slope. You can replace that coefficient with any real number, let's call it w, and the equation will still produce a line: $y = wx$. Figure 3.1 shows how changing the parameter w affects the model. The set of all equations you can generate this way is denoted as $M = \{y = wx \mid w \in \mathbb{R}\}$. This is read, "All equations $y = wx$ such that w is a real number."

Figure 3.1. Different values of the parameter *w* result in different linear equations. The set of all these linear equations is what constitutes the linear model *M*.



M is a set of all possible models. Choosing a value for w generates a candidate model $M(w): y = wx$. The regression algorithms that you'll write in TensorFlow will iteratively converge to progressively better values for the model's parameter w. An optimal parameter, let's call it w^* (pronounced *w star*), is the best-fit equation $M(w^*)$: $y = w^*x$.

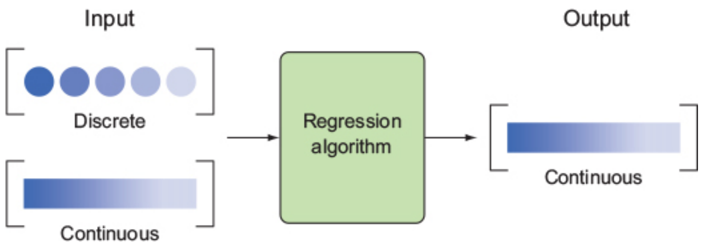
In the most general sense, a regression algorithm tries to design a function, let's call it *f*, that maps an input to an output. The function's domain is a real-valued vector $\mathbb{R}^d$, and its range is the set of real numbers $\mathbb{R}$.

Note

Regression can also be posed with multiple outputs, as opposed to just one real number. In that case, we call it *multivariate regression*.

The input of the function could be continuous or discrete. But the output must be continuous, as demonstrated in figure 3.2.

Figure 3.2. A regression algorithm is meant to produce continuous output. The input is allowed to be discrete or continuous. This distinction is important because discrete-valued outputs are handled better by classification, which is discussed in the next chapter.



Note

Regression predicts continuous outputs, but sometimes that's overkill. Sometimes we just want to predict a discrete output, such as 0 or 1, but nothing in between. Classification is a technique better suited for such tasks, and it's discussed in chapter 4.

We'd like to discover a function *f* that agrees well with the given data points, which are essentially input/output pairs. Unfortunately, the number of possible functions is infinite, so we'll have no luck trying them out one by one. Having too many options available to choose from is usually a bad idea. It behooves us to tighten the scope of all the functions we want to deal with. For example, if we look at only straight lines to fit a set of data points, the search becomes much easier.

Exercise 3.1

How many possible functions exist that map 10 integers to 10 integers? For example, let $f(x)$ be a function that can take numbers 0 through 9 and produce numbers 0 through 9. One example is the identity function that mimics its input—for example, $f(0) = 0$, $f(1) = 1$, and so on. How many other functions exist?

Answer

$10^{10} = 10,000,000,000$

3.1.1. How do you know the regression algorithm is working?

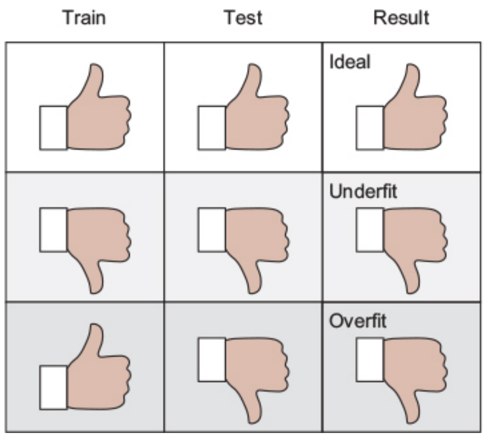
Let's say you're trying to sell a housing-market-predictor algorithm to a real estate firm. The algorithm predicts housing prices given properties such as the number of bedrooms and lot size. Real estate companies can easily make millions with such information, but they need some proof that the algorithm works before buying it from you.

To measure the success of the learning algorithm, you'll need to understand two important concepts, variance and bias:

- *Variance* indicates how sensitive a prediction is to the training set that was used. Ideally, how you choose the training set shouldn't matter—meaning a lower variance is desired.
- *Bias* indicates the strength of assumptions made about the training dataset. Making too many assumptions might make the model unable to generalize, so you should prefer low bias as well.

