Math 118 Mini Project

Author: Jiayu Li

- UID: 605-348-766
- Date: 03/15/2023

Understanding the Proof of Linear Regression and Implementing it in Python

A supervised machine learning task has four major components:

1. The *predictor variables* X and the *target variable* Y, which we aim to predict using X.
2. The *model* f. We treat $f(X)$ as our estimate of Y.
3. The learning algorithm, which in this quarter lecture note we will do by hand.
4. *The loss function* $\mathcal{L}$. The quantity $\mathcal{L}(f(X), Y)$ is a measure of how well the model f "fits" the data (X, Y).

In the next, we will explore each of these components in an interpretable setting -- linear regression.

Linear Regression

Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables. In this project, we will explore the proof of linear regression and implement it using Python.

Part 1: Understanding the Proof of Linear Regression

Linear regression aims to fit a line to a set of data points that best represents the relationship between the dependent variable and the independent variable(s). In linear regression, we can use a *linear* model for the data. In the 1-dimensional case, this means that our model f has the form

$$f(x) = ax + b \approx y \ .$$

There are two parameters: the slope a and the intercept b. By changing the slope and intercept, we get different models. We say that f belongs to a *family* of models $\mathcal{M}$, with each model corresponding to a different choice of a and b. The goal of linear regression is to find the values of a and b that minimize the sum of the squared errors between the predicted values of y and the actual values of y. This is known as the least-squares method.

The least squares method works by finding the values of m and b that minimize the following function:

$$J(a, b) = (1/2a) * \sum_{i=1}^{n} (y_{pred_i} - y_i)^2$$

where y_{pred_i} is the predicted value of y for the i-th input, y_i is the actual value of y, and n is the number of inputs.

To find the values of m and b that minimize $J(a, b)$, we take the partial derivatives of $J(a, b)$ with respect to m and b and set them equal to zero. This gives us two equations that we can solve for a and b:

$$a = \sum_{i=1}^{n}(x_i - x_{mean}) * (y_i - y_{mean}) \sum_{i=1}^{n}(x_i - x_{mean})^2$$

$$b = y_{mean} - a * x_{mean}$$

where x_mean and y_mean are the means of the input and target variables, respectively.

Our learning task now is to find good choices for a and b, given some data.

Predictor and Target Data

Part 2: Implementing Linear Regression in Python

Python implementation:

To implement linear regression in Python. Let's now generate some synthetic data to use as our example.

