CHAPTER 15

Multiple Regression

I don't look at a problem and put variables in there that don't affect it.
—Bill Parcells

Although the VP is pretty impressed with your predictive model, she thinks you can do better. To that end, you've collected additional data: you know how many hours each of your users works each day, and whether they have a PhD. You'd like to use this additional data to improve your model.

Accordingly, you hypothesize a linear model with more independent variables:

$$\text{minutes} = \alpha + \beta_1 \text{friends} + \beta_2 \text{work hours} + \beta_3 \text{phd} + \varepsilon$$

Obviously, whether a user has a PhD is not a number—but, as we mentioned in Chapter 11, we can introduce a *dummy variable* that equals 1 for users with PhDs and 0 for users without, after which it's just as numeric as the other variables.

The Model

Recall that in Chapter 14 we fit a model of the form:

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

Now imagine that each input x_i is not a single number but rather a vector of k numbers, $x_{i1}, ..., x_{ik}$. The multiple regression model assumes that:

$$y_i = \alpha + \beta_1 x_{i1} + . . . + \beta_k x_{ik} + \varepsilon_i$$

In multiple regression the vector of parameters is usually called β. We'll want this to include the constant term as well, which we can achieve by adding a column of 1s to our data:

```
beta = [alpha, beta_1, ..., beta_k]
```

and:

```
x_i = [1, x_i1, ..., x_ik]
```

Then our model is just:

```
from scratch.linear_algebra import dot, Vector

def predict(x: Vector, beta: Vector) -> float:
    """assumes that the first element of x is 1"""
    return dot(x, beta)
```

In this particular case, our independent variable `x` will be a list of vectors, each of which looks like this:

```
[1,    # constant term
 49,   # number of friends
 4,    # work hours per day
 0]    # doesn't have PhD
```

Further Assumptions of the Least Squares Model

There are a couple of further assumptions that are required for this model (and our solution) to make sense.

The first is that the columns of x are *linearly independent*—that there's no way to write any one as a weighted sum of some of the others. If this assumption fails, it's impossible to estimate `beta`. To see this in an extreme case, imagine we had an extra field `num_acquaintances` in our data that for every user was exactly equal to `num_friends`.

Then, starting with any `beta`, if we add *any* amount to the `num_friends` coefficient and subtract that same amount from the `num_acquaintances` coefficient, the model's predictions will remain unchanged. This means that there's no way to find *the* coefficient for `num_friends`. (Usually violations of this assumption won't be so obvious.)

The second important assumption is that the columns of x are all uncorrelated with the errors ε. If this fails to be the case, our estimates of `beta` will be systematically wrong.

For instance, in Chapter 14, we built a model that predicted that each additional friend was associated with an extra 0.90 daily minutes on the site.

Imagine it's also the case that:

- People who work more hours spend less time on the site.

- People with more friends tend to work more hours.

That is, imagine that the "actual" model is:

$$\text{minutes} = \alpha + \beta_1 \text{friends} + \beta_2 \text{work hours} + \varepsilon$$

where β_2 is negative, and that work hours and friends are positively correlated. In that case, when we minimize the errors of the single-variable model:

$$\text{minutes} = \alpha + \beta_1 \text{friends} + \varepsilon$$

we will underestimate β_1.

Think about what would happen if we made predictions using the single-variable model with the "actual" value of β_1. (That is, the value that arises from minimizing the errors of what we called the "actual" model.) The predictions would tend to be way too large for users who work many hours and a little too large for users who work few hours, because $\beta_2 < 0$ and we "forgot" to include it. Because work hours is positively correlated with number of friends, this means the predictions tend to be way too large for users with many friends, and only slightly too large for users with few friends.

The result of this is that we can reduce the errors (in the single-variable model) by decreasing our estimate of β_1, which means that the error-minimizing β_1 is smaller than the "actual" value. That is, in this case the single-variable least squares solution is biased to underestimate β_1. And, in general, whenever the independent variables are correlated with the errors like this, our least squares solution will give us a biased estimate of β_1.