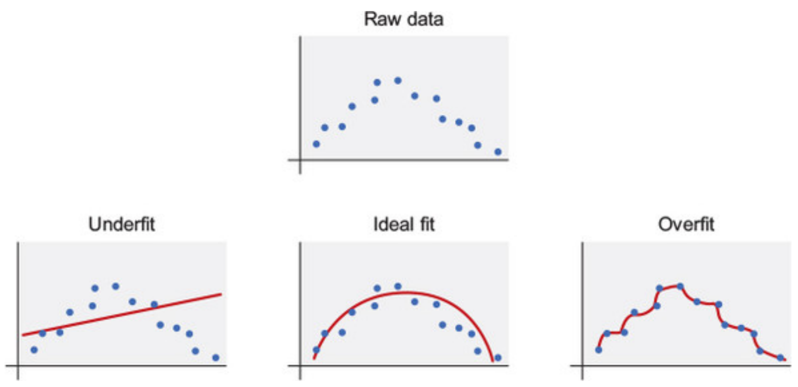
If a model is too flexible, it may accidentally memorize the training data instead of resolving useful patterns. You can imagine a curvy function passing through every point of a dataset, appearing to produce no error. If that happens, we say the learning algorithm *overfits* the data. In this case, the best-fit curve will agree with the training data well; but it may perform abysmally when evaluated on the testing data (see figure 3.3).

Figure 3.3. Ideally, the best-fit curve fits well on both the training data and the test data. If we witness it fitting poorly with the test data and the training data, there's a chance that our model is underfitting. On the other hand, if it performs poorly on the test data but well on the training data, we know the model is overfitting.



On the other end of the spectrum, a not-so-flexible model may generalize better to unseen testing data, but would score relatively low on the training data. That situation is called *underfitting*. A too-flexible model has high variance and low bias, whereas a too-strict model has low variance and high bias. Ideally, you want a model with both low-variance error and low-bias error. That way, it both generalizes to unseen data and captures the regularities of the data. See figure 3.4 for examples of a model underfitting and overfitting data points in 2D.

Figure 3.4. Examples of underfitting and overfitting the data



Concretely, the variance of a model is a measure of how badly the responses fluctuate, and the bias is a measure of how badly the response is offset from the ground-truth. You want your model to achieve accurate (low-bias) as well as reproducible (low-variance) results.

Exercise 3.2

Let's say your model is $M(w): y = wx$. How many possible functions can you generate if the values of the weight parameter w must be integers between 0 and 9 (inclusive)?

Answer

Only 10: $\{y = 0, y = x, y = 2x, \ldots, y = 9x\}$.

In summary, measuring how well your model does on the training data isn't a great indicator of its generalizability. Instead, you should evaluate your model on a separate batch of testing data. You might find out that your model performs great on the data you trained it with, but it performs terribly on the test data, in which case your model is likely overfitting the training data. If the testing error is around the same as the training error, and both errors are similar, then your model may be fitting well, or underfitting if that error is high.

This is why, to measure success in machine learning, you partition the dataset into two groups: a training dataset and a testing dataset. The model is learned using the training dataset, and performance is evaluated on the testing dataset (exactly how you evaluate performance is described in the next section). Out of the many possible weight parameters you can generate, the goal is to find one that best fits the data. The way you measure *best fit* is by defining a cost function, which is discussed in greater detail in the following section.

3.2. LINEAR REGRESSION

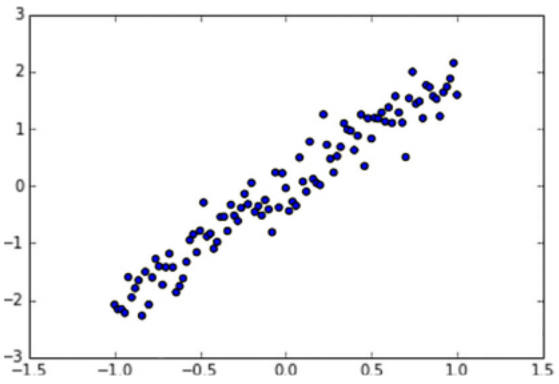
Let's start by creating fake data for a leap into the heart of linear regression. Create a Python source file called regression.py, and follow along with the following listing to initialize data. The code will produce output similar to figure 3.5.

