Module 7: Linear and Multiple Regressions

Quick Reference Guide

Learning Outcomes:

1. Differentiate between linear and nonlinear models
2. Fit a simple linear regression line using Plotly
3. Compute the squared error and absolute error
4. Recognize how loss functions react to outliers
5. Apply multiple techniques to minimize a loss function
6. Predict outcomes using a multiple linear regression model
7. Recognize ordinal, nominal, and categorical data within a dataset

Welcome to Linear Regression

One of the most important problems in machine learning, regression, can be solved using a linear model.

The regression problem

You start by actually defining the regression problem. Given a **set of features**, you want to predict a **real-valued outcome**. An example of a regression question: Given information about a person, what is their weight in pounds or kilograms?

There are two types of model:

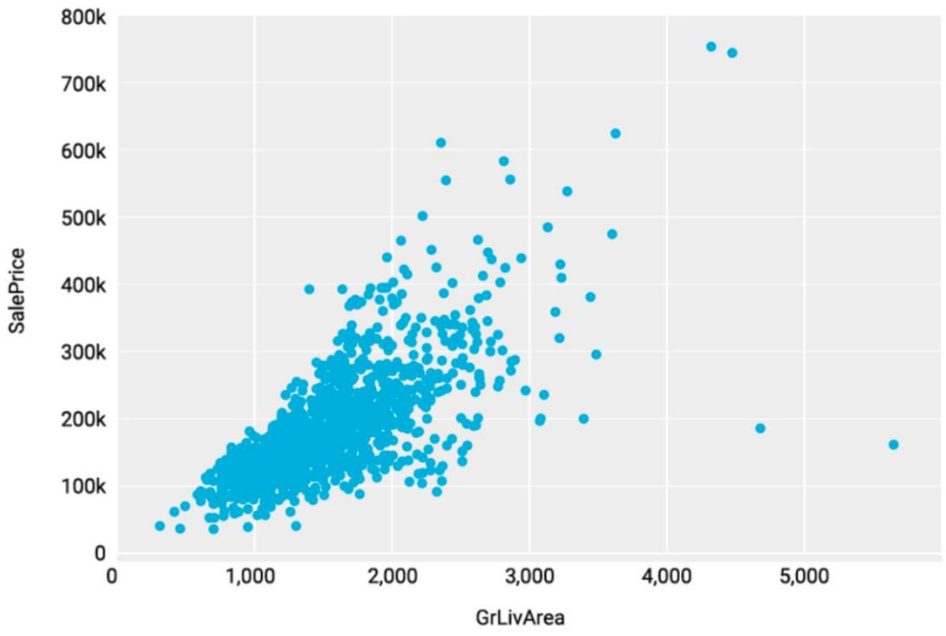
- A very **simple model** might take only one feature. Such a model is not going to give you a perfect answer.
- A more **complex model** might take multiple features, such as a numerical feature, age, and a categorical feature, sex. You could also

look at other features, like country. Such additional information can improve your model.

	Input	Output
Simple model	Singular feature For example, given someone's age, what is their weight?	Generalized answer For example, one-year-olds weigh approximately 10kg (2.2lb)
Complex model	Multiple features Given someone's age, sex, and country, what is their weight?	Additional information can improve the model

Simple regression

Simple regression has only **one explanatory variable**. Consider an example. Given the gross living area (GrLivArea), what is the sales price (SalePrice) of a house? To get a sense of how such a model might work, you can look at a scatterplot.



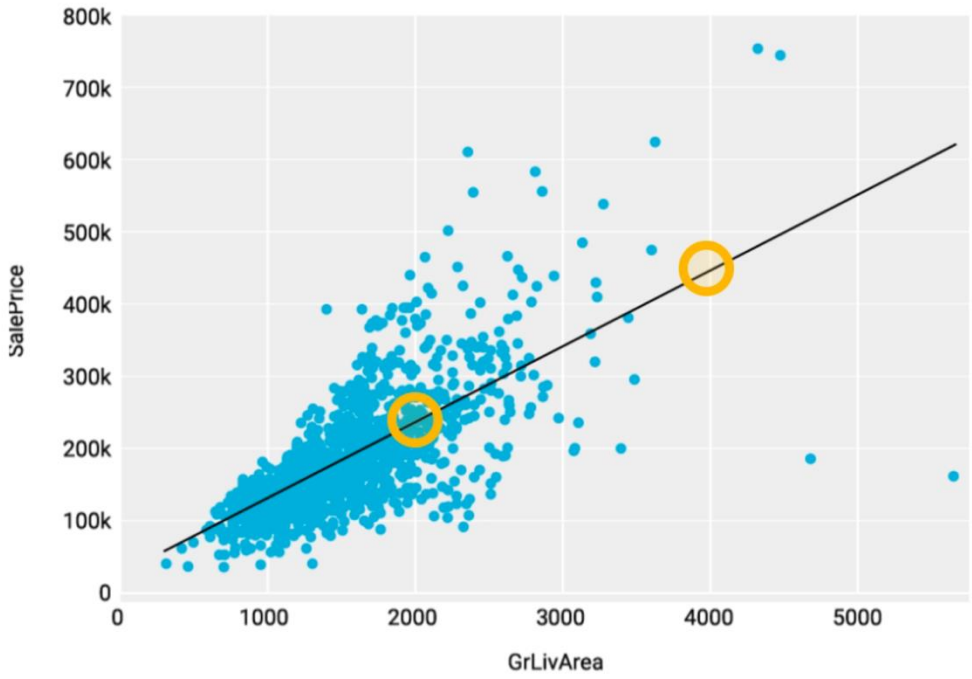
On the y-axis, you have the sale price, and that is the outcome. You want to predict this real-valued outcome. On the x-axis, you have a single feature, the gross living area. There is a clear relationship here. Living area and price are jointly distributed random variables with some significant correlation. The regression model attempts to capture this correlation as best it can.

