Simple Linear Regressions

Imagine you're moving out of town and need to sell your house. How do you determine a fair asking price? You might compare it to your neighbor's house, which has the same number of bedrooms—but yours has a bigger backyard, so you think it should be worth more. Another friend's house has the same square footage and number of bedrooms as you but is located in a more preferable neighborhood, which could increase its value.

If you had a dataset with details on all the houses in town, including their features and sale prices, how could you use it to predict your home's value? You could employ a linear regression model.

What is Linear Regression?

A linear regression is a model of the relationship between one dependent variable and one or more independent variables. In our example, the price of a house is *dependent* on several factors - the number of bedrooms, bathrooms in the house, its location, and so on. If your house has 3 bedrooms, you can't change that in the immediate future so it is what it is. Formally, the 'bedrooms' feature is *independent*.

When Should You Use Linear Regression?

You've learned what linear regression is and how to apply it, but how can you be sure it works for your data? Before diving into the code, check if your data meets some key assumptions. For now, let's assume our model will predict prices using one predictor[1]. These assumptions guide the model's reliability.

[1] For a better understanding of this term and those to follow, please refer to my previous post introducing regressions.

1. Linearity

It is hard to understand the relationship between variables just by looking at the data, so let's plot it. Here is a scatter plot of the variables price of a house against the square footage of a house:

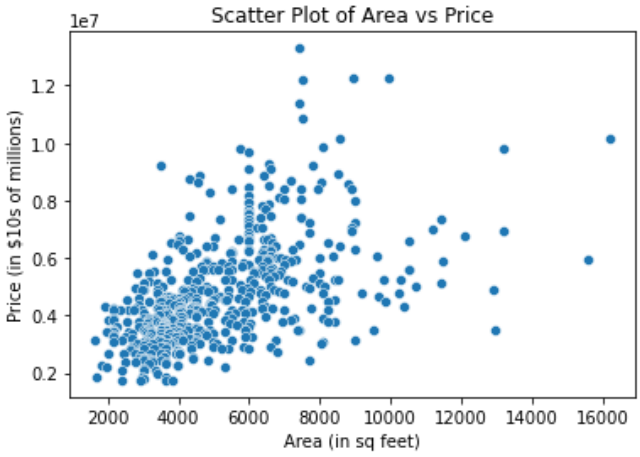

Figure 1[2]

Squint your eyes to the bottom left of the graph - the trend looks like as the area of a house increases, the price increases too. If you had to run a line through the data points that would best depict the relationship it would look like this:

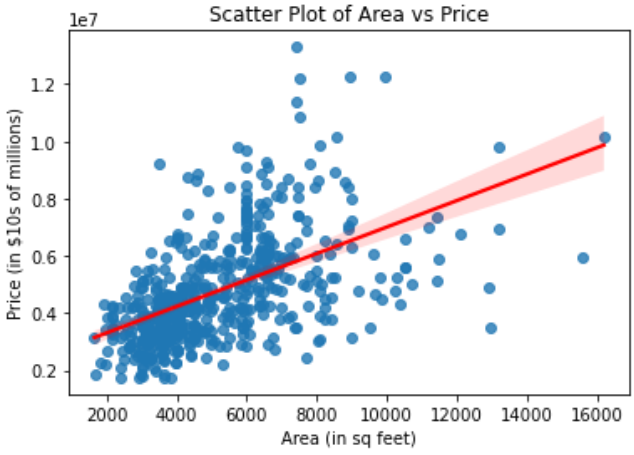

Figure 2

The *line of best fit* a.k.a the regression line running through the data could be at any angle, but as long as the trajectory of the data looks like a line, you know that the relationship is *linear*.

[2] You can view how this and the following graphs have been built in my code file.

2. Independence of observations

This assumption tells us that each data point in the problem is not correlated with each other. Suppose your aunt talks to her neighbors about how awesome it is to have five bathrooms in her 4-bedroom house. One neighbor, completely mesmerized by the idea, decides to renovate his two-bedroom home just to add three more bathrooms. Your aunt's house just influenced his decision, which means their house prices are no longer independent observations. In real life, you'll see a lack of independence between observations in financial data. Stock prices, for instance, rarely move in isolation—if one stock crashes, it can trigger a domino effect on others. In such cases, the assumption of independence is clearly violated.

When observations are highly correlated, linear regressions struggle to correctly estimate the effect of each variable. The model then assigns misleading importance to features, leading to unstable and unreliable predictions.

3. Homoscedasticity

This tongue-twister term refers to a condition in which the variance of the residual in a regression model is constant. *Residuals* are the differences between the actual values and the predicted values from a model. When homoscedasticity holds, linear regressions are reliable throughout the range of independent values and the model can produce more accurate and less biased predictions. In the graph below, residuals are indicated by the green lines drawn from each blue observation example dots to the line of best fit. In a homoscedastic dataset, the lengths of the green lines do not vary a lot.