Listing 3.1. Visualizing raw input

```
import numpy as np                                                       1
import matplotlib.pyplot as plt                                          2

x_train = np.linspace(-1, 1, 101)                                        3
y_train = 2 * x_train + np.random.randn(*x_train.shape) * 0.33           4

plt.scatter(x_train, y_train)                                            5
plt.show()                                                               5
```

- ***1*** **Imports NumPy to help generate initial raw data**
- ***2*** **Uses matplotlib to visualize the data**
- ***3*** **The input values are 101 evenly spaced numbers between –1 and 1.**
- ***4*** **The output values are proportional to the input but with added noise.**
- ***5*** **Uses matplotlib's function to generate a scatter plot of the data**

Figure 3.5. Scatter plot of `y = x + (noise)`



Now that you have some data points available, you can try fitting a line. At the very least, you need to provide TensorFlow with a score for each candidate parameter it tries. This score assignment is commonly called a *cost function*. The higher the cost, the worse the model parameter will be. For example, if the best-fit line is $y = 2x$, a parameter choice of 2.01 should have low cost, but the choice of –1 should have higher cost.

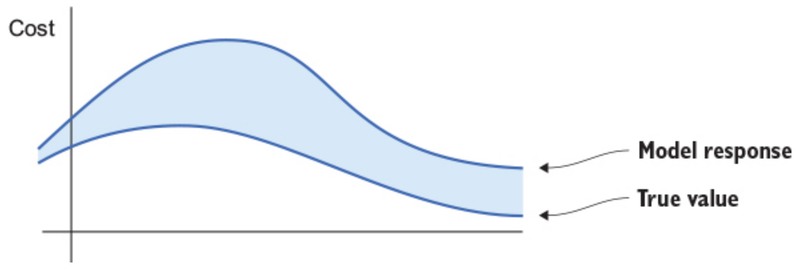
After you define the situation as a cost-minimization problem, as denoted in figure 3.6, TensorFlow takes care of the inner workings and tries to update the parameters in an efficient way to eventually reach the best possible value. Each step of looping through all your data to update the parameters is called an *epoch*.

Figure 3.6. Whichever parameter *w* minimizes, the cost is optimal. Cost is defined as the norm of the error between the ideal value with the model response. And, lastly, the response value is calculated from the function in the model set.

$$w* = \arg\ min_w\ cost(\underbrace{Y_{model}, Y_{ideal}}_{|\underbrace{Y_{model}}_{M(w, X)} - Y_{ideal}|})$$

In this example, the way you define *cost* is by the sum of errors. The error in predicting *x* is often calculated by the squared difference between the actual value *f*(*x*) and the predicted value *M*(*w*, *x*). Therefore, the cost is the sum of the squared differences between the actual and predicted values, as seen in figure 3.7.

Figure 3.7. The cost is the norm of the point-wise difference between the model response and the true value.



Update your previous code to look like the following listing. This code defines the cost function and asks TensorFlow to run an optimizer to find the optimal solution for the model parameters.

Listing 3.2. Solving linear regression

```
import tensorflow as tf                                                    1
import numpy as np                                                         1
import matplotlib.pyplot as plt                                            1

learning_rate = 0.01                                                       2
training_epochs = 100                                                      2

x_train = np.linspace(-1, 1, 101)                                          3
y_train = 2 * x_train + np.random.randn(*x_train.shape) * 0.33             3

X = tf.placeholder(tf.float32)                                             4
Y = tf.placeholder(tf.float32)                                             4

def model(X, w):                                                           5
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")                                       6

y_model = model(X, w)                                                      7
```

```
cost = tf.square(Y-y_model)                                                7

train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost) 8

sess = tf.Session()                                                        9
init = tf.global_variables_initializer()                                   9
sess.run(init)                                                             9

for epoch in range(training_epochs):                                       10
  for (x, y) in zip(x_train, y_train):                                     11
    sess.run(train_op, feed_dict={X: x, Y: y})                             12

w_val = sess.run(w)                                                        13

sess.close()                                                               14
plt.scatter(x_train, y_train)                                              15
y_learned = x_train*w_val                                                  16
plt.plot(x_train, y_learned, 'r')                                          16
plt.show(    )                                                             16
```