Simple linear regression

One very simple model is the **simple linear regression model**. In such a model, you assume the relationship between the feature and the outcome is **linear**.

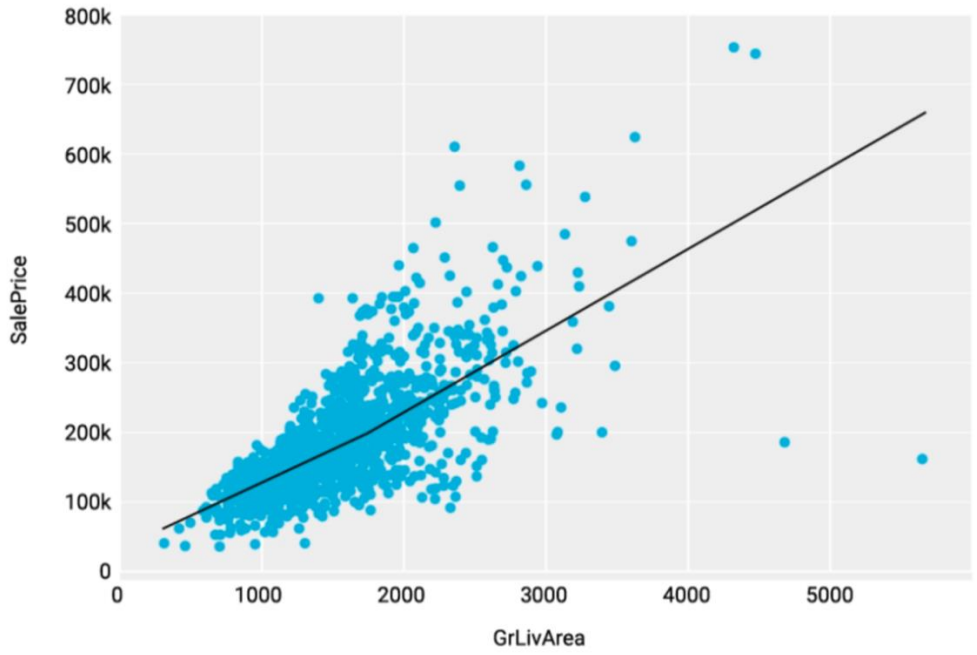
For example, according to the data in this graph, the model predicts that:

- A 2,000-square-foot house will sell for somewhere around $225,000
- A 4,000-square-foot house will sell for somewhere around $450,000
- Generally, the price of a house is approximately $100 per square foot



Regression does not necessarily have to be linear.

This is a slightly different model generated using Plotly.



This model is known as a **LOWESS model**. It has a trendline that is not exactly straight. It is not that different from the linear model, but it is **non-linear**.

Defining a model

A model is an **idealized representation** of a **system**. All models are approximations. Essentially, all models are wrong but some are useful.

Linear models are linear because the predictions they make are a **linear combination of the inputs** they receive.

Introducing the Tips Dataset

You can build a linear model to perform regression – for example, on the tips dataset. In many countries, it is common when you are done eating at a

restaurant to pay a tip or a gratuity. The tips dataset comes from the United States, where there is the expectation of a tip that is between 15% and 20%.

This specific dataset is part of the **Seaborn statistical library**.

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

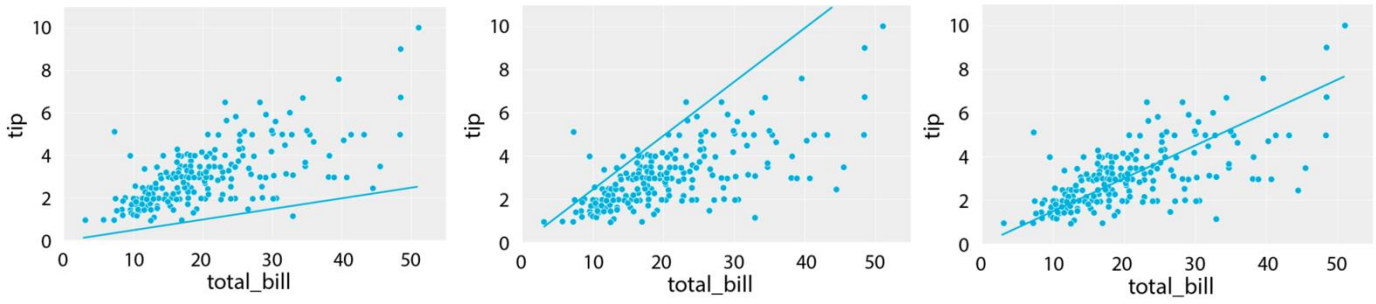
Each row of this dataframe represents one table that was served by a worker at a restaurant. For example, in the first row, at a table the total bill was $16.99 and the tip that was left was $1.01, 6% of the total bill. For this, you can use just a **single feature**, the total bill, to predict a tip.

One possible model is to assume that every party tips a fixed percentage of the total bill:

- This assumption is wrong (not perfect accuracy), but it may be useful
- This model ignores other information that might be useful

Plotting the data

Once you plot this data, there is clearly a **correlation** between the tip and the total bill:



Observations from the data:

- If you draw a line for a 5% tip, the prediction is too low
- The 25% tip model yields predictions, which are too high
- If you consider a 15% tip, this looks about right

An automated machine learning algorithm may later decide on 14.37%.

Creating a Plotly and Scikit-Learn Model

To build simple linear regression models, you can use various libraries, including **scikit-learn** and **Plotly**. Scikit-learn is very powerful, whereas Plotly is very simple.