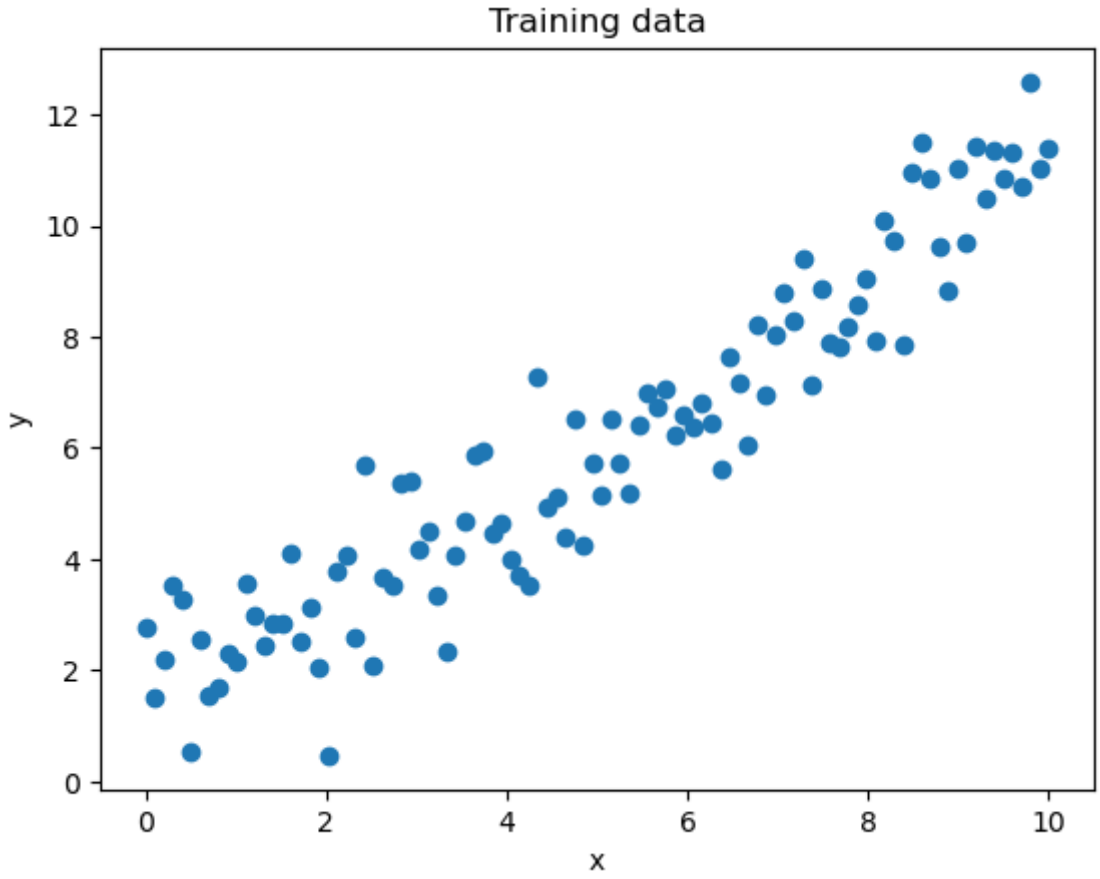
Let's now implement linear regression using Python. We will use the scikit-learn library to split the data into training and testing sets, train the model, and make predictions. We will also use the matplotlib library to plot the training data.

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

In [29]:
```
# controls random number generation
# Generate random data
np.random.seed(0)
# true model is linear with a = 1 and b = 1
a = 1
b = 1

n = 100
x = np.linspace(0,10,n)
y = a * x + b + np.random.randn(n) # final term is random noise
```

In [30]:
```
# Plot training data
plt.scatter(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Training data')
plt.show()
```



The output should be a scatter plot of the training data.

Next, we split the data into training and testing sets:

In [6]:
```
# Split data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, ra
```

In [7]:
```
# Train the model
model = LinearRegression()
model.fit(x_train.reshape(-1, 1), y_train)
```