- **_1_ Imports TensorFlow for the learning algorithm. You'll need NumPy to set up the initial data. And you'll use matplotlib to visualize your data.**
- **_2_ Defines constants used by the learning algorithm. They're called hyperparameters.**
- **_3_ Sets up fake data that you'll use to find a best-fit line**
- **_4_ Sets up the input and output nodes as placeholders because the value will be injected by x_train and y_train**
- **_5_ Defines the model as y = w*X**
- **_6_ Sets up the weights variable**
- **_7_ Defines the cost function**
- **_8_ Defines the operation that will be called on each iteration of the learning algorithm**
- **_9_ Sets up a session and initializes all variables**
- **_10_ Loops through the dataset multiple times**
- **_11_ Loops through each item in the dataset**
- **_12_ Updates the model parameter(s) to try to minimize the cost function**
- **_13_ Obtains the final parameter value**
- **_14_ Closes the session**
- **_15_ Plots the original data**
- **_16_ Plots the best-fit line**

As figure 3.8 shows, you've just solved linear regression using TensorFlow! Conveniently, the rest of the topics in regression are just minor modifications of listing 3.2. The entire pipeline involves updating model parameters using TensorFlow, as summarized in figure 3.9.

Figure 3.8. Linear regression estimate shown by running listing 3.2

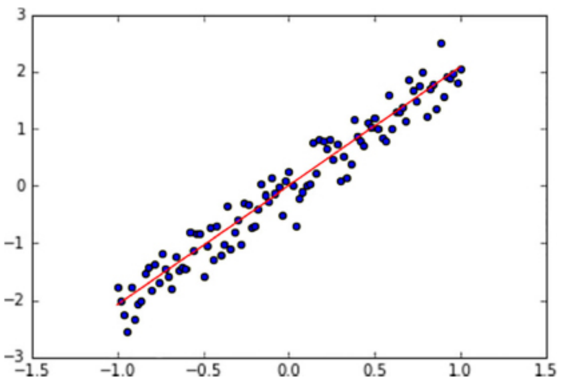
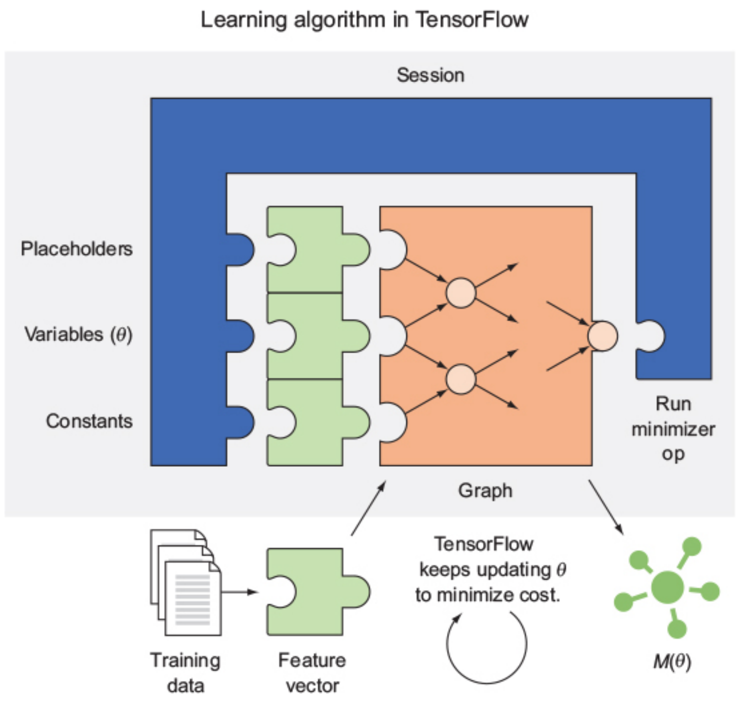


Figure 3.9. The learning algorithm updates the model's parameters to minimize the given cost function.



You've just learned how to implement a simple regression model in TensorFlow. Making further improvements is simply a matter of enhancing the model with the right medley of variance and bias, as we discussed earlier. For example, the linear regression model you've designed so far is burdened with a strong bias; it expresses only a limited set of functions, such as linear functions. In the next section, you'll try a more flexible model. You'll notice how only the TensorFlow graph needs to be rewired, while everything else (such as preprocessing, training, evaluation) stays the same.

3.3. POLYNOMIAL MODEL

Linear models may be an intuitive first guess, but rarely are real-world correlations so simple. For example, the trajectory of a missile through space is curved relative to the observer on Earth. Wi-Fi signal strength degrades with an inverse square law. The change in height of a flower over its lifetime certainly isn't linear.