The process of building a scikit-learn linear regression model has three steps:

1. Instantiate a linear model object
2. Fit the model
3. Use the resulting object to make predictions

Programming regression models

You can do some programming in a Jupyter notebook to read a dataset, fit a model, and make predictions with that model. The **goal** is to train an AI to try and **predict the tip column**, from the **total bill**. In this instance, you use the tips dataset from the Seaborn library:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3

3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

First, you set aside the features you want to use to predict in a variable called features:

features = data[[“total_bill”]]

Then, set aside what you want to predict:

tip = data[“tip”]

You tell the linear model library that you want a linear regression object:

f = linear_model.LinearRegression(fit_intercept = False)

When you run these four lines of code, you have set aside features and tip and created a linear regression model.

Next, you need to fit the model and tell it what you want to use to predict:

f.fit(features, tip)

Now the model is trained. But how good of a job is it really doing?

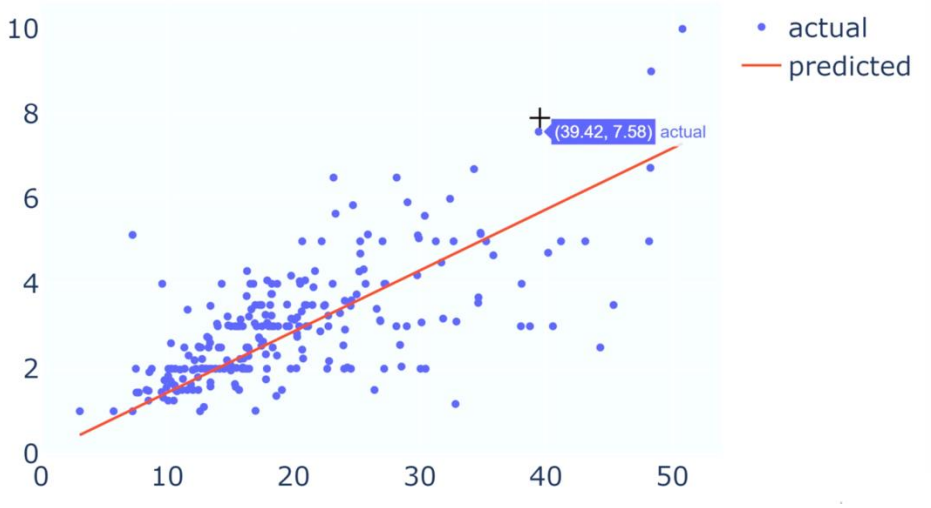
To determine this, you can create a new column in the table, prediction, to compare it against reality:

data[“prediction”] = f.predict(features)

So in addition to the true reality, the actual tip, you have what the model believes.

	total_bill	tip	sex	smoker	day	time	size	prediction
0	16.99	1.01	Female	No	Sun	Dinner	2	2.442005
1	10.34	1.66	Male	No	Sun	Dinner	3	1.486188
2	21.01	3.50	Male	No	Sun	Dinner	3	3.019807
3	23.68	3.31	Male	No	Sun	Dinner	2	3.403571
4	24.59	3.61	Female	No	Sun	Dinner	4	3.534367
...	...	...	...	...	...	...	...	...
239	29.03	5.92	Male	No	Sat	Dinner	3	4.172537
240	27.18	2.00	Female	Yes	Sat	Dinner	2	3.906633
241	22.67	2.00	Male	Yes	Sat	Dinner	2	3.258402
242	17.82	1.75	Male	No	Sat	Dinner	2	2.561302
243	18.78	3.00	Female	No	Thur	Dinner	2	2.699285

In row 240, you had a $27 total bill and the true tip was $2, but the model predicted almost $4. Which is pretty far off. But on other tables, it does a pretty good job. Now, to see this tip and these predictions on a plot, you can summarize how well the model seemed to do with a Plotly plot.



Some of the datapoints show the tip was a lot higher than was predicted. For example, where the actual tip was $7.58, the model predicted a tip just

shy of $6. The red line is what the model has learned and it does a reasonably good job at predicting tips.

What is the model exactly?

The model or the equation for a line has two key properties:

- **Intercept**: With **f.intercept = false**, you told the model to use an intercept of 0.0
- **Slope**: You can get the slope by asking for **f.coef_** and you get back 0.1437, so the slope of that line is 14.37%

What does a model look like without a zero intercept?

You can create a new model:

f_w_int = linear_model.LinearRegression(fit_intercept = True)

When you do that, and you try and make predictions, you get an error. Why? The new model has not been trained and it needs to see the data before it can make predictions. So, as before, you need to fit the model:

f_w_int.fit(features, tip)

Now when you predict a table with a total bill of $100, you get $11.42.

f_w_int.predict([[100]])

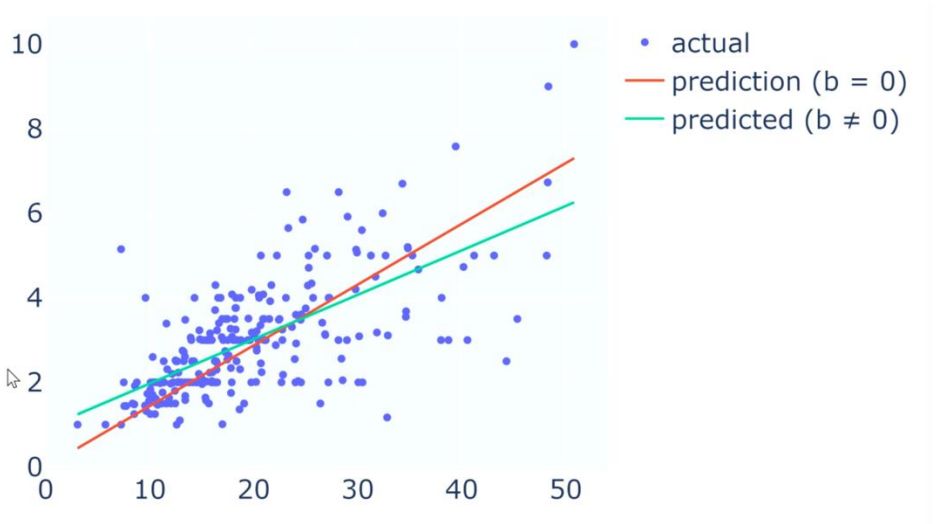
This is quite different than the other model, which would have predicted a $14.37 tip. So at least it is making different predictions.