Figure 3

But, that's not the case with the housing dataset used. Instead of forming an even spread, the data points start fanning out as house prices increase. This creates a cone-shaped pattern, a classic sign of **heteroscedasticity**—the opposite of what you want. Heteroscedasticity can be dealt with in many ways: logarithmic transformations, using Robust Standard Errors, or simply omitting outliers from the dataset which could allow us to handle inconsistencies more accurately. To demonstrate this, I've built a Heteroscedasticity Function of Fun™. Follow the instructions at the end of my code file to see how tweaking the dataset—by filtering out rows based on residual percentiles—impacts model predictions using various evaluation metrics.

4. Normality of error distribution

The error term (ε) represents the difference between the observed value and the true, unobservable regression function prediction. Since a perfect line of fit can rarely be modeled, we settle for a line of best fit—an estimate of the true relationship between variables. Because we can't know the exact difference between an ideal prediction and the actual value, we use residuals as approximations of error terms. This assumption states that the distribution of residuals should be normal.

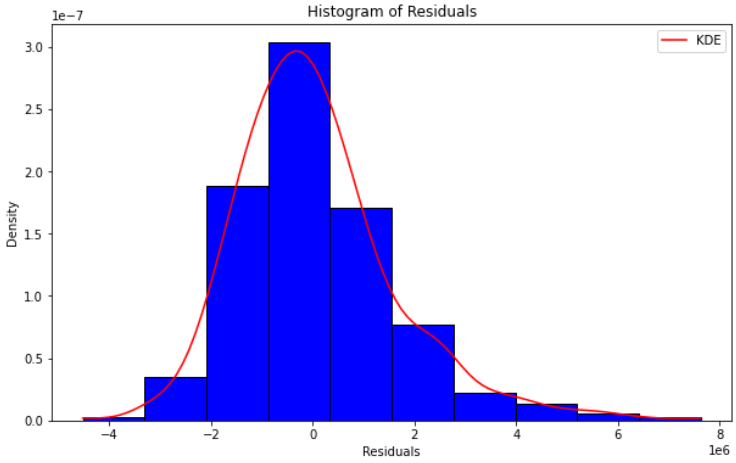

Figure 4

In this dataset, the residuals generally follow a normal distribution, though there's a slight skew to the right. This suggests that there may be room for improvement in the model, such as applying a logarithmic transformation, which can help improve the accuracy of the model's predictions.

Now that we've covered the key assumptions, let's put linear regression to work and estimate your house price.

Can We Build it? Yes, We Can!

With Python, initializing and training a linear regression model is remarkably straightforward—it takes just two lines of code:

```
model = LinearRegression()
model.fit(X, Y)
```

The first line sets up the model, and the second line uses the fit() function to learn the relationship between square footage and house price. Python's Scikit-learn library makes it that easy to initialize a LinearRegression model, but what's going on inside the 'black box'? In machine learning, when we call something a 'black box', we mean that the internal process is hidden from us. Scikit-learn makes it easy to use this function, but we still want to understand what's going on inside. Even though the code for a linear regression is easy to reproduce, the math is going to take a little more effort. Let's start easy by looking at what happens behind the scenes of a simple linear regression. Let's break it down step by step.

Key Notation in Linear Regression

Before diving into the math, here are some fundamental terms:

- x_i - An independent observation (e.g. area of the 5th house in the dataset would be x_5).
- y_i - A dependent observation (e.g. price of the 2nd house in the dataset would be y_2).
- $\bar{x}$ - The mean of the independent variable *X* (read as "x-bar").
- $\hat{y}_i$ - The predicted value of y_i. (read as "y-hat-i"). Hats in statistics indicate predicted values—when they're not used for sun protection.
- *X/Y* - The generalized term for independent and dependent variables (used in equations).
- $\sum_{i=1}^{n}$ - Indicates summation of all values from 1 to n - the length of the dataset.

The goal is to train a linear regression model on X and Y to predict future Y values given new X values. The dataset will be split 80% for training, and 20% for testing to evaluate the model's performance.

Math in the Black Box

Now that we've covered the necessary assumptions and notation for simple linear regression, you can start applying them to build our model and make predictions. Let's dive into the math behind how coefficients are estimated. Simple linear regressions follow the equation:

$$Y = \beta 0 + \beta 1X + \varepsilon$$

Where

1. $\boldsymbol{\beta 0}$ is the y-intercept. It is the predicted value of Y when X is 0 and is denoted as the black dot in the plot below.