When data points appear to form smooth curves rather than straight lines, you need to change your regression model from a straight line to something else. One such approach is to use a polynomial model. A *polynomial* is a generalization of a linear function. The *nth* degree polynomial looks like the following:

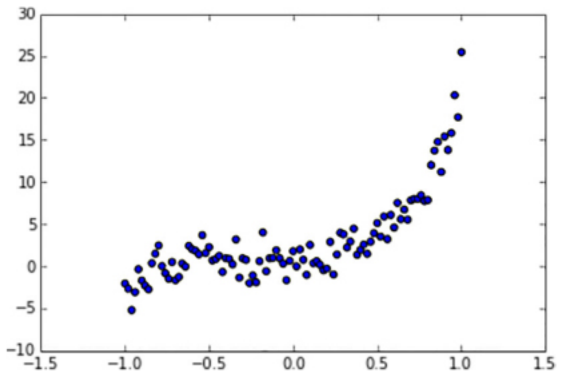
$$f(x) = w_n x^n + \ldots + w_1 x + w_0$$

Note

When $n = 1$, a polynomial is simply a linear equation $f(x) = w_1 x + w_0$.

Consider the scatter plot in figure 3.10, showing the input on the x-axis and the output on the y-axis. As you can tell, a straight line is insufficient to describe all the data. A polynomial function is a more flexible generalization of a linear function.

Figure 3.10. Data points like this aren't suitable for a linear model.



Let's try to fit a polynomial to this kind of data. Create a new file called polynomial.py, and follow along with the next listing.

Listing 3.3. Using a polynomial model

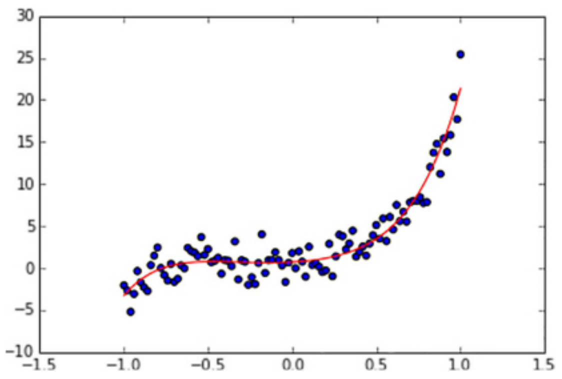
```
import tensorflow as tf                                              1
import numpy as np                                                   1
import matplotlib.pyplot as plt                                      1
                                                                     1
learning_rate = 0.01                                                 1
training_epochs = 40                                                 1

trX = np.linspace(-1, 1, 101)                                        2

num_coeffs = 6                                                       3
trY_coeffs = [1, 2, 3, 4, 5, 6]                                      3
trY = 0                                                              3
for i in range(num_coeffs):                                          3
    trY += trY_coeffs[i] * np.power(trX, i)                          3

trY += np.random.randn(*trX.shape) * 1.5                             4

plt.scatter(trX, trY)                                                5
plt.show()                                                           5

X = tf.placeholder(tf.float32)                                       6
Y = tf.placeholder(tf.float32)                                       6

def model(X, w):                                                     7
    terms = []                                                       7
    for i in range(num_coeffs):                                      7
        term = tf.multiply(w[i], tf.pow(X, i))                       7
        terms.append(term)                                           7
    return tf.add_n(terms)                                           7

w = tf.Variable([0.] * num_coeffs, name="parameters")                8
y_model = model(X, w)                                                8
```

```
cost = (tf.pow(Y-y_model, 2))                                                9
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)  9

sess = tf.Session()                                                         10
init = tf.global_variables_initializer()                                    10
sess.run(init)                                                              10
                                                                            10
for epoch in range(training_epochs):                                        10
    for (x, y) in zip(trX, trY):                                            10
        sess.run(train_op, feed_dict={X: x, Y: y})                          10
                                                                            10
w_val = sess.run(w)                                                         10
print(w_val)                                                                10

sess.close()                                                                11

plt.scatter(trX, trY)                                                       12
trY2 = 0                                                                    12
for i in range(num_coeffs):                                                 12
    trY2 += w_val[i] * np.power(trX, i)                                     12
                                                                            12
plt.plot(trX, trY2, 'r')                                                    12
plt.show()                                                                  12
```