Next, consider what happens when you give it a $0 table (a table where you bought nothing):

f_w_int.predict([[0]])

The model predicts that the guests would tip $0.92. When you ask it directly for its slope, you get 10.5%. When asked for its intercept, you get $0.92.

To understand the difference between these two models, you can plot them both on the same axis:



The red line represents the model with a zero y-intercept and the green line represents the model, which was free to pick any intercept it wanted to maximize the quality of the fit. The slope for the second model is shallower. It is only a 10% tip, compared to the 14.3%-ish tip that the other model has. But it has a non-zero intercept, meaning it starts somewhere above zero.

So Plotly is great for visualizing simple linear regression, but it is not going to generalize into models with large numbers of features.

Defining Loss

If you have different models, you need to consider **evaluating the quality** of a linear regression model.

If it is not really clear which model is better, you can resolve this conundrum by selecting a **loss function**.

Loss functions let you numerically compute the 'badness' of a prediction. A model with more loss will be considered worse. Loss functions also let you characterize the error, also known as loss, that results from a particular choice of model or model parameters.

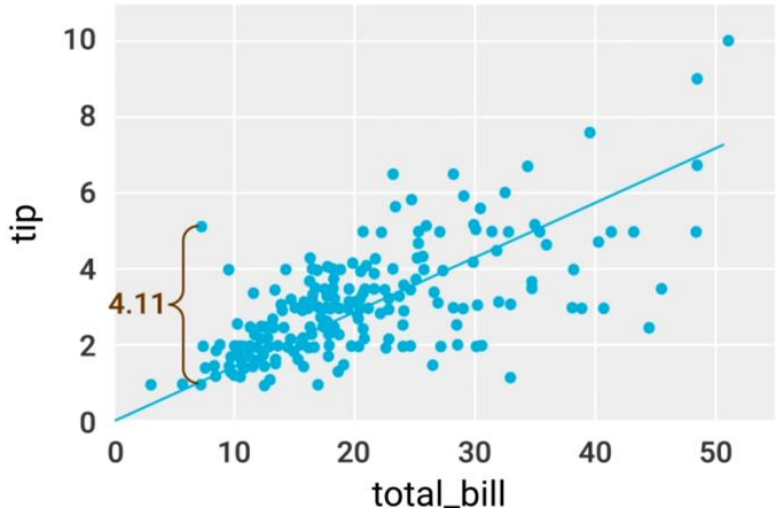
The L2 function

The **most common loss function** is L2, or squared loss. The function consists of several variables.

$$L(y, \hat{y}) = (y - \hat{y})^2$$

The letter L represents the **loss function**, y represents the **outcome** you are trying to predict, and $\hat{y}$ is the **prediction** you want to make with a model.

As an example, consider a linear model where the bill is your predictor variable, the slope is 0.1437, and the observed tip is y.



As one of its predictions, the model estimates that if the bill is \$7.25, then the tip will be \$1.04. But the true tip for this particular table was \$5.15.

How do you compute the loss for this particular misprediction? Well, y and $\hat{y}$ are \$5.15 and \$1.04 respectively. The squared loss for this prediction is the difference between those two numbers, which is $(4.11)^2$ or 16.89.

What about the loss over the whole dataset? You may want to know how a model performs, not just on one observation, but on everything in the dataset. One common approach is to compute the average loss over all of the points. This is known as the **mean squared error (MSE)**. You can think of the mean squared error as the average of the square of the distance of each datapoint from your model's regression line.

Computing L2 Loss

You can use pandas to compute the mean squared error on your L2 losses. So in this table, you have the total bill, tip, and predictions.

	total_bill	tip	sex	smoker	day	time	size	prediction
0	16.99	1.01	Female	No	Sun	Dinner	2	2.442005
1	10.34	1.66	Male	No	Sun	Dinner	3	1.486188
2	21.01	3.50	Male	No	Sun	Dinner	3	3.019807
3	23.68	3.31	Male	No	Sun	Dinner	2	3.403571
4	24.59	3.61	Female	No	Sun	Dinner	4	3.534367

You can create a temporary column, such as **l2_loss**, for the L2 losses of each prediction.

	total_bill	tip	sex	smoker	day	time	size	prediction	l2_loss
0	16.99	1.01	Female	No	Sun	Dinner	2	2.442005	2.050638
1	10.34	1.66	Male	No	Sun	Dinner	3	1.486188	0.030211
2	21.01	3.50	Male	No	Sun	Dinner	3	3.019807	0.230585
3	23.68	3.31	Male	No	Sun	Dinner	2	3.403571	0.008756
4	24.59	3.61	Female	No	Sun	Dinner	4	3.534367	0.005720

To do this, you take the tip and the prediction, compute their difference, and then simply square it.

data["l2_loss"] = (data["tip"] – data["prediction"]) ** 2
data.head(5)

You can draw some conclusions based on the tip and prediction:

- When the tip and prediction are really close, the L2 loss is small
- When the tip is far off from the prediction, the L2 loss is big

Next, you can compute the mean squared error. There are a few different ways to do this. One of them is using a part of the NumPy library.

np.mean(data["l2_loss"])

You get back an average L2 loss of 1.178. You can also do this natively in pandas, with the same outcome.

data["l2_loss"].mean()

Typically, to calculate the mean squared error in practice, you would use the mean squared error function provided by scikit-learn metrics. To do that, you simply ask it to compute the mean squared error between two arrays. In this case, one array is data tip, and the other is the prediction of your model.

from sklearn.metrics import mean_squared_error
mean_squared_error(data["tip"], f.predict(data[["total_bill"]]))

All three of these different computations give the exact same mean squared error value.

