Notes for ISL Chapter 3: Linear Regression

Justin Burruss

2025-08-06

Background

The Python code and notes below are for the ISL study group. This is to try out some of the concepts from Chapter 3 on the `Advertising` data set. The document was created in RMarkdown with the Python code running via the `reticulate` library plus a little LaTeX.

Our ground rule: when implementing concepts from the chapter, just use basic Python + Pandas + NumPy. It's OK to use more when visualizing or evaluating results.

Loading the data set

The Pandas library ("Panda" as in "**Pan**el **da**ta") makes it easy to load the CSV and offers some functionality similar to `data.frame` or `data.table` in R. First we load our data from the CSV.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import timeit

# read CSV

c = pd.read_csv(r"E:\docs\Classes\ISL\Advertising.csv") \
    .rename(columns={"Unnamed: 0": "Id"}) \
    .set_index('Id')
```

Simple linear regression

In Chapter 3 we start with a linear regression using independent variable `TV` and dependent variable `sales`. The most direct way to do this in NumPy is as a least squares problem of the form $A\mathbf{x} = \mathbf{b}$.

Least squares with NumPy

Our target $\mathbf{b}$ is just our sales vector, which we get as a NumPy array using `.values`. Our matrix A needs to have two columns since we have our intercept term along with the slope term. NumPy makes this easy with `column_stack` and `np.ones`.

```
b = c['sales'].values
A = np.column_stack((c['TV'].values, np.ones(len(c['TV'].values))))
```

With our vector $\mathbf{b}$ and matrix A all set up, it's now just a one-liner to get the least squares fit.

```
x, RSS, rank, S = np.linalg.lstsq(A, b, rcond=None)
print("Expected coeffs of 0.0475 for Beta_1 and 7.03 for Beta_0, found",
      x.round(4))
```

```
## Expected coeffs of 0.0475 for Beta_1 and 7.03 for Beta_0, found [0.0475 7.0326]
```

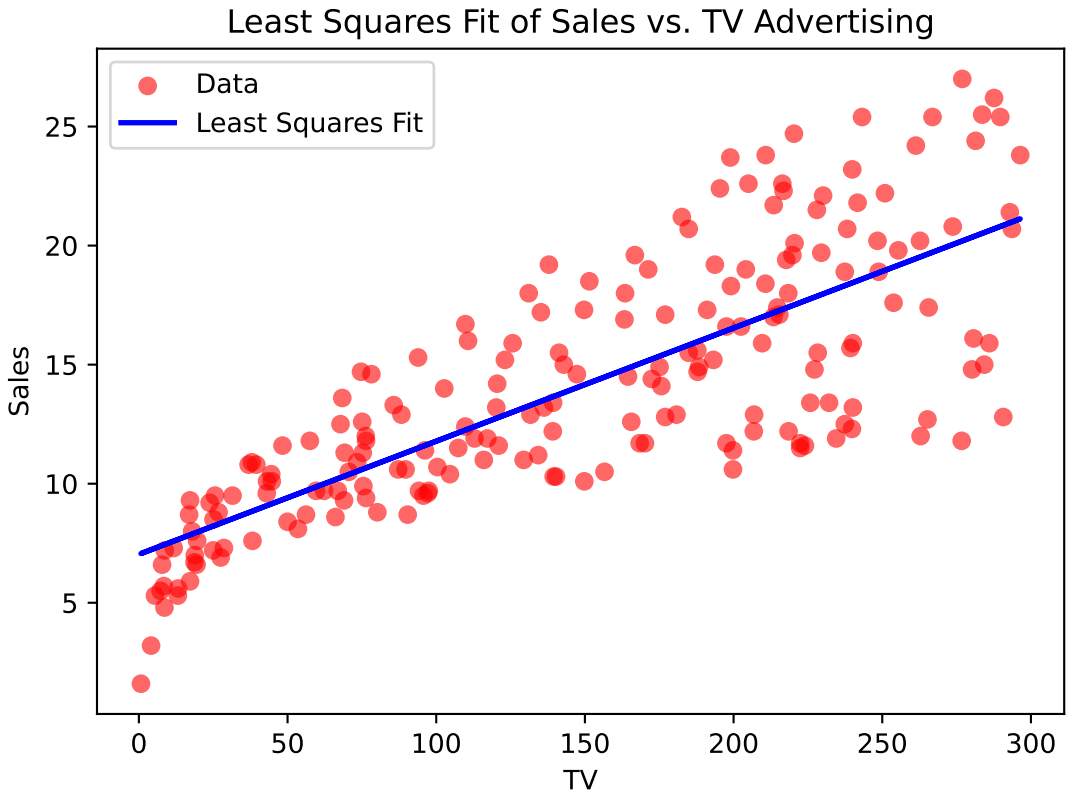
Let's check that we find the same RSS as `np.linalg.lstsq` within reasonable rounding & precision differences. Our fit is $A\mathbf{x}$, which in Python we express as `A@x`. The at symbol `@` is like `*` for mATrices. To calculate RSS we sum the square of the differences between $A\mathbf{x}$ and $\mathbf{b}$.

```
print("RSS from lstsq is", RSS[0].round(7),
      "versus manual RSS calc", np.sum((A@x - b)**2).round(7))
```

```
## RSS from lstsq is 2102.5305831 versus manual RSS calc 2102.5305831
```

That's really all there is to it with NumPy. We can visualize the fit $A\mathbf{x}$ versus actual $\mathbf{b}$ as was done in Figure 3.1 in Chapter 3.

```
plt.figure(1)
plt.scatter(c['TV'], b, color='red', label='Data', alpha=0.6)
plt.plot(c['TV'], A@x, color='blue', label='Least Squares Fit', linewidth=2)
plt.title('Least Squares Fit of Sales vs. TV Advertising')
plt.xlabel('TV')
plt.ylabel('Sales')
plt.legend()
plt.show()
```



Minimizing RSS

Taking a step back, in the chapter we're given in equation 3.3 that residual sum of squares (RSS) is defined as

$$RSS = (y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_1)^2 + (y_2 - \hat{\beta}_0 - \hat{\beta}_1 x_2)^2 + ... + (y_n - \hat{\beta}_0 - \hat{\beta}_1 x_n)^2$$

then are given the equation 3.4 minimizers and told that they follow "using some calculus". Let's step through and verify. Rather than mixing in our $A\mathbf{x} = \mathbf{b}$ notation we'll stick with the same notation as the book (see table below).

Table 1: Notation

Notation	Description
y	our dependent variable sales
$\hat{y}$	our estimate for the dependent variable sales
x	will be for our independent variable TV
$\hat{\beta}_0$	is our estimate for the intercept
$\hat{\beta}_1$	is our estimate for the slope, i.e., for $\hat{y} = \hat{\beta}_1 x + \hat{\beta}_0$

First we recognize that RSS is a function of $\hat{\beta}_0$ and $\hat{\beta}_1$.

$$\begin{aligned} RSS(\hat{\beta}_0, \hat{\beta}_1) &= (y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_1)^2 + (y_2 - \hat{\beta}_0 - \hat{\beta}_1 x_2)^2 + ... + (y_n - \hat{\beta}_0 - \hat{\beta}_1 x_n)^2 \\ &= \sum_{i=1}^{n} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2 \end{aligned}$$

We can recognize that this is some quadratic function where the slope at the minimum error is zero, so if we want to minimize the error we ought to solve for the partial derivatives being zero. We'll use the notation $RSS_{\hat{\beta}_0}$ to indicate the partial derivative of the RSS function with respect to $\hat{\beta}_0$.

$$\begin{aligned} RSS_{\hat{\beta}_0} &= \sum_{i=1}^{n}(2)(-1)(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i) && \text{chain rule} \\ &= -2\sum_{i=1}^{n}(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i) && \text{pull out the constant} \end{aligned}$$

Then we set to zero and step through the algebra to get $\hat{\beta}_0$ on one side of the equation.

$$\begin{aligned} 0 &= -2\sum_{i=1}^{n}(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i) \\ &= \sum_{i=1}^{n}(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i) && \text{multiply by } -\tfrac{1}{2} \\ &= \sum_{i=1}^{n}(y_i) - \sum_{i=1}^{n}(\hat{\beta}_0) - \sum_{i=1}^{n}(\hat{\beta}_1 x_i) && \text{summation is linear} \\ \sum_{i=1}^{n}(\hat{\beta}_0) &= \sum_{i=1}^{n}(y_i) - \sum_{i=1}^{n}(\hat{\beta}_1 x_i) && \text{move term to other side} \\ n\hat{\beta}_0 &= \sum_{i=1}^{n}(y_i) - \sum_{i=1}^{n}(\hat{\beta}_1 x_i) && \text{sum over constant} \\ n\hat{\beta}_0 &= \sum_{i=1}^{n}(y_i) - \hat{\beta}_1 \sum_{i=1}^{n}(x_i) && \text{pull out constant} \\ \hat{\beta}_0 &= \frac{1}{n}\sum_{i=1}^{n}(y_i) - \hat{\beta}_1 \frac{1}{n}\sum_{i=1}^{n}(x_i) && \text{multiply by } \tfrac{1}{n} \end{aligned}$$

We recognize that $\frac{1}{n}\sum_{i=1}^{n}(y_i) = \bar{y}$ and $\frac{1}{n}\sum_{i=1}^{n}(x_i) = \bar{y}$ by definition, so we end up with

$$\begin{aligned} \hat{\beta}_0 &= \bar{y} - \hat{\beta}_1 \frac{1}{n}\sum_{i=1}^{n}(x_i) && \text{definition of } \bar{y} \\ \hat{\beta}_0 &= \bar{y} - \hat{\beta}_1 \bar{x} && \text{definition of } \bar{x} \end{aligned}$$

We'll use the notation $RSS_{\hat{\beta}_1}$ to indicate the partial derivative of the RSS function with respect to $\hat{\beta}_1$.

$$
\begin{aligned}
RSS_{\hat{\beta}_1} &= \sum_{i=1}^{n}(2)(-x_i)(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i) && \text{chain rule} \\
&= -2\sum_{i=1}^{n}(x_i)(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i) && \text{pull out the constant} \\
&= -2\sum_{i=1}^{n}(x_i y_i - x_i\hat{\beta}_0 - x_i\hat{\beta}_1 x_i) && \text{distribute the } x_i \\
&= -2\sum_{i=1}^{n}(x_i y_i - x_i\hat{\beta}_0 - {x_i}^2\hat{\beta}_1) && x_i \text{ times } x_i
\end{aligned}
$$

Now we set to zero and step through the necessary algebra to get $\hat{\beta}_1$ on one side of the equation. One step is to substitute in $\bar{y} - \hat{\beta}_1\bar{x}$ for $\hat{\beta}_0$ per what we found above.

$$
\begin{aligned}
0 &= -2\sum_{i=1}^{n}(x_i y_i - x_i\hat{\beta}_0 - {x_i}^2\hat{\beta}_1) \\
&= \sum_{i=1}^{n}(x_i y_i - x_i\hat{\beta}_0 - {x_i}^2\hat{\beta}_1) && \text{multiply by } -\tfrac{1}{2} \\
\sum_{i=1}^{n}(x_i\hat{\beta}_0) + \sum_{i=1}^{n}({x_i}^2\hat{\beta}_1) &= \sum_{i=1}^{n}(x_i y_i) && \text{move terms other side} \\
\hat{\beta}_0\sum_{i=1}^{n}(x_i) + \hat{\beta}_1\sum_{i=1}^{n}({x_i}^2) &= \sum_{i=1}^{n}(x_i y_i) && \text{pull out constants } \hat{\beta}_0 \text{ and } \hat{\beta}_1 \\
(\bar{y} - \hat{\beta}_1\bar{x})\sum_{i=1}^{n}(x_i) + \hat{\beta}_1\sum_{i=1}^{n}({x_i}^2) &= \sum_{i=1}^{n}(x_i y_i) && \text{substitute for } \hat{\beta}_0 \\
\bar{y}\sum_{i=1}^{n}(x_i) - \hat{\beta}_1\bar{x}\sum_{i=1}^{n}(x_i) + \hat{\beta}_1\sum_{i=1}^{n}({x_i}^2) &= \sum_{i=1}^{n}(x_i y_i) && \text{summation is linear} \\
-\hat{\beta}_1\bar{x}\sum_{i=1}^{n}(x_i) + \hat{\beta}_1\sum_{i=1}^{n}({x_i}^2) &= \sum_{i=1}^{n}(x_i y_i) - \bar{y}\sum_{i=1}^{n}(x_i) && \text{move term to other side} \\
\hat{\beta}_1[-\bar{x}\sum_{i=1}^{n}(x_i) + \sum_{i=1}^{n}({x_i}^2)] &= \sum_{i=1}^{n}(x_i y_i) - \bar{y}\sum_{i=1}^{n}(x_i) && \text{factor out } \hat{\beta}_1 \\
\hat{\beta}_1[\sum_{i=1}^{n}({x_i}^2) - \bar{x}\sum_{i=1}^{n}(x_i)] &= \sum_{i=1}^{n}(x_i y_i) - \bar{y}\sum_{i=1}^{n}(x_i) && \text{rearrange} \\
\hat{\beta}_1 &= \frac{\sum_{i=1}^{n}(x_i y_i) - \bar{y}\sum_{i=1}^{n}(x_i)}{\sum_{i=1}^{n}({x_i}^2) - \bar{x}\sum_{i=1}^{n}(x_i)} && \text{divide}
\end{aligned}
$$

This is not the final form given in the book, but it should give the same answer. Let's check.

```
print("Expected Beta_1 of 0.0475 vs manual calc of",
      ((np.sum(c['TV']*b)-b.mean()*np.sum(c['TV'])) /
       (np.sum(c['TV']**2)-c['TV'].mean()*np.sum(c['TV']))).round(4))
```

```
## Expected Beta_1 of 0.0475 vs manual calc of 0.0475
```

```
print("Expected Beta_0 of 7.03 vs manual calc of",
      (b.mean() - c['TV'].mean() *
       ((np.sum(c['TV']*b)-b.mean()*np.sum(c['TV'])) /
        (np.sum(c['TV']**2)-c['TV'].mean()*np.sum(c['TV'])))).round(4))
```

```
## Expected Beta_0 of 7.03 vs manual calc of 7.0326
```

For completeness we can keep going to see if we can get $\hat{\beta}_1$ defined under the same equation we were given in the book. First, let's visit the denominator. The book gives as the denominator $\sum_{i=1}^{n}[(x_i - \bar{x})^2]$, in other words n times $Var(X)$. We can get there with some algebra.

$$
\begin{aligned}
\sum_{i=1}^{n}({x_i}^2) - \bar{x}\sum_{i=1}^{n}(x_i) &= \sum_{i=1}^{n}({x_i}^2) - \bar{x}n\bar{x} && \text{definition of } \bar{x} \\
&= \sum_{i=1}^{n}({x_i}^2) - \bar{x}^2 n && \bar{x} \text{ times } \bar{x} \\
&= \sum_{i=1}^{n}({x_i}^2) - 2\bar{x}^2 n + \bar{x}^2 n && \text{add cancelling } \pm\bar{x}^2 \text{ terms} \\
&= \sum_{i=1}^{n}({x_i}^2) - 2\bar{x}^2 n + \sum_{i=1}^{n}(\bar{x}^2) && \text{rewrite term as a sum} \\
&= \sum_{i=1}^{n}({x_i}^2) - 2\bar{x}\sum_{i=1}^{n}(x_i) + \sum_{i=1}^{n}(\bar{x}^2) && \text{definition of } \bar{x} \\
&= \sum_{i=1}^{n}({x_i}^2 - 2\bar{x}x_i + \bar{x}^2) && \text{summation is linear} \\
&= \sum_{i=1}^{n}[(x_i - \bar{x})^2] && (a-b)^2 = a^2 - 2ab + b^2
\end{aligned}
$$

For the numerator the book provides $\sum_{i=1}^{n}[(x_i - \bar{x})(y_i - \bar{y})]$, i.e., n times $Cov(X, Y)$. Let's see how to get

there.

$$
\begin{aligned}
\sum_{i=1}^{n}(x_i y_i) - \bar{y}\sum_{i=1}^{n}(x_i) &= \sum_{i=1}^{n}(x_i y_i) - \bar{y}n\bar{x} && \text{definition of } \bar{x} \\
&= \sum_{i=1}^{n}(x_i y_i) - \bar{y}n\bar{x} + \bar{y}n\bar{x} - \bar{y}n\bar{x} && \text{add cancelling } \pm\bar{y}n\bar{x} \text{ terms} \\
&= \sum_{i=1}^{n}(x_i y_i) - \bar{y}n\bar{x} + \bar{y}n\bar{x} - \sum_{i=1}^{n}(\bar{y}\bar{x}) && \text{rewrite term as a sum} \\
&= \sum_{i=1}^{n}(x_i y_i) - \bar{y}n\bar{x} + \bar{x}\sum_{i=1}^{n}(y_i) - \sum_{i=1}^{n}(\bar{y}\bar{x}) && \text{definition of } \bar{y} \\
&= \sum_{i=1}^{n}(x_i y_i) - \bar{y}\sum_{i=1}^{n}(x_i) + \bar{x}\sum_{i=1}^{n}(y_i) - \sum_{i=1}^{n}(\bar{y}\bar{x}) && \text{definition of } \bar{x} \\
&= \sum_{i=1}^{n}(x_i y_i - \bar{y}x_i + \bar{x}y_i - \bar{y}\bar{x}) && \text{summation is linear} \\
&= \sum_{i=1}^{n}[(x_i - \bar{x})(y_i - \bar{y})] && (a-b)(c-d) = ac - ad - bc + bd
\end{aligned}
$$

If the numerator is n times $Cov(X, Y)$ and the denominator is n times $Var(X)$, we could cancel out the ns and calculate $\hat{\beta}_1$ as

$$\frac{Cov(X, Y)}{Var(X)}$$

Is there any difference in how long it takes in Python between calculating these different ways? We can use `timeit` to check. Let's also check that we get the same answer for all three. Note that Pandas has built-in `cov()` and `var()`; they default to sample covariance & variance, so we use `ddof=0` to use population covariance & variance.

```
setup = """
import pandas as pd
import numpy as np
c = pd.read_csv(r"E:\docs\Classes\ISL\Advertising.csv") \
    .rename(columns={"Unnamed: 0": "Id"}) \
    .set_index('Id')
b = c['sales'].values
"""

code_1 = """
((np.sum(c['TV']*b)-b.mean()*np.sum(c['TV'])) /
 (np.sum(c['TV']**2)-c['TV'].mean()*np.sum(c['TV'])))
"""

code_2 = """
(np.sum((c['TV']-c['TV'].mean())*(b-b.mean())) /
 np.sum((c['TV']-c['TV'].mean())**2))
"""

code_3 = """
c['TV'].cov(c['sales'], ddof=0) / c['TV'].var(ddof=0)
```

```
"""

print("Check time to calculate Beta_1 2,000 times:\n",
      np.round(timeit.timeit(code_1, setup=setup, number=2000), 2),
      "for first derivation\n",
      np.round(timeit.timeit(code_2, setup=setup, number=2000), 2),
      "for book derivation\n",
      np.round(timeit.timeit(code_3, setup=setup, number=2000), 2),
      "for Pandas cov(x,y)/var(x)\n")
```

```
## Check time to calculate Beta_1 2,000 times:
##  0.59 for first derivation
##  0.73 for book derivation
##  0.25 for Pandas cov(x,y)/var(x)
```

```
print("Check expected Beta_1 of 0.0475 vs. our three manual calcs:\n",
      np.round(eval(code_1), 4), "for first derivation\n",
      np.round(eval(code_2), 4), "for book derivation\n",
      np.round(eval(code_3), 4), "for Pandas cov(x,y)/var(x)")
```

```
## Check expected Beta_1 of 0.0475 vs. our three manual calcs:
##  0.0475 for first derivation
##  0.0475 for book derivation
##  0.0475 for Pandas cov(x,y)/var(x)
```

Everything ties out. Next up is multiple linear regression.

Multiple linear regression

On page 81 of the text we're given:

> Unlike the simple linear regression estimates given in (3.4), the multiple regression coefficient estimates have somewhat complicated forms that are most easily represented using matrix algebra. For this reason, we do not provide them here.

Let's take a look. Let's say:

- we put our n observations of the dependent variable y into vector $\mathbf{y}$, i.e., a vector of the y_i values
- we put our n observations of $m-1$ different independent variables x into $n \times m$ matrix X having n rows of m independent variables after adding in a column of ones for the intercept term
- we out our m coefficients $\beta_0, \ldots, \beta_{m-1}$ into vector $\mathbf{w}$ ("w" as in weights)

Then our RSS that we want to minimize is:

$$RSS = (\mathbf{y} - X\mathbf{w})^2$$

We still have a quadratic that we can minimize by finding where the derivative is zero, though now we're talking vectors, so we'll be looking for where the gradient with respect to w ∇_w is zero. Maybe this is easier to find if we first expand that expression, taking care to transpose as required:

$$\begin{aligned}(\mathbf{y} - X\mathbf{w})^2 &= (\mathbf{y} - X\mathbf{w})^T(\mathbf{y} - X\mathbf{w}) \\ &= \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T X\mathbf{w} + \mathbf{w}^T X^T \mathbf{w}\end{aligned}$$

Now we find the gradient:

$$\nabla_w = 0 - 2X^T\mathbf{y} + 2X^TX\mathbf{w}$$

Let's set it to zero and solve for w:

$$\begin{aligned}
-2X^T\mathbf{y} + 2X^TX\mathbf{w} &= 0 && \\
-2X^T\mathbf{y} + 2X^TX\mathbf{w} + 2X^T\mathbf{y} &= 0 + 2X^T\mathbf{y} && \text{add } 2X^T\mathbf{y} \text{ to both sides} \\
2X^TX\mathbf{w} &= 2X^T\mathbf{y} && \text{cancellation} \\
\frac{1}{2}2X^TX\mathbf{w} &= \frac{1}{2}2X^T\mathbf{y} && \text{multiply both sides by } \tfrac{1}{2} \\
X^TX\mathbf{w} &= X^T\mathbf{y} && \text{cancellation} \\
(X^TX)^{-1}X^TX\mathbf{w} &= (X^TX)^{-1}X^T\mathbf{y} && \text{multiply both sides by } (X^TX)^{-1} \\
\mathbf{w} &= (X^TX)^{-1}X^T\mathbf{y} && \text{cancellation}
\end{aligned}$$

Is (X^TX) always invertible? No, what if for instance we didn't notice that one of our x was linearly dependent on another of our x. That said, we should be fine as `np.linalg.lstsq` already supports that scenario. We can test this out by adding a linearly dependent column and comparing the results between calling `np.linalg.lstsq` directly, using multiplication and inversion with `@` and `np.linalg.inv`, and using multiplication and pseudo-inversion with `@` and `np.linalg.pinv`. Let's match variable names with what we just derived above.

```
# make our first and second x linearly dependent
X = np.column_stack((c['TV'].values,
                     2 * c['TV'].values,
                     np.ones(len(c['TV'].values))))
y = b
```

Now to attempt $(X^TX)^{-1}X^T\mathbf{y}$ on a non-invertible matrix. We'll get an error, so we'll use `try/except` to fail gracefully.

Note that the `.T` is for transpose, so for instance X^TX with NumPy will be `X.T @ X`.

```
# multiplication and inverse will fail
try:
    np.linalg.inv(X.T @ X) @ X.T @ y
except np.linalg.LinAlgError as e:
    print("LinAlgError:", e)
```

```
## LinAlgError: Singular matrix
```

It fails as expected. The pseudo-inverse should still work in place of $(X^TX)^{-1}X^T\mathbf{y}$, and our least squares should give the same answer.

```
# multiplication and pseudo-inverse will succeed
np.linalg.pinv(X.T @ X) @ X.T @ y
```

```
## array([0.00950733, 0.01901466, 7.03259355])
```

```
# least squares will succeed
np.linalg.lstsq(X, y, rcond=None)[0]
```

```
## array([0.00950733, 0.01901466, 7.03259355])
```

It worked. Arguably for our purposes here it will be best to always use `np.linalg.lstsq`. Now on to the multiple linear regression.

We can re-create table 3.4 in chapter 3 as a one-liner with Pandas built-in `corr()`. To print just the upper triangle as was done in the book we can use a mask.

```
print("Correlation matrix\n",
      c.corr()
      .round(4)
      .where(np.triu(np.ones_like(c.corr(), dtype=bool)))
      .fillna(''))
```

```
## Correlation matrix
##              TV   radio newspaper   sales
## TV         1.0  0.0548    0.0566  0.7822
## radio            1.0    0.3541  0.5762
## newspaper               1.0  0.2283
## sales                        1.0000
```

We see that `newspaper` is correlated with `radio`, so we can expect some issues with a multiple linear regression using all four, but we continue as was done in the book just to check the hand-coded fit. Let's return to using `A` and `b` as we did earlier. First we set up our `A` to have *four* columns.

```
A = np.column_stack((c['newspaper'].values,
                     c['radio'].values,
                     c['TV'].values,
                     np.ones(len(c['TV'].values))))
```

Our vector `b` remains unchanged. As before it's just a one-liner to get the least squares fit. Let's also check that we get the same answer using `np.linalg.inv` and `np.linalg.pinv`.

```
x, RSS, rank, S = np.linalg.lstsq(A, b, rcond=None)

print("   Book coeffs: -0.001, 0.189, 0.046, 2.939\n  lstsq coeffs: "
      + ', '.join("{:.3f}".format(value) for value in np.round(x, 3)) +
      "\n  @ inv coeffs: "
      + ', '.join("{:.3f}".format(value) for value in
                  np.round(np.linalg.inv(A.T @ A) @ A.T @ b, 3)) +
      "\n @ pinv coeffs: "
      + ', '.join("{:.3f}".format(value) for value in
                  np.round(np.linalg.pinv(A.T @ A) @ A.T @ b, 3)))
```

```
##    Book coeffs: -0.001, 0.189, 0.046, 2.939
##   lstsq coeffs: -0.001, 0.189, 0.046, 2.939
##   @ inv coeffs: -0.001, 0.189, 0.046, 2.939
##  @ pinv coeffs: -0.001, 0.189, 0.046, 2.939
```

The `np.linalg.lstsq` approach was very straightforward, and the results tied with the book and the other methods. Next up: let's model an interaction term as was done in equation 3.31.

Interaction term

Equation 3.31 gives us a model for `sales` based on `TV`, `radio`, and the product `TV * radio`.

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2$$

We'll see if we can code that up and get the same answer as the book, sticking with the $A\mathbf{x} = \mathbf{b}$ approach. As before, we use `column_stack` to create the columns of our new A.

```
A = np.column_stack((c['TV'].values * c['radio'].values,
                     c['radio'].values,
                     c['TV'].values,
                     np.ones(len(c['TV'].values))))
```

Same vector `b`, another one-liner to fit. We'll check against the given coefficients in table 3.9 of the book.

```
x, RSS, rank, S = np.linalg.lstsq(A, b, rcond=None)

print("Expected coeffs of 0.0011, 0.0289, 0.0191, 6.7502 found: "
      + ', '.join("{:.4f}".format(value) for value in np.round(x, 4)))
```

```
## Expected coeffs of 0.0011, 0.0289, 0.0191, 6.7502 found: 0.0011, 0.0289, 0.0191, 6.7502
```

Everything ties.

All three of these types of models were very straightforward to fit to the `Advertising` data.