- ***1*** **Imports the relevant libraries and initializes the hyperparameters**
- ***2*** **Sets up fake raw input data**
- ***3*** **Sets up raw output data based on a fifth-degree polynomial**
- ***4*** **Adds noise**
- ***5*** **Shows a scatter plot of the raw data**
- ***6*** **Defines the nodes to hold values for input/output pairs**
- ***7*** **Defines your polynomial model**
- ***8*** **Sets up the parameter vector to all zeros**
- ***9*** **Defines the cost function just as before**
- ***10*** **Sets up the session and runs the learning algorithm just as before**
- ***11*** **Closes the session when done**
- ***12*** **Plots the result**

The final output of this code is a fifth-degree polynomial that fits the data, as shown in figure 3.11.

Figure 3.11. The best-fit curve smoothly aligns with the nonlinear data.

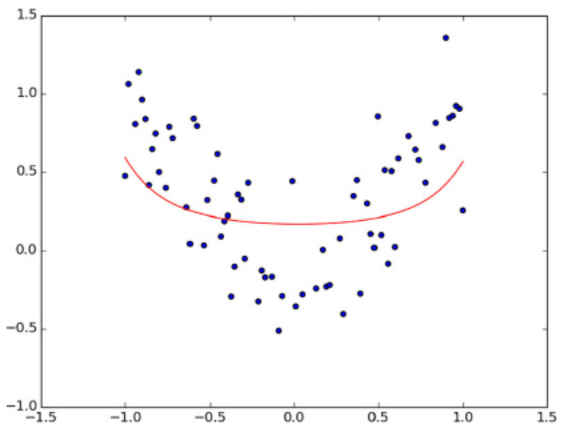


Don't be fooled by the wonderful flexibility of polynomials, as shown in the previous section. Just because higher-order polynomials are extensions of lower ones doesn't mean you should always prefer to use the more flexible model.

In the real world, raw data rarely forms a smooth curve mimicking a polynomial. Imagine you're plotting house prices over time. The data likely will contain fluctuations. The goal of regression is to represent the complexity in a simple mathematical equation. If your model is too flexible, the model may be overcomplicating its interpretation of the input.

Take, for example, the data presented in figure 3.12. You try to fit an eighth-degree polynomial into points that appear to follow the equation $y = x^2$. This process fails miserably as the algorithm tries its best to update the nine coefficients of the polynomial.

Figure 3.12. When the model is too flexible, a best-fit curve can look awkwardly complicated or unintuitive. We need to use regularization to improve the fit, so that the learned model performs well against test data.



Regularization is a technique to structure the parameters in a form you prefer, often to solve the problem of overfitting. In this case, you anticipate the learned coefficients to be 0 everywhere except for the second term, thus producing the curve $y = x^2$. The regression algorithm has no idea about this, so it may produce curves that score well but look strangely overcomplicated.

To influence the learning algorithm to produce a smaller coefficient vector (let's call it w), you add that penalty to the loss term. To control how significantly you want to weigh the penalty term, you multiply the penalty by a constant non-negative number, λ, as follows:

$$Cost(X, Y) = Loss(X, Y) + \lambda|\omega|$$

If λ is set to 0, regularization isn't in play. As you set λ to larger and larger values, parameters with larger norms will be heavily penalized. The choice of norm varies case by case, but parameters are typically measured by their L1 or L2 norm. Simply put, regularization reduces some of the flexibility of the otherwise easily tangled model.

To figure out which value of the regularization parameter λ performs best, you must split your dataset into two disjointed sets. About 70% of the randomly chosen input/output pairs will consist of the training dataset. The remaining 30% will be used for testing. You'll use the function provided in the following listing for splitting the dataset.