Optimizing L2 Loss

Once you know how to use the black box scikit-learn linear regression library to create an AI model from data, you can explore the process by which this model gets trained. You do this by focusing on the fact that the mean squared error is simply a function of your choice of slope, or θ.

Consider this table, which has the total bill, the prediction, and the L2 loss:

	total_bill	tip	sex	smoker	day	time	size	prediction	l2_loss
0	16.99	1.01	Female	No	Sun	Dinner	2	2.442005	2.050638
1	10.34	1.66	Male	No	Sun	Dinner	3	1.486188	0.030211
2	21.01	3.50	Male	No	Sun	Dinner	3	3.019807	0.230585
3	23.68	3.31	Male	No	Sun	Dinner	2	3.403571	0.008756
4	24.59	3.61	Female	No	Sun	Dinner	4	3.534367	0.005720
...	...	...	...	...	...	...	...	...	...
239	29.03	5.92	Male	No	Sat	Dinner	3	4.172537	3.053627
240	27.18	2.00	Female	Yes	Sat	Dinner	2	3.906633	3.635249
241	22.67	2.00	Male	Yes	Sat	Dinner	2	3.258402	1.583576
242	17.82	1.75	Male	No	Sat	Dinner	2	2.561302	0.658212
243	18.78	3.00	Female	No	Thur	Dinner	2	2.699285	0.090430

The goal is to find the choice of θ, where the mean squared error of the L2 loss values is minimized. You can also compute the L2 loss of that resulting prediction by taking your prediction, subtracting out the tip, and then squaring. This gives you some insight into how you could train an AI model.

When training an AI model, scikit-learn's process involves trying out different values of θ until it has minimized the loss as much as possible.

Simplifying the process for optimizing L2 loss

You can make the process quicker by doing it in one line.

mean_squared_error(data["prediction"], data["tip"])

So, take the total bill and multiply it by whatever θ you choose. You can try it out for various values like an 8% tip, 20% tip, and 30% tip, to have three different choices.

mean_squared_error(data["total_bill"] * 0.2, data["tip"])

The mean squared error for a 20% tip is 2.667486278688525.

mean_squared_error(data["total_bill"] * 0.08, data["tip"])

The mean squared error for an 8% tip is 3.08881276852459.

mean_squared_error(data["total_bill"] * 0.3, data["tip"])

The mean squared error for a 30% tip is 12.66543732377049.

These are the mean squared errors for these three models. If you are using L2 loss to evaluate the quality of models, the mean squared error for the 20% tip model is the best.

How do you find the best model?

One technique to find the best model is trying a variety of values of θ to find the one with the lowest mean squared error. To make it clearer, imagine

creating a function called **mse_given_theta()**. You give it a slope, which will give your parameter θ. It is going to return the mean squared error of the total bill times θ against the tip. That is your prediction.

```
def mse_given_theta(theta):
    return mean_squared_error(data["total_bill"] * theta, data["tip"])
```

In other words, this single function call lets you elaborate or explore the different space of θ. This will be the key that you use to optimize your θ.

Next, you are going to use the **mse_given_theta()** function to find the best slope. In this instance, to create a list of 100 θs between 0.1 and 0.2:

```
thetas = np.linspace(0.1, 0.2, 100)
```

You can compute the MSEs for these θs:

```
mses = [mse_given_theta(theta) for theta in thetas]
```

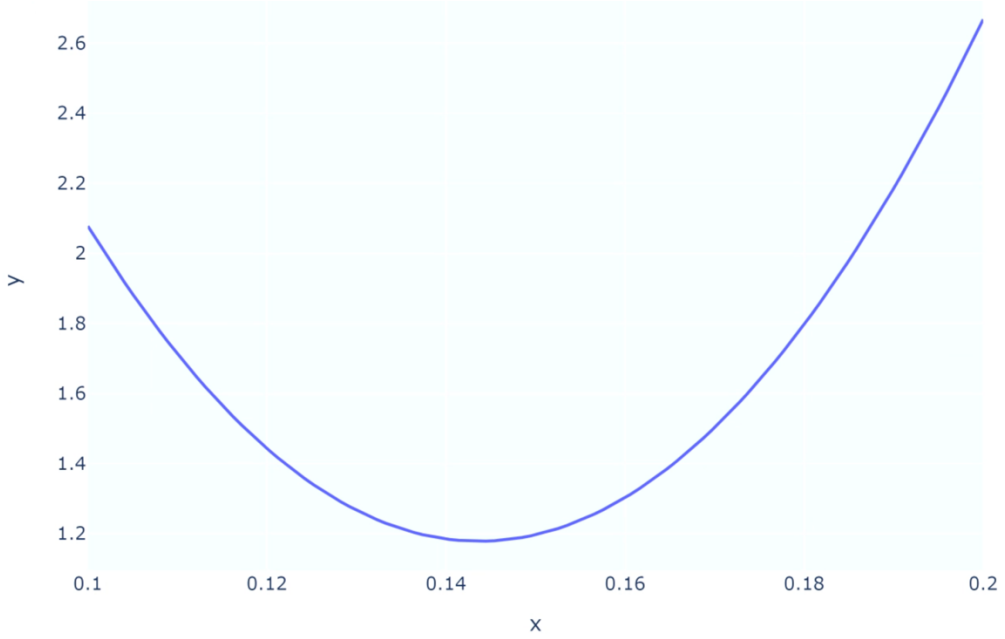
When you run this line of code, you get back mean squared errors in a list of values that initially decrease in value and then start to increase again. This tells you that the right data is in the middle of the list.