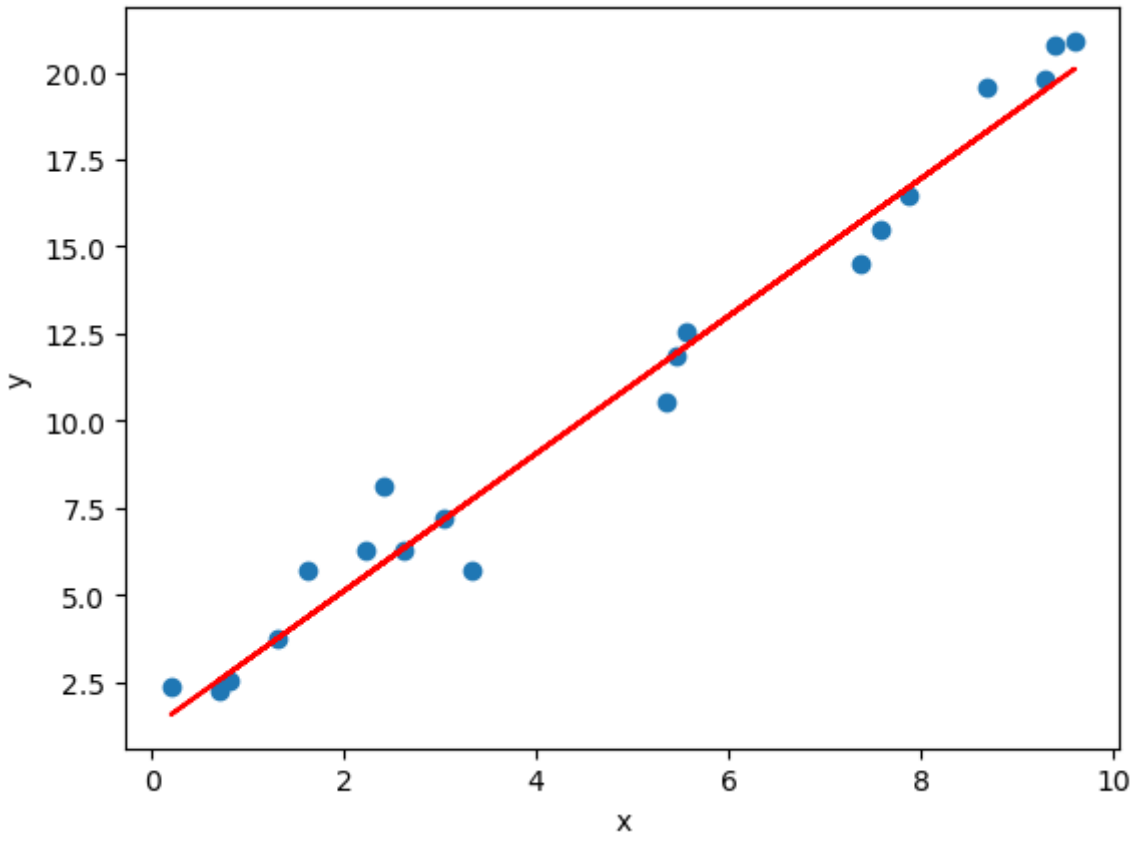
Out[7]:
```
LinearRegression()
```

In [8]:
```
# Make predictions on the testing set
y_pred = model.predict(x_test.reshape(-1, 1))
```

Finally, we can plot the testing set and the predicted values:

In [9]:
```
# Plot testing set and predicted values
plt.scatter(x_test, y_test)
plt.plot(x_test, y_pred, color='red')
plt.xlabel('x')
plt.ylabel('y')
```

Out[9]: Text(0, 0.5, 'y')



Some of these models look better than others! How do we pick, systematically? Well, that's where the loss function $\mathcal{L}$ comes in. The most common choice in linear regression is the *mean-square error*, which is defined as follows:

$$\mathcal{L}(f(X), Y) = \frac{1}{n}\left[(y_1 - f(x_1))^2 + (y_2 - f(x_2))^2 + \cdots + (y_n - f(x_n))^2\right]$$

A term like $(y_i - f(x_i))^2$ is large when $f(x_i)$ is very different from y_i -- that is, when our prediction is off! So, if a model has a low mean-square error $\mathcal{L}$, then this indicates that the model "fits the data" well.

Let's implement the mean-square erro for linear regression. The error depends on the parameters a and b. `numpy` array operations make this very easy.

So, let's see if we can learn some good parameters for the data. First, let's formalize the model as a function.