Fitting the Model

As we did in the simple linear model, we'll choose `beta` to minimize the sum of squared errors. Finding an exact solution is not simple to do by hand, which means we'll need to use gradient descent. Again we'll want to minimize the sum of the squared errors. The error function is almost identical to the one we used in Chapter 14, except that instead of expecting parameters `[alpha, beta]` it will take a vector of arbitrary length:

```python
from typing import List

def error(x: Vector, y: float, beta: Vector) -> float:
    return predict(x, beta) - y
```

```
def squared_error(x: Vector, y: float, beta: Vector) -> float:
    return error(x, y, beta) ** 2

x = [1, 2, 3]
y = 30
beta = [4, 4, 4]  # so prediction = 4 + 8 + 12 = 24

assert error(x, y, beta) == -6
assert squared_error(x, y, beta) == 36
```

If you know calculus, it's easy to compute the gradient:

```
def sqerror_gradient(x: Vector, y: float, beta: Vector) -> Vector:
    err = error(x, y, beta)
    return [2 * err * x_i for x_i in x]

assert sqerror_gradient(x, y, beta) == [-12, -24, -36]
```

Otherwise, you'll need to take my word for it.

At this point, we're ready to find the optimal `beta` using gradient descent. Let's first write out a `least_squares_fit` function that can work with any dataset:

```
import random
import tqdm
from scratch.linear_algebra import vector_mean
from scratch.gradient_descent import gradient_step


def least_squares_fit(xs: List[Vector],
                      ys: List[float],
                      learning_rate: float = 0.001,
                      num_steps: int = 1000,
                      batch_size: int = 1) -> Vector:
    """
    Find the beta that minimizes the sum of squared errors
    assuming the model y = dot(x, beta).
    """
    # Start with a random guess
    guess = [random.random() for _ in xs[0]]

    for _ in tqdm.trange(num_steps, desc="least squares fit"):
        for start in range(0, len(xs), batch_size):
            batch_xs = xs[start:start+batch_size]
            batch_ys = ys[start:start+batch_size]

            gradient = vector_mean([sqerror_gradient(x, y, guess)
                                    for x, y in zip(batch_xs, batch_ys)])
            guess = gradient_step(guess, gradient, -learning_rate)

    return guess
```

We can then apply that to our data:

```
from scratch.statistics import daily_minutes_good
from scratch.gradient_descent import gradient_step

random.seed(0)
# I used trial and error to choose num_iters and step_size.
# This will run for a while.
learning_rate = 0.001

beta = least_squares_fit(inputs, daily_minutes_good, learning_rate, 5000, 25)
assert 30.50 < beta[0] < 30.70  # constant
assert  0.96 < beta[1] <  1.00  # num friends
assert -1.89 < beta[2] < -1.85  # work hours per day
assert  0.91 < beta[3] <  0.94  # has PhD
```

In practice, you wouldn't estimate a linear regression using gradient descent; you'd get the exact coefficients using linear algebra techniques that are beyond the scope of this book. If you did so, you'd find the equation:

$$\text{minutes} = 30.58 + 0.972 \text{ friends} - 1.87 \text{ work hours} + 0.923 \text{ phd}$$

which is pretty close to what we found.

Interpreting the Model

You should think of the coefficients of the model as representing all-else-being-equal estimates of the impacts of each factor. All else being equal, each additional friend corresponds to an extra minute spent on the site each day. All else being equal, each additional hour in a user's workday corresponds to about two fewer minutes spent on the site each day. All else being equal, having a PhD is associated with spending an extra minute on the site each day.

What this doesn't (directly) tell us is anything about the interactions among the variables. It's possible that the effect of work hours is different for people with many friends than it is for people with few friends. This model doesn't capture that. One way to handle this case is to introduce a new variable that is the *product* of "friends" and "work hours." This effectively allows the "work hours" coefficient to increase (or decrease) as the number of friends increases.

Or it's possible that the more friends you have, the more time you spend on the site *up to a point*, after which further friends cause you to spend less time on the site. (Perhaps with too many friends the experience is just too overwhelming?) We could try to capture this in our model by adding another variable that's the *square* of the number of friends.

Once we start adding variables, we need to worry about whether their coefficients "matter." There are no limits to the numbers of products, logs, squares, and higher powers we could add.