Plotting data in Plotly

The best way to find the best θ is to make a plot using the following syntax:

```
px.line(x = thetas, y = mses)
```

You get back a plot, where x is the θ and y is the mean squared error.



Now you have a plot showing that your choice of tip percentage gives you back the mean squared error. The minimum mean squared error happens right where your scikit-learn model picked θ.

Optimizing the mean squared error

In terms of math notation, reflect on how you found the optimal θ.

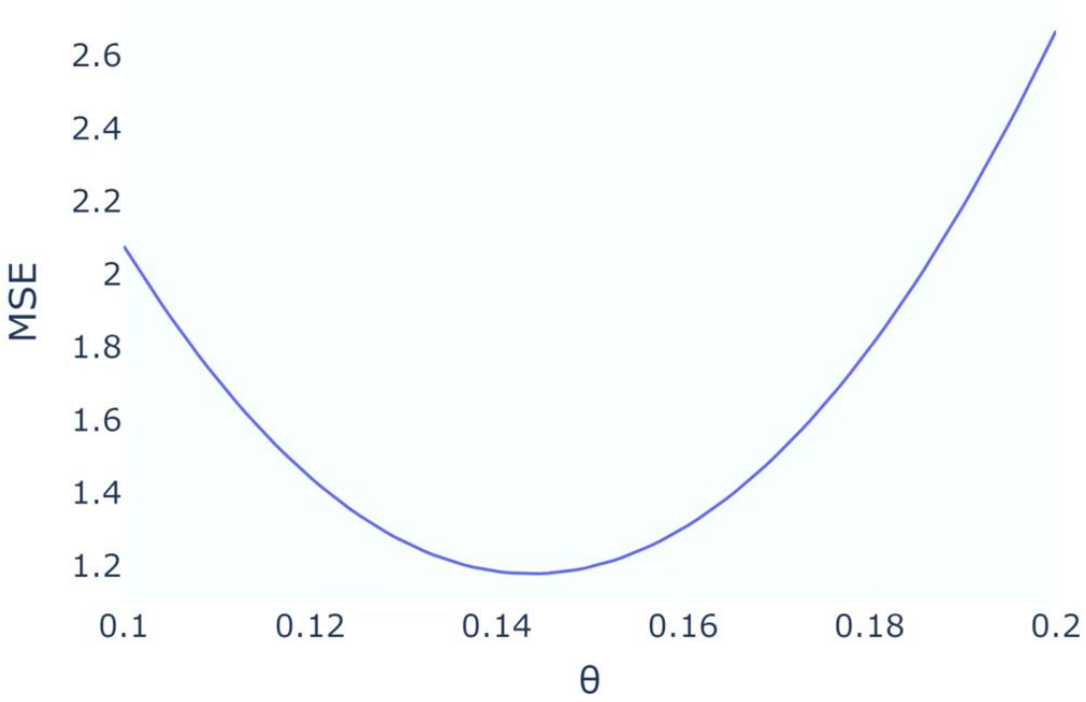
Formally, the model is given as $\hat{y}_i = \theta x_i$:

- $\hat{y}_i$ is the i^{th} tip prediction
- x_i is the i^{th} total bill (the input data)
- θ is the parameter of the system (the slope)

The x and y values are given in advance and the choice of θ is arbitrary. For each choice of θ, the loss function returns a value that evaluates the model on the data that is used.

Using SciPy Optimize to Optimize L2 Loss

So, you have seen how to use a visual plot of the mean squared error versus θ to find the optimizing value. And in this case θ is around 0.144.



Next, consider how scikit-learn does this. It uses the function **scipy.optimize.minimize()**. This is an arbitrary function minimization library.

For example, you can say:

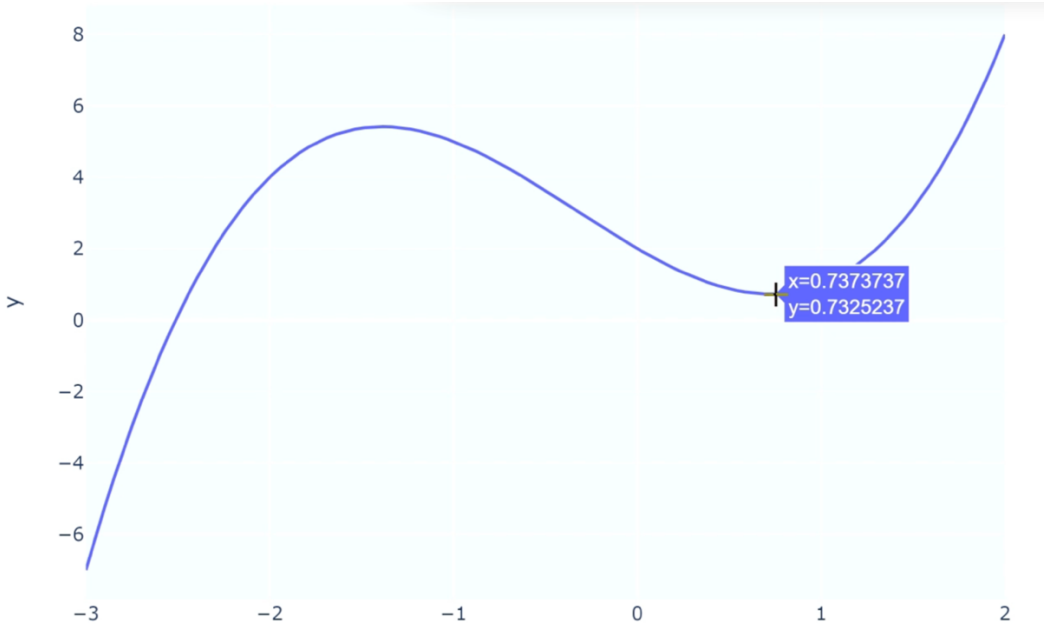
```
def g(x):
    return x**3 + x**2 – 3*x + 2
```