In [13]:
```
def f(x, a, b):
    return a*x + b
```

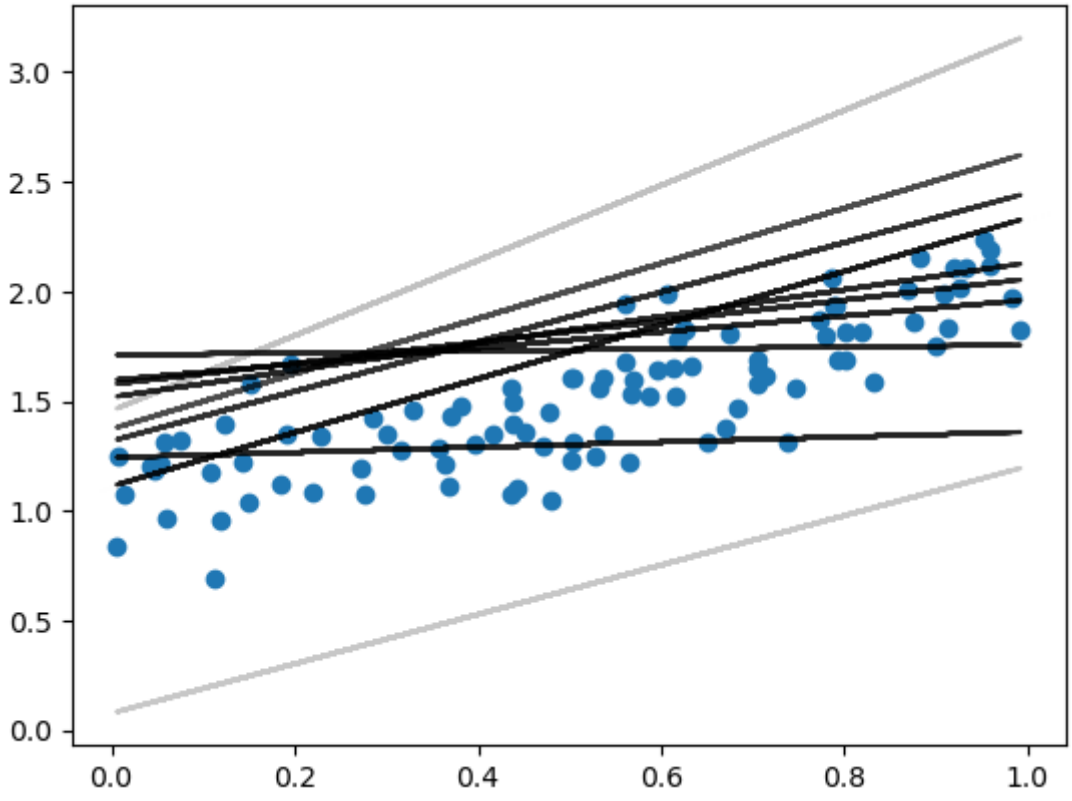
In [7]:
```
def linear_MSE(x, y, a, b):
    preds = f(x, a, b)
    return ((y - y_pred)**2).mean()
```

Now let's go back to our plot of the data, and show how all those candidate models fare with regards to the MSE loss function. We're going to tune our visualization so that the models with lower MSE are drawn thicker:

In [8]:
```
fig, ax = plt.subplots(1)
ax.scatter(x,y)

for i in range(10):
    # a and b both random between 0 and 2
    a = 2*np.random.rand()
    b = 2*np.random.rand()

    ax.plot(x,
            f(x, a, b),
            color = "black",
            alpha = 1 - min(linear_MSE(x, y, a, b), 1))
```



Hey, this looks pretty good! The models that have lower MSE (darker lines) "look close" to the data.

Let's see if we can estimate a and b. One way to do this is by simply generating a lot of random possibilities and picking the best one. Let's plot a number of models and highlight the best one in a different color.