Goodness of Fit

Again we can look at the R-squared:

```
from scratch.simple_linear_regression import total_sum_of_squares

def multiple_r_squared(xs: List[Vector], ys: Vector, beta: Vector) -> float:
    sum_of_squared_errors = sum(error(x, y, beta) ** 2
                                for x, y in zip(xs, ys))
    return 1.0 - sum_of_squared_errors / total_sum_of_squares(ys)
```

which has now increased to 0.68:

```
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta) < 0.68
```

Keep in mind, however, that adding new variables to a regression will *necessarily* increase the R-squared. After all, the simple regression model is just the special case of the multiple regression model where the coefficients on "work hours" and "PhD" both equal 0. The optimal multiple regression model will necessarily have an error at least as small as that one.

Because of this, in a multiple regression, we also need to look at the *standard errors* of the coefficients, which measure how certain we are about our estimates of each β_i. The regression as a whole may fit our data very well, but if some of the independent variables are correlated (or irrelevant), their coefficients might not *mean* much.

The typical approach to measuring these errors starts with another assumption—that the errors ε_i are independent normal random variables with mean 0 and some shared (unknown) standard deviation σ. In that case, we (or, more likely, our statistical software) can use some linear algebra to find the standard error of each coefficient. The larger it is, the less sure our model is about that coefficient. Unfortunately, we're not set up to do that kind of linear algebra from scratch.

Digression: The Bootstrap

Imagine that we have a sample of n data points, generated by some (unknown to us) distribution:

```
data = get_sample(num_points=n)
```

In Chapter 5, we wrote a function that could compute the `median` of the sample, which we can use as an estimate of the median of the distribution itself.

But how confident can we be about our estimate? If all the data points in the sample are very close to 100, then it seems likely that the actual median is close to 100. If approximately half the data points in the sample are close to 0 and the other half are close to 200, then we can't be nearly as certain about the median.

If we could repeatedly get new samples, we could compute the medians of many samples and look at the distribution of those medians. Often we can't. In that case we can *bootstrap* new datasets by choosing *n* data points *with replacement* from our data. And then we can compute the medians of those synthetic datasets:

```
from typing import TypeVar, Callable

X = TypeVar('X')        # Generic type for data
Stat = TypeVar('Stat')  # Generic type for "statistic"

def bootstrap_sample(data: List[X]) -> List[X]:
    """randomly samples len(data) elements with replacement"""
    return [random.choice(data) for _ in data]

def bootstrap_statistic(data: List[X],
                        stats_fn: Callable[[List[X]], Stat],
                        num_samples: int) -> List[Stat]:
    """evaluates stats_fn on num_samples bootstrap samples from data"""
    return [stats_fn(bootstrap_sample(data)) for _ in range(num_samples)]
```

For example, consider the two following datasets:

```
# 101 points all very close to 100
close_to_100 = [99.5 + random.random() for _ in range(101)]

# 101 points, 50 of them near 0, 50 of them near 200
far_from_100 = ([99.5 + random.random()] +
               [random.random() for _ in range(50)] +
               [200 + random.random() for _ in range(50)])
```

If you compute the `medians` of the two datasets, both will be very close to 100. However, if you look at:

```
from scratch.statistics import median, standard_deviation

medians_close = bootstrap_statistic(close_to_100, median, 100)
```

you will mostly see numbers really close to 100. But if you look at:

```
medians_far = bootstrap_statistic(far_from_100, median, 100)
```

you will see a lot of numbers close to 0 and a lot of numbers close to 200.

The `standard_deviation` of the first set of medians is close to 0, while that of the second set of medians is close to 100:

```
assert standard_deviation(medians_close) < 1
assert standard_deviation(medians_far) > 90
```

(This extreme a case would be pretty easy to figure out by manually inspecting the data, but in general that won't be true.)

Standard Errors of Regression Coefficients

We can take the same approach to estimating the standard errors of our regression coefficients. We repeatedly take a `bootstrap_sample` of our data and estimate `beta` based on that sample. If the coefficient corresponding to one of the independent variables (say, `num_friends`) doesn't vary much across samples, then we can be confident that our estimate is relatively tight. If the coefficient varies greatly across samples, then we can't be at all confident in our estimate.