That is just some arbitrary function that scipy.optimize will be able to minimize. The scipy.optimized library does this automagically. You can give it the function, g, and a place to start, 1,000.

```
scipy.optimize.minimize(g, x0 = 1000)
```

It will come up with 0.72. It believes that the x which minimizes g is 0.72.

You can see how it did, by plotting g.



When the function is plotted, you see that the minimizing value is around 0.72. The mean squared error function is also a function that you could minimize. So if you say **scipy.optimize.minimize**, give it **(mse_given_theta)**, and a starting value of x0 = 0.2, it will find an optimal value of 0.1437. This is the same value that you got straight out of your scikit-learn model.

Absolute and Huber Loss

Before, the loss function that you used was the mean squared error or L2 loss. However, it is not the only loss function you can use.

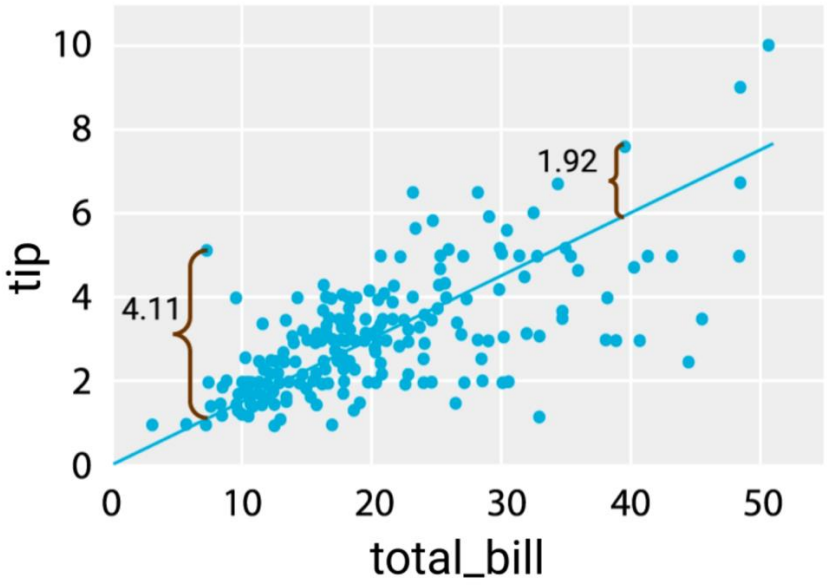
Another common loss function is the **L1 or absolute loss**:

$$L(\mathcal{D}) = \frac{1}{n}\sum_i^n L(y_i, \hat{y}_i)$$

It is defined in almost the exact same way as squared loss, but rather than squaring the difference between your prediction and the true outcome, you take the absolute value of that difference.

Mean absolute error (MAE)

Just like the mean squared error across an entire dataset, you can also define the overall loss across an entire dataset by defining the **mean absolute error**. The choice of loss function matters. Different loss functions are going to give you different parameters.



In the tips dataset, for example:

- 14.37% does not optimize the MAE, as the MAE is 0.778
- 15% yields a slightly lower error, as the MAE is 0.770

Why is the optimal theta different?

First, consider how the two loss functions handle **outliers**. An outlier is a point that is much different than a typical point drawn from its distribution.

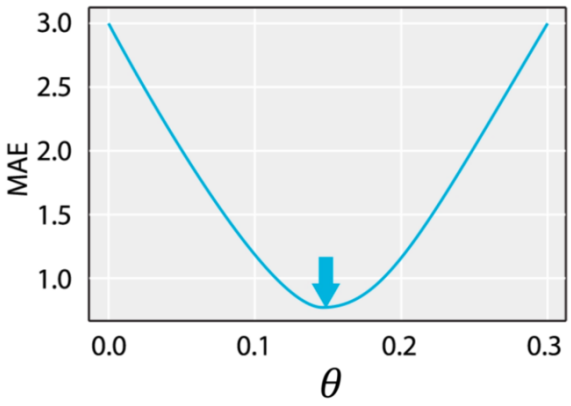
Looking at the graph:

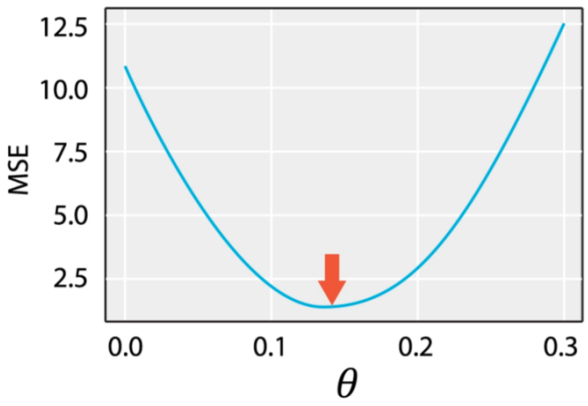
- L2 loss is 16.89
- L1 loss is only 4.11

The L2 loss gives a much higher penalty to outliers than the L1 or absolute loss. So as a datapoint gets further and further away from the prediction, you are going to get an error that increases with the square of that distance. Thus, you can say that since the L2 loss is much more strongly affected by outliers, the optimal parameters for the mean squared error (MSE) are also going to be more affected by outliers.