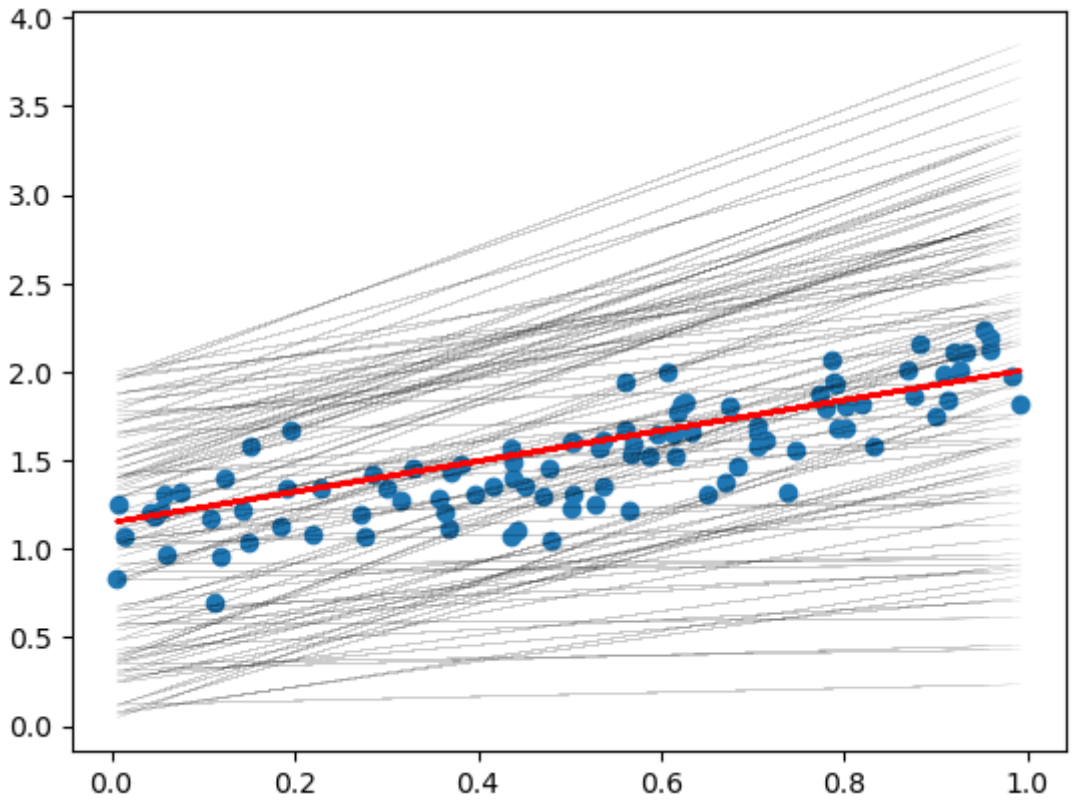
In [9]:
```
fig, ax = plt.subplots(1)
ax.scatter(x,y)

best_a, best_b = 0, 0
best_error = np.inf
for i in range(100):
    a = 2*np.random.rand()
    b = 2*np.random.rand()

    error = linear_MSE(x, y, a, b)
    if error < best_error:
        best_error = error
        best_a, best_b = a,b
    preds = f(x, a, b)
    ax.plot(x, preds, color = "black", alpha = 0.2, linewidth = .1)

best_preds = f(x, best_a, best_b)
ax.plot(x, best_preds, color = "red")
```

Out[9]:
```
[<matplotlib.lines.Line2D at 0x107d34460>]
```



In [10]:
```
# true parameters were a = b = 1
best_a, best_b
```

Out[10]:
```
(0.8636016969818487, 1.1484688275661288)
```

Of course, in real life we wouldn't determine the parameters this way. Instead, we'd minimize the function.

In [11]:
```
from scipy.optimize import minimize
res = minimize(lambda z: linear_MSE(x, y, z[0], z[1]), np.array([0,0]))
best_a, best_b = res.x
best_a, best_b
```

Out[11]:
```
(0.9765299071953119, 1.015261733328459)
```

This is exactly what is going on "under the hood" of most prepackaged machine learning algorithms, which we'll begin to see in the next few lectures.

Having obtained the optimal parameters, we are now able to make predictions on unseen data. For example:

In [12]:
```
f(np.array([0.7]), best_a, best_b) # model prediction when X = 0.7
```

Out[12]:
```
array([1.69883267])
```

Conclusion

In this project, we have explored the proof of linear regression and implemented it in Python. Linear regression is a powerful tool in the field of statistics and machine learning, and understanding its proof is essential for its successful application.

In this quarter, we did linear regression "by hand." We generated some synthetic predictor data and target data. We then modeled the data using a family of one-dimensional linear models, and selected from among the many possibilities using the mean square error loss function. Choosing the model that minimized the loss function led to a good "fit" to the data.

This pattern applies to essentially all problems in (supervised) machine learning:

1. Collect some predictor data and target data.
2. Define a family of models and loss function.
3. Find the element of the model family that minimizes the loss function.

There are a few outstanding issues that I haven't proved here. The biggest one is that "fitting the data" is not actually what we usually care about -- we care about *predicting* unseen data. It turns out that fitting the data too closely can actually be counter productive in this case. This is the problem of *overfitting*.