Listing 3.4. Splitting the dataset into testing and training sets

```
def split_dataset(x_dataset, y_dataset, ratio):                   1
    arr = np.arange(x_dataset.size)                                2
    np.random.shuffle(arr)                                         2
    num_train = int(ratio * x_dataset.size)                        3
    x_train = x_dataset[arr[0:num_train]]                          4
    x_test = x_dataset[arr[num_train:x_dataset.size]]              4
    y_train = y_dataset[arr[0:num_train]]                          5
    y_test = y_dataset[arr[num_train:x_dataset.size]]              5
    return x_train, x_test, y_train, y_test                        6
```

- ***1*** **Takes the input and output dataset as well as the desired split ratio**
- ***2*** **Shuffles a list of numbers**
- ***3*** **Calculates the number of training examples**
- ***4*** **Uses the shuffled list to split the x_dataset**
- ***5*** **Likewise, splits the y_dataset**
- ***6*** **Returns the split x and y datasets**

Exercise 3.3

A Python library called *scikit-learn* supports many useful data-preprocessing algorithms. You can call a function in scikit-learn to do exactly what listing 3.4 achieves. Can you find this function on the library's documentation? Hint: http://scikit-learn.org/stable/modules/classes.html#module-sklearn.model_selection (http://scikit-learn.org/stable/modules/classes.html#module-sklearn.model_selection).

Answer

It's called `sklearn.model_selection.train_test_split`.

With this handy tool, you can begin testing which value of λ performs best on your data. Open a new Python file, and follow along with this listing.

Listing 3.5. Evaluating regularization parameters

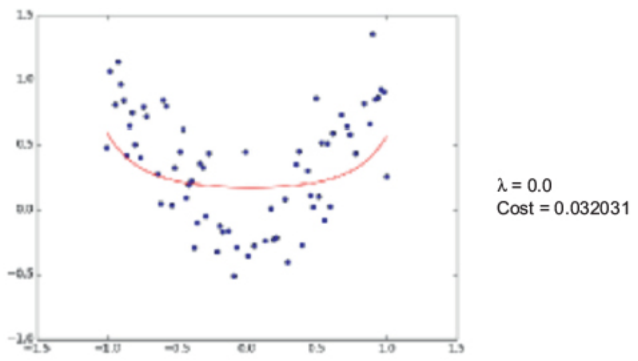
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

learning_rate = 0.001
training_epochs = 1000
reg_lambda = 0.

x_dataset = np.linspace(-1, 1, 100)

num_coeffs = 9
y_dataset_params = [0.] * num_coeffs
y_dataset_params[2] = 1
```

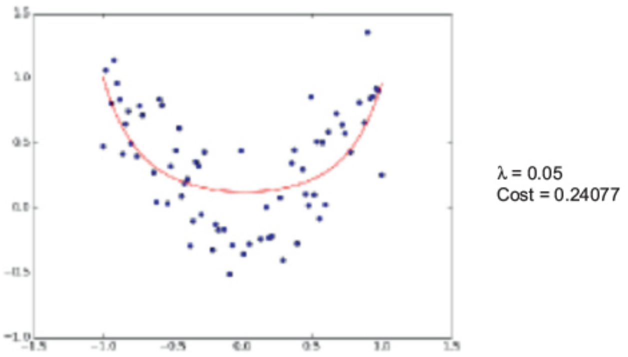
```
y_dataset = 0
for i in range(num_coeffs):
    y_dataset += y_dataset_params[i] * np.power(x_dataset, i)
y_dataset += np.random.randn(*x_dataset.shape) * 0.3

(x_train, x_test, y_train, y_test) = split_dataset(x_dataset, y_dataset, 0.7)

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

def model(X, w):
    terms = []
    for i in range(num_coeffs):
        term = tf.multiply(w[i], tf.pow(X, i))
        terms.append(term)
    return tf.add_n(terms)

w = tf.Variable([0.] * num_coeffs, name="parameters")
y_model = model(X, w)
cost = tf.div(tf.add(tf.reduce_sum(tf.square(Y-y_model)),
                     tf.multiply(reg_lambda, tf.reduce_sum(tf.square(w)))),
              2*x_train.size)
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

for reg_lambda in np.linspace(0,1,100):
    for epoch in range(training_epochs):
        sess.run(train_op, feed_dict={X: x_train, Y: y_train})
    final_cost = sess.run(cost, feed_dict={X: x_test, Y:y_test})
    print('reg lambda', reg_lambda)
    print('final cost', final_cost)