Does that mean that MSE is worse? No, both the MSE and MAE can each be better in different circumstances and, in practice, the difference is not necessarily that large.

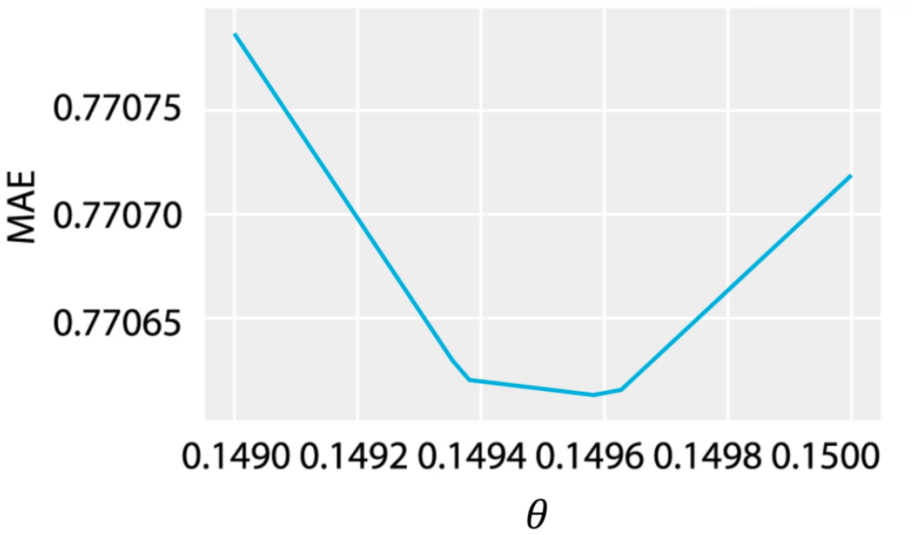
Now you can also understand the difference between these loss functions by just looking at the shape of the MSE and MAE on the tips dataset.





On the left, the MAE is a function of θ. On the right, note the MSE. The bottom of the MAE is just slightly to the left of the bottom of that MSE curve.

One potential issue you might run into if you select the MAE, is that it is **piecewise linear**. In other words, it consists of a bunch of straight line segments.



Numerical methods sometimes have a hard time optimizing functions that have this sort of shape.

Multiple Linear Regression

Multiple linear regression uses **multiple features** to make predictions. So for example, you can make predictions using data from multiple columns.

	total_bill	tip	sex	smoker	day	time	size	prediction
0	16.99	1.01	Female	No	Sun	Dinner	2	2.442005
1	10.34	1.66	Male	No	Sun	Dinner	3	1.486188
2	21.01	3.50	Male	No	Sun	Dinner	3	3.019807
3	23.68	3.31	Male	No	Sun	Dinner	2	3.403571
4	24.59	3.61	Female	No	Sun	Dinner	4	3.534367
...	...	...	...	...	...	...	...	...

In this instance, you can use the two **numeric** features, total bill and size.

```
features = data[["total_bill", "size"]]
tip = data["tip"]
f2 = linear_model.LinearRegression(fit_intercept = False)
f2.fit(features, tip)
```

When you fit the model, you get back a linear regression model. This is a very user-friendly library that will take any dimensionality of data that you throw at it.

Once you have created this f2 model, you can ask for its coefficients using **f2.coef_**. The coefficients for this model are 0.10 and point 0.36. Now you have a coefficient that goes with the total bill and a coefficient that goes with the size.

You can use this model to make predictions. Say, if you want to make a prediction for a table with a $10 total bill and three people seated:

```
f2.predict([[10, 3]])
```

So 10 is the argument for total bill and 3 is the argument for size. The prediction is $2.09.

The original model does not know how to use the number of people seated:

```
f.predict([[10]])
```

Its prediction will be just $1.43. So these two different models provide different predictions.

Now one thing to note: The f2 model was trained on two-dimensional data and so it can only make predictions on two-dimensional data. So if it was trained on both features, you must give it both features in order to make a prediction. That is also true for a one-dimensional model. You cannot add information it does not know about. With any linear models, if you have *k* parameters, then you need *k* features.

Comparing predictions

Next, you can compare the predictions side-by-side. The prediction column in your table is the prediction from the one-dimensional model as trained on total bill. Now you will add a second column, which is the prediction of your new model that uses both features, total bill and size.

data["prediction"] = f.predict(data[["total_bill"]])
data["prediction_2d"] = f2.predict(data[["total_bill", "size"]])

For each of the rows in the table, it goes through and computes a prediction in a column called prediction_2d.

	total_bill	tip	sex	smoker	day	time	size	prediction	prediction_2d
0	16.99	1.01	Female	No	Sun	Dinner	2	2.442005	2.435290
1	10.34	1.66	Male	No	Sun	Dinner	3	1.486188	2.127653
2	21.01	3.50	Male	No	Sun	Dinner	3	3.019807	3.202249
3	23.68	3.31	Male	No	Sun	Dinner	2	3.403571	3.109052
4	24.59	3.61	Female	No	Sun	Dinner	4	3.534367	3.924894
...	...	...	...	...	...	...	...	...	...

When exploring the results of these two models side-by-side, some predictions are very close, but some are very different.

How do you know which model is better? You could use loss functions to compute the mean squared error:

mean