Coding Your Own Linear Regression Model

One task that you will almost certainly be required to do other data science courses (especially if you are a MIDS student) is to write up some of your statistical / machine learning models from scratch. This can be a very valuable exercise, as it ensures that you understand what is actually going on behind the scenes of the models you use ever day, and that you don't just think of them as "black boxes".

To get a little practice doing this, today you will be coding up your own linear regression model!

(If you are using this site but aren't actually in this class, you are welcome to skip this exercise if you'd like -- this is more about practicing Python in anticipation of the requirements of other courses than developing your applied data science skills.)

There are, broadly speaking, two approaches you can take to coding up your own model:

1. you can write the model by defining a new function, or
2. you can write the model by defining a new class with associated methods (making a model that works the way a model works in `scikit-learn`).

Whether you do 1 or 2 is very much a matter of choice and style. Approach one, for example, is more consistent with what is called a *functional* style of programming, while approach two is more consistent with an *object-oriented* style of programming. Python can readily support both approaches, so either would work fine.

In these exercises, however, I will ask you to use approach number 2 as this *tends* to be the more difficult approach, and so practicing approach 2 will be extra useful in preparing you for other classes (HA! Pun...). In particular, our goal is to implement a linear regression model that has the same "initialize-fit-predict-score" API (application programming interface -- a fancy name for the methods a class supports) as `scikit-learn` models. That means your model should be able to do all of the following:

1. **Initialize** a new model.
2. **Fit** a linear model when given a numpy vector (`y`) and a numpy matrix (`X`) with the syntax `my_model.fit(X, y)`.
3. **Predict** values when given a new `numpy` matrix (`X_test`) with the syntax `my_model.predict(X_test)`.
4. Return the **model coefficients** through the property `my_model.coefficients` (not quite what is used in `sklearn`, but let's use that interface).

Also, bear in mind that throughout these exercises, we'll be working in `numpy` instead of `pandas`, just as we do in `scikit-learn`. So assume that before using your model, your

user has already converted their data from `pandas` into `numpy` arrays.

(1) Define a new Class called `MyLinearModel` with methods for `__init__`, `fit`, `predict`, and an attribute for `coefficients`. For now, we don't need any initialization *arguments*, just an `__init__` function.

As you get your code outline going, start by just having each method `pass`:

```
def my_method(self):
    pass
```

This will allow your methods to run without errors (they just don't do anything). Then we can double back to each method to get them working one by one.

In [ ]:
```
class MyLinearModel:
    def __init__(self):
        self.coefficients = None
        pass

    def fit(self, X, y):
        pass

    def predict(self, X):
        pass
```

(2) Now define your `fit` method. This is the method that should actually run your linear regression. In case you've forgotten your linear algebra, remember that for linear regressions, $\beta = (X'X)^{-1}X'Y$, a fact you can see explained in detail on page four here.

Note that once you have written the code to do a linear regression, you'll need to put your outputs (your coefficents) somewhere. I recommend making an attribute for your class where you can store your coefficients.

(As a reminder: the normal multiply operator (`*`) in `numpy` implies scalar multiplication. Use `@` for matrix multiplication).

HINT: Remember that linear regressions require a vector of 1s in the `X` matrix. As the package writer, you get to decide whether users are expected to provide a matrix `X` that already has a vector of 1s, or whether you expect the user to provide a matrix `X` that doesn't have a vector of 1s (in which case you will need to add a vector of 1s before you fit the model).

In [ ]:
```
import numpy as np


class MyLinearModel:
    def __init__(self):
        self.coefficients = None

    def fit(self, X, y):
        X = np.hstack((np.ones((X.shape[0], 1)), X))
```

```
        # Compute the coefficients using the normal equation
        # beta = (X'X)^{-1}X'y
        X_transpose = X.T
        beta = np.linalg.inv(X_transpose @ X) @ X_transpose @ y

        # Store the coefficients
        self.coefficients = beta

    def predict(self, X):
        # This method will be defined later
        pass
```

```
c:\Users\kbagh\miniconda3\Lib\site-packages\numpy\_distributor_init.py:30: UserWarni
ng: loaded more than 1 DLL from .libs:
c:\Users\kbagh\miniconda3\Lib\site-packages\numpy\.libs\libopenblas64__v0.3.21-gcc_1
0_3_0.dll
c:\Users\kbagh\miniconda3\Lib\site-packages\numpy\.libs\libopenblas64__v0.3.23-246-g
3d31191b-gcc_10_3_0.dll
  warnings.warn("loaded more than 1 DLL from .libs:"
```

(3) As you write code, it is good to test your code as you work. With that in mind, let's create some toy data. First, create a 100 x 2 matrix where each column is normally distributed. Then create a vector `y` that is a linear combination of those two columns **plus** a vector of normally distributed noise **and** a constant term.

In other words, we want to create data where we *know* exactly what coefficients we should see so when we test our regression, we know if the results are accurate!