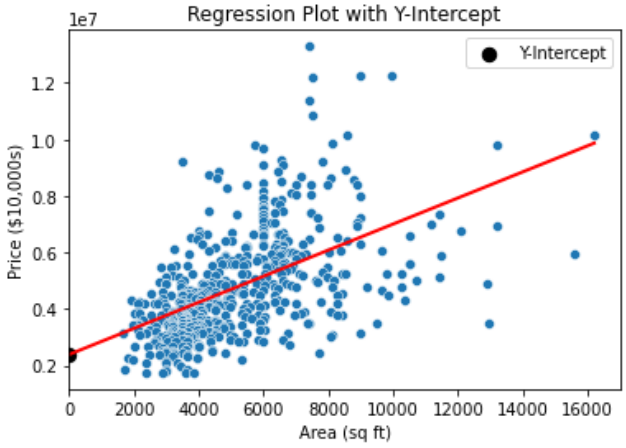

Figure 5

2. $\boldsymbol{\beta 1}$ is the slope of the regression line, which captures the rate at which Y changes for every change in unit of X. It is calculated using two points (x_1, y_1) and (x_2, y_2) with the formula: $\frac{(y_2 - y_1)}{(x_2 - x_1)}$. This formula tells us how the independent variable X and the dependent variable Y co-vary, which helps us understand the strength and direction of the relationship between them.

3. $\boldsymbol{\varepsilon}$ is the error term - the unexplained variation in Y, covered in our normality of error distribution assumption.

This equation is very similar to the equation of a line in geometry; it involves an intercept, a slope, and a constant term. But wait— if you don't know any coordinates on the line, how are you to calculate the slope itself? That's where **Ordinary Least Squares (OLS)** comes in.

Linear regressions are synonymous with OLS regressions. The ordinary Least Squares method is used to estimate linear regression coefficients by minimizing the sum of square residuals (SSR). We do this by adjusting coefficients to make the prediction as close as possible to reality. Now, let's dive into the adjustment process. You'll remember, that residuals are the difference between the observed (y_i) and predicted ($\widehat{y_i}$) values[3]. Mathematically, the smallest difference between any two numbers is 0, meaning they are equal. This scenario can be reflected in the equation:

$$y_i - \widehat{y_i} = 0$$

Let's replace $\widehat{y_i}$ with the regression equation we use to predict Y values, $\beta 0 + \beta 1X + \varepsilon$:

$$y_i - (\beta 0 + \beta 1X + \varepsilon) = 0$$

Recall from the normality of error distribution assumption we learned that residuals are an approximation for error terms. Therefore residuals can be estimated using the right-hand side of this error-term equation:

$$\varepsilon = y_i - (\beta 0 + \beta 1X)$$

To find the sum of all of the residuals squared, the following expression can be used:

$$\sum_{i=1}^{n} (y_i - (\beta 0 + \beta 1 x_i))^2$$

From here, we use calculus optimization - specifically partial derivatives - to find the most optimal equation for calculating $\beta 1$. The calculus behind this optimization is a bit too much to cover here, but if you're interested in the details, I've linked a great resource! For now, let's fast-forward to the solution:

[3] https://www.xlstat.com/en/solutions/features/ordinary-least-squares-regression-ols

$$\beta 1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \quad \text{and} \quad \beta 0 = \bar{y} - \beta 1 \bar{x}$$

Given these equations, here's how I would calculate $\beta 0$ and $\beta 1$ in Python manually. The code includes two methods one building a linear regression model: one uses the Scikit-learn library and the other is manually coded. The code file is documented step-by-step and also gives a better understanding of how to compute the math explained above:

```
def betaone(xtrain, ytrain):
    xbar = np.mean(xtrain)
    ybar = np.mean(ytrain)

    xycov = 0
    xsdev = 0
    for i in range(len(xtrain)):
        xdev = xtrain[i] - xbar
        ydev = ytrain[i] - ybar
        xycov += xdev * ydev
        xsdev += xdev * xdev

    return (xycov / xsdev).item()

def betazero(xtrain, ytrain, c):
    xbar = np.mean(xtrain)
    ybar = np.mean(ytrain)

    return ybar - coeff * xbar
```

Predictions

We have our coefficients, it's time for the magic to begin! From the Python code above we have realized the β values: the y-intercept is equal to 2512254.26 and the slope of the line is 425.72984. What this tells us is that the price of a house with no square footage is $ 2,512,254.26. While this might seem confusing, think of it this way: the price of just an empty lot of land with no house on it is a little over 2 and a half million dollars. With every 1 square footage increase in the house space, the price increases by roughly $425.73.