The only subtlety is that, before sampling, we'll need to `zip` our `x` data and `y` data to make sure that corresponding values of the independent and dependent variables are sampled together. This means that `bootstrap_sample` will return a list of pairs `(x_i, y_i)`, which we'll need to reassemble into an `x_sample` and a `y_sample`:

```
from typing import Tuple

import datetime

def estimate_sample_beta(pairs: List[Tuple[Vector, float]]):
    x_sample = [x for x, _ in pairs]
    y_sample = [y for _, y in pairs]
    beta = least_squares_fit(x_sample, y_sample, learning_rate, 5000, 25)
    print("bootstrap sample", beta)
    return beta

random.seed(0) # so that you get the same results as me

# This will take a couple of minutes!
bootstrap_betas = bootstrap_statistic(list(zip(inputs, daily_minutes_good)),
                                      estimate_sample_beta,
                                      100)
```

After which we can estimate the standard deviation of each coefficient:

```
bootstrap_standard_errors = [
    standard_deviation([beta[i] for beta in bootstrap_betas])
    for i in range(4)]

print(bootstrap_standard_errors)

# [1.272,    # constant term, actual error = 1.19
#  0.103,    # num_friends,   actual error = 0.080
#  0.155,    # work_hours,    actual error = 0.127
#  1.249]    # phd,           actual error = 0.998
```

(We would likely get better estimates if we collected more than 100 samples and used more than 5,000 iterations to estimate each `beta`, but we don't have all day.)

We can use these to test hypotheses such as "does β_i equal 0?" Under the null hypothesis $\beta_i = 0$ (and with our other assumptions about the distribution of ε_i), the statistic:

$t_j = \widehat{\beta}_j / \widehat{\sigma}_j$

which is our estimate of β_j divided by our estimate of its standard error, follows a *Student's t-distribution* with "$n - k$ degrees of freedom."

If we had a `students_t_cdf` function, we could compute p-values for each least-squares coefficient to indicate how likely we would be to observe such a value if the actual coefficient were 0. Unfortunately, we don't have such a function. (Although we would if we weren't working from scratch.)

However, as the degrees of freedom get large, the t-distribution gets closer and closer to a standard normal. In a situation like this, where n is much larger than k, we can use `normal_cdf` and still feel good about ourselves:

```
from scratch.probability import normal_cdf

def p_value(beta_hat_j: float, sigma_hat_j: float) -> float:
    if beta_hat_j > 0:
        # if the coefficient is positive, we need to compute twice the
        # probability of seeing an even *larger* value
        return 2 * (1 - normal_cdf(beta_hat_j / sigma_hat_j))
    else:
        # otherwise twice the probability of seeing a *smaller* value
        return 2 * normal_cdf(beta_hat_j / sigma_hat_j)

assert p_value(30.58, 1.27)   < 0.001  # constant term
assert p_value(0.972, 0.103)  < 0.001  # num_friends
assert p_value(-1.865, 0.155) < 0.001  # work_hours
assert p_value(0.923, 1.249)  > 0.4    # phd
```

(In a situation *not* like this, we would probably be using statistical software that knows how to compute the t-distribution, as well as how to compute the exact standard errors.)

While most of the coefficients have very small p-values (suggesting that they are indeed nonzero), the coefficient for "PhD" is not "significantly" different from 0, which makes it likely that the coefficient for "PhD" is random rather than meaningful.

In more elaborate regression scenarios, you sometimes want to test more elaborate hypotheses about the data, such as "at least one of the β_j is nonzero" or "β_1 equals β_2 *and* β_3 equals β_4." You can do this with an *F-test*, but alas, that falls outside the scope of this book.

Regularization

In practice, you'd often like to apply linear regression to datasets with large numbers of variables. This creates a couple of extra wrinkles. First, the more variables you use, the more likely you are to overfit your model to the training set. And second, the more nonzero coefficients you have, the harder it is to make sense of them. If the goal is to *explain* some phenomenon, a sparse model with three factors might be more useful than a slightly better model with hundreds.

Regularization is an approach in which we add to the error term a penalty that gets larger as `beta` gets larger. We then minimize the combined error and penalty. The more importance we place on the penalty term, the more we discourage large coefficients.