In [ ]:
```
np.random.seed(0)

X = np.random.randn(100, 2)

true_coefficients = np.array([3, -2])
intercept = 5

noise = np.random.randn(100)

y = intercept + X @ true_coefficients + noise


X[:5], y[:5]
```

Out[ ]:
```
(array([[ 1.76405235,  0.40015721],
        [ 0.97873798,  2.2408932 ],
        [ 1.86755799, -0.97727788],
        [ 0.95008842, -0.15135721],
        [-0.10321885,  0.4105985 ]]),
 array([ 9.12266078,  3.21504838, 13.65688933,  8.8082434 ,  4.50927797]))
```

The following is the first 5 entries for both rows in matrix X. As well, it has the first 5 entries for vector y, which is the linear combination of the columns in matrix X

(4) Now test whether you `fit` method generates the correct coefficients. Remember the choice you made in Question 2 about whether your package expects the users' `X` matrix to include a vector of 1s when you test!

In [ ]:
```
print("The following is the true coefficients that we want to properly derive")
true_coefficients
```

```
The following is the true coefficients that we want to properly derive
```

Out[ ]:
```
array([ 3, -2])
```

In [ ]:
```
linear_model = MyLinearModel()
linear_model.fit(X, y)

print(
    "The following is first, the intercept, and second, the coefficients our linear
)
print(
    "What we can see is that the coefficients are very close to what the true coeff
)
linear_model.coefficients
```

```
The following is first, the intercept, and second, the coefficients our linear regre
ssion attempted to dervie
What we can see is that the coefficients are very close to what the true coefficient
s should be, meaning our model works well
```

Out[ ]:
```
array([ 4.94858887,  3.1104649 , -2.0540371 ])
```

(5) Now let's make the statisticians proud, and in addition to storing the coefficients, let's store the standard errors for our estimated coefficients as another attribute. Recall that the simplest method of calculating the variance covariance matrix for β is using the formula $\sigma^2(X'X)^{-1}$, where σ^2 is the variance of the error terms of your regression. The standard errors for your coefficient estimates will be the diagonal values of that matrix. See page six here for a full derivation.

In [ ]:
```
class MyLinearModel:
    def __init__(self):
        self.coefficients = None
        self.std_errors = None

    def fit(self, X, y):
        X_with_intercept = np.hstack((np.ones((X.shape[0], 1)), X))

        X_transpose = X_with_intercept.T
        beta = np.linalg.inv(X_transpose @ X_with_intercept) @ X_transpose @ y

        self.coefficients = beta

        residuals = y - X_with_intercept @ beta

        residual_variance = np.sum(residuals**2) / (
            X_with_intercept.shape[0] - X_with_intercept.shape[1]
        )
```

```
        varcov_beta = residual_variance * np.linalg.inv(X_transpose @ X_with_interc

        self.std_errors = np.sqrt(np.diag(varcov_beta))

    def predict(self, X):
        # Predictions method will be defined later
        pass
```

(6) Now let's also add an R-squared attribute to the model.

In [ ]:
```
class MyLinearModel:
    def __init__(self):
        self.coefficients = None
        self.std_errors = None
        self.r_squared = None

    def fit(self, X, y):
        X_with_intercept = np.hstack((np.ones((X.shape[0], 1)), X))

        X_transpose = X_with_intercept.T
        beta = np.linalg.inv(X_transpose @ X_with_intercept) @ X_transpose @ y

        self.coefficients = beta

        residuals = y - X_with_intercept @ beta

        residual_variance = np.sum(residuals**2) / (
            X_with_intercept.shape[0] - X_with_intercept.shape[1]
        )

        varcov_beta = residual_variance * np.linalg.inv(X_transpose @ X_with_interc

        self.std_errors = np.sqrt(np.diag(varcov_beta))

        # Calculate R-squared
        ss_res = np.sum(residuals**2)
        ss_tot = np.sum((y - np.mean(y)) ** 2)
        self.r_squared = 1 - ss_res / ss_tot

    def predict(self, X):
        # This method will be defined later
        pass
```

In [ ]:
```
linear_model = MyLinearModel()
linear_model.fit(X, y)


print(
    "The following below is the fitted coefficients, the standard errors, and the R
)
linear_model.coefficients, linear_model.std_errors, linear_model.r_squared
```

```
The following below is the fitted coefficients, the standard errors, and the R-squar
ed value of our linear regression model
```

Out[ ]:
```
(array([ 4.94858887,  3.1104649 , -2.0540371 ]),
 array([0.09674321, 0.09376987, 0.09433688]),
 0.9431757673307737)
```

(7) Now we'll go ahead and cheat a little. Use `statsmodels` to fit your model with your toy data to ensure your standard errors and r-squared are correct!