To predict the price of a house, we must use the already-established formula:

$$Y = \beta 0 + \beta 1 X + \varepsilon$$

We assume that the OLS method has minimized the error term ε to 0, and so we will use the rest of the equation $Y = \beta 0 + \beta 1 X$ to predict Y values. Iterating over each X value, we must multiply it by the coefficient and add the y-intercept value to the result. Here's how to predict all the Y values manually in Python:

```
def slr_predictions(xtest, c,i):
    ypred = []

    for x in xtest:
        y = i + c*x
        ypred.append(y)
    return ypred
```

And there you have it - we have just predicted the prices of houses using a simple linear regression.

Evaluating the Model

Now that you've built the model, how do you know if it works well? There are three key performance metrics to evaluate the model's effectiveness:

- Mean Absolute Error: calculates the average absolute residual error. In simple terms, MAE tells you how far the model's predictions are from the actual values, on average.

- Mean Squared Error: calculates the average squared residual error. Squaring the errors penalizes larger discrepancies more than smaller ones, which helps MSE highlight significant mistakes in predictions.

- Root Mean Squared Error: is the square root of the MSE metric. Because the MSE is a squared value, it is not in the same units as the predicted variable, like MAE is. Thus we take the root of MSE or RMSE to get the average magnitude of the error.

- R squared score: tells us how well the model can explain variance in the predicted variable. Also known as the coefficient of determination, the closer an R^2 value is to 1 the better the model, while a value closer to 0 suggests that the model doesn't explain much of the variance in the data.

  The R squared is calculated by using the formula: $1 - \frac{SSR}{SST}$. SSR - Sum of Squared Residuals - is the total squared error in the predictions, and SST - Total Sum of Squares - measures the total variance in the actual values. SST is calculated using the formula:

$$\sum_{i=1}^{n} (y_i - \overline{y})^2$$

By comparing SSR to SST, the R^2 score tells us how well the model explains the variation in the dependent variable.

In my code file, I've demonstrated how these 3 model evaluation metrics can be coded manually in Python.

Interpreting the Model

You're so close! You've built your model and evaluated the model, and now that we have our metrics, how do we interpret them in the context of our dataset?

1. By taking the absolute differences between predicted and actual values, MAE is not concerned with the *direction* of the residuals. A smaller MAE means your model is more accurate. But what does 'small' really mean? It's subjective and depends on the dataset. For our house price dataset, with prices ranging from $1.75 million to $13.30 million, an MAE of 1,474,748.13 feels pretty large. That means, on average, the model could be off by almost *1.5 million dollars*. Not exactly comforting if you're relying on this model to sell a house.

2. MSE calculates the average squared residual errors, and it's particularly useful for highlighting large errors. However, because it squares the residuals, MSE can be disproportionately influenced by outliers, which might make your model seem worse than it is. Our MSE score of 3,675,286,604,768 is in the trillions — yikes! To put it simply, this is a sign that tells us our model has room for improvement.

3. The RMSE is just the square root of the MSE score. This makes the error easier to interpret because it's in the same units as the original data (dollars). The RMSE score of 1,917,103.70 means that according to this metric, our model's predictions could be off by nearly 1.9 million dollars, on average. If I were the model, I'd be failing this report card.

4. Lastly, the R-squared score explains how well the model can predict the variance in the data. In our case, the R^2 score is 0.27, meaning the model can explain only 27% of the data's variance. Generally, an R-squared value over 50-60% is considered good. The closer to 1, the better; though models typically don't get much higher than 85-90%. So, while this score isn't terrible, it's clear that our model could use some more ~~studying~~ training.

The Silver Lining

Don’t be discouraged by these results. There’s plenty to learn from them! But first, let’s look at some red flags we ignored earlier. In assumption 3, we saw the dataset form a cone shape when plotted, a sure sign that the model might not be a great fit due to heteroscedasticity. I urge you to go back and play with the Heteroscedasticity Function of Fun™. By tweaking the residuals, you can see how the model's scores improve when the data is more consistent. Second, remember our error distribution had a slight skew? This tells us that we’re not as close to a perfect normal distribution as we’d like. The better our residuals match a normal distribution, the more reliable our model predictions will be. So now, you’ve learned what to watch out for!

In this post, we’ve learned:

- What linear regressions are
- The assumptions of linear regressions, and how to assess if your dataset fits the assumptions
- The math behind simple linear regressions, including key notations and expressions
- How to build a simple linear regression using the scikit-learn library in Python
- How to build a simple linear regression manually in Python
- How to predict Y values using Sklearn and manually in Python
- How to evaluate the model using MAE, MSE, RMSE, and R Squared metrics with the Sklearn library and manually in Python
- How to interpret model evaluations
- And as a bonus: How to create the graphs that help visualize this data!

Whew, that’s a lot to take in! But don’t worry — this is just the beginning. Stay tuned for a deep dive into the black box of multiple linear regression, where we’ll add more complexity by handling even more independent variables.