For example, in *ridge regression*, we add a penalty proportional to the sum of the squares of the `beta_i` (except that typically we don't penalize `beta_0`, the constant term):

```python
# alpha is a *hyperparameter* controlling how harsh the penalty is.
# Sometimes it's called "lambda" but that already means something in Python.
def ridge_penalty(beta: Vector, alpha: float) -> float:
    return alpha * dot(beta[1:], beta[1:])

def squared_error_ridge(x: Vector,
                        y: float,
                        beta: Vector,
                        alpha: float) -> float:
    """estimate error plus ridge penalty on beta"""
    return error(x, y, beta) ** 2 + ridge_penalty(beta, alpha)
```

We can then plug this into gradient descent in the usual way:

```python
from scratch.linear_algebra import add

def ridge_penalty_gradient(beta: Vector, alpha: float) -> Vector:
    """gradient of just the ridge penalty"""
    return [0.] + [2 * alpha * beta_j for beta_j in beta[1:]]

def sqerror_ridge_gradient(x: Vector,
                           y: float,
                           beta: Vector,
                           alpha: float) -> Vector:
    """
    the gradient corresponding to the ith squared error term
    including the ridge penalty
    """
    return add(sqerror_gradient(x, y, beta),
               ridge_penalty_gradient(beta, alpha))
```

And then we just need to modify the `least_squares_fit` function to use the `sqerror_ridge_gradient` instead of `sqerror_gradient`. (I'm not going to repeat the code here.)

With `alpha` set to 0, there's no penalty at all and we get the same results as before:

```
random.seed(0)
beta_0 = least_squares_fit_ridge(inputs, daily_minutes_good, 0.0,  # alpha
                                 learning_rate, 5000, 25)
# [30.51, 0.97, -1.85, 0.91]
assert 5 < dot(beta_0[1:], beta_0[1:]) < 6
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_0) < 0.69
```

As we increase `alpha`, the goodness of fit gets worse, but the size of `beta` gets smaller:

```
beta_0_1 = least_squares_fit_ridge(inputs, daily_minutes_good, 0.1,  # alpha
                                   learning_rate, 5000, 25)
# [30.8, 0.95, -1.83, 0.54]
assert 4 < dot(beta_0_1[1:], beta_0_1[1:]) < 5
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_0_1) < 0.69

beta_1 = least_squares_fit_ridge(inputs, daily_minutes_good, 1,  # alpha
                                 learning_rate, 5000, 25)
# [30.6, 0.90, -1.68, 0.10]
assert 3 < dot(beta_1[1:], beta_1[1:]) < 4
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_1) < 0.69

beta_10 = least_squares_fit_ridge(inputs, daily_minutes_good,10,  # alpha
                                  learning_rate, 5000, 25)
# [28.3, 0.67, -0.90, -0.01]
assert 1 < dot(beta_10[1:], beta_10[1:]) < 2
assert 0.5 < multiple_r_squared(inputs, daily_minutes_good, beta_10) < 0.6
```

In particular, the coefficient on "PhD" vanishes as we increase the penalty, which accords with our previous result that it wasn't significantly different from 0.

Usually you'd want to `rescale` your data before using this approach. After all, if you changed years of experience to centuries of experience, its least squares coefficient would increase by a factor of 100 and suddenly get penalized much more, even though it's the same model.

Another approach is *lasso regression*, which uses the penalty:

```
def lasso_penalty(beta, alpha):
    return alpha * sum(abs(beta_i) for beta_i in beta[1:])
```

Whereas the ridge penalty shrank the coefficients overall, the lasso penalty tends to force coefficients to be 0, which makes it good for learning sparse models. Unfortunately, it's not amenable to gradient descent, which means that we won't be able to solve it from scratch.

For Further Exploration

- Regression has a rich and expansive theory behind it. This is another place where you should consider reading a textbook, or at least a lot of Wikipedia articles.
- scikit-learn has a `linear_model` module that provides a `LinearRegression` model similar to ours, as well as ridge regression, lasso regression, and other types of regularization.
- Statsmodels is another Python module that contains (among other things) linear regression models.