sess.close()
```

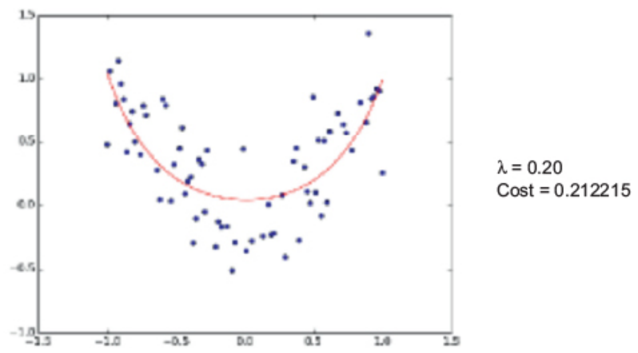
- **1 Imports the relevant libraries and initializes the hyperparameters**
- **2 Creates a fake dataset, $y = x^2$**
- **3 Splits the dataset into 70% training and 30% testing using listing 3.4**
- **4 Sets up the input/output placeholders**
- **5 Defines your model**
- **6 Defines the regularized cost function**
- **7 Sets up the session**
- **8 Tries various regularization parameters**
- **9 Closes the session**

If you plot the corresponding output per each regularization parameter from listing 3.5, you can see how the curve changes as λ increases. When λ is 0, the algorithm favors using the higher-order terms to fit the data. As you start penalizing parameters with a high L2 norm, the cost decreases, indicating that you're recovering from overfitting, as shown in figure 3.13.

Figure 3.13. As you increase the regularization parameter to some extent, the cost decreases. This implies that the model was originally overfitting the data, and regularization helped add structure.







3.5. APPLICATION OF LINEAR REGRESSION

Running linear regression on fake data is like buying a new car and never driving it. This awesome machinery begs to manifest itself in the real world! Fortunately, many datasets are available online to test your newfound knowledge of regression:

- The University of Massachusetts Amherst supplies small datasets of various types: www.umass.edu/statdata/statdata (http://www.umass.edu/statdata/statdata).
- Kaggle contains all types of large-scale data for machine-learning competitions: www.kaggle.com/datasets (http://www.kaggle.com/datasets).
- Data.gov is an open data initiative by the US government that contains many interesting and practical datasets: https://catalog.data.gov.

A good number of datasets contain dates. For example, there's a dataset of all phone calls to the 3-1-1 non-emergency line in Los Angeles, California. You can obtain it at http://mng.bz/6vHx (http://mng.bz/6vHx). A good feature to track could be the frequency of calls per day, week, or month. For convenience, the following listing allows you to obtain a weekly frequency count of data items.

Listing 3.6. Parsing raw CSV datasets

```
import csv                                                    1
import time                                                   2

def read(filename, date_idx, date_parse, year, bucket=7):

    days_in_year = 365

    freq = {}                                                 3
    for period in range(0, int(days_in_year / bucket)):
        freq[period] = 0

    with open(filename, 'rb') as csvfile:                     4
        csvreader = csv.reader(csvfile)
        csvreader.next()
        for row in csvreader:
            if row[date_idx] == '':
                continue
            t = time.strptime(row[date_idx], date_parse)
            if t.tm_year == year and t.tm_yday < (days_in_year-1):
                freq[int(t.tm_yday / bucket)] += 1

    return freq

freq = read('311.csv', 0, '%m/%d/%Y', 2014)                  5
```

- ***1* For easily reading CSV files**
- ***2* For using useful date functions**
- ***3* Sets up initial frequency map**
- ***4* Reads data and aggregates count per period**
- ***5* Obtains a weekly frequency count of 3-1-1 phone calls in 2014**

This code gives you the training data for linear regression. The `freq` variable is a dictionary that maps a period (such as a week) to a frequency count. A year has 52 weeks, so you'll have 52 data points, if you leave `bucket=7` as is.

Now that you have data points, you have exactly the input and output necessary to fit a regression model by using the techniques covered in this chapter. More practically, the learned model can be used to interpolate or extrapolate frequency counts.

3.6. SUMMARY

- Regression is a type of supervised machine learning for predicting continuous-valued output.
- By defining a set of models, you greatly reduce the search space of possible functions. Moreover, TensorFlow takes advantage of the differentiable property of the functions by running its efficient gradient-descent optimizers to learn the parameters.
- You can easily modify linear regression to learn polynomials or other more complicated curves.
- To avoid overfitting your data, you regularize the cost function by penalizing larger-valued parameters.

- If the output of the function isn't continuous, a classification algorithm should be used instead (see the next chapter).
- TensorFlow enables you to solve linear-regression machine-learning problems effectively and efficiently, and hence make useful predictions about important matters, such as agricultural production, heart conditions, housing prices, and more.