In [ ]:
```
import statsmodels.api as sm

# Add a constant term to X for the intercept
X_with_intercept = sm.add_constant(X)

model = sm.OLS(y, X_with_intercept)
results = model.fit()

r_squared_sm = results.rsquared
std_errors_sm = results.bse
coefficients_sm = results.params

print(
    "The following below is the fitted coefficients, the standard errors, and the R
)
print(
    "In comparison to above, we clearly see that the linear model class worked exac
)
coefficients_sm, std_errors_sm, r_squared_sm
```

```
The following below is the fitted coefficients, the standard errors, and the R-squar
ed value from statsmodels
In comparison to above, we clearly see that the linear model class worked exactly as
intended
```

Out[ ]:
```
(array([ 4.94858887,  3.1104649 , -2.0540371 ]),
 array([0.09674321, 0.09376987, 0.09433688]),
 0.9431757673307737)
```

(8) Now implement `predict`! Then test it against your original `X` data -- do you get back something very close to your true `y`?

In [ ]:
```
class MyLinearModel:
    def __init__(self):
        self.coefficients = None
        self.std_errors = None
        self.r_squared = None

    def fit(self, X, y):
        X_with_intercept = np.hstack((np.ones((X.shape[0], 1)), X))

        X_transpose = X_with_intercept.T
        beta = np.linalg.inv(X_transpose @ X_with_intercept) @ X_transpose @ y

        self.coefficients = beta

        residuals = y - X_with_intercept @ beta
```

```
        residual_variance = np.sum(residuals**2) / (
            X_with_intercept.shape[0] - X_with_intercept.shape[1]
        )

        varcov_beta = residual_variance * np.linalg.inv(X_transpose @ X_with_interc

        self.std_errors = np.sqrt(np.diag(varcov_beta))

        # Calculate R-squared
        ss_res = np.sum(residuals**2)
        ss_tot = np.sum((y - np.mean(y)) ** 2)
        self.r_squared = 1 - ss_res / ss_tot

    def predict(self, X):
        # Check if the model has been fit
        if self.coefficients is None:
            raise ValueError("Model has not been fit yet.")

        X_with_intercept = np.hstack((np.ones((X.shape[0], 1)), X))

        return X_with_intercept @ self.coefficients
```

In [ ]:
```
linear_model = MyLinearModel()
linear_model.fit(X, y)
y_pred = linear_model.predict(X)

print(
    "As we can see here, the first 5 predictors are incredibly close to the true va
)
y_pred[:5], y[:5]
```

```
As we can see here, the first 5 predictors are incredibly close to the true values,
showing that the predict method works as intended
```

Out[ ]:
```
(array([ 9.61367403,  3.39004125, 12.76492748,  8.21469887,  3.7841457 ]),
 array([ 9.12266078,  3.21504838, 13.65688933,  8.8082434 ,  4.50927797]))
```

(9) Finally, create the *option* of fitting the model with or without a constant term. In other words, create an option so that, if the user passes a numpy array *without* a constant term, your code will add a vector of 1s before fitting the model. As in `scikit-learn`, make this an option you set during initialization.

In [ ]:
```
class MyLinearModel:
    def __init__(self, fit_intercept=True):
        self.coefficients = None
        self.std_errors = None
        self.r_squared = None
        self.fit_intercept = fit_intercept

    def fit(self, X, y):
        if self.fit_intercept:
            X_with_intercept = np.hstack((np.ones((X.shape[0], 1)), X))
        else:
            X_with_intercept = X
```

```
        X_transpose = X_with_intercept.T
        beta = np.linalg.inv(X_transpose @ X_with_intercept) @ X_transpose @ y

        self.coefficients = beta

        residuals = y - X_with_intercept @ beta

        residual_variance = np.sum(residuals**2) / (
            X_with_intercept.shape[0] - X_with_intercept.shape[1]
        )

        varcov_beta = residual_variance * np.linalg.inv(X_transpose @ X_with_interc

        self.std_errors = np.sqrt(np.diag(varcov_beta))

        ss_res = np.sum(residuals**2)
        ss_tot = np.sum((y - np.mean(y)) ** 2)
        self.r_squared = 1 - ss_res / ss_tot

    def predict(self, X):
        if self.coefficients is None:
            raise ValueError("Model has not been fit yet.")

        if self.fit_intercept:
            X_with_intercept = np.hstack((np.ones((X.shape[0], 1)), X))
        else:
            X_with_intercept = X

        return X_with_intercept @ self.coefficients
```

In [ ]:
```
# Instantiate the model without an intercept, fit it, and predict
linear_model_no_intercept = MyLinearModel(fit_intercept=False)
linear_model_no_intercept.fit(X, y)
coefficients_no_intercept = linear_model_no_intercept.coefficients

# Instantiate the model with an intercept (default), fit it, and predict
linear_model_with_intercept = MyLinearModel()
linear_model_with_intercept.fit(X, y)
coefficients_with_intercept = linear_model_with_intercept.coefficients

print(
    "The following below shows that the code will properly check whether or not a c
)
coefficients_no_intercept, coefficients_with_intercept
```

```
The following below shows that the code will properly check whether or not a constan
t term was added
```

Out[ ]:
```
(array([ 3.12204331, -1.38230335]),
 array([ 4.94858887,  3.1104649 , -2.0540371 ]))
```

In [ ]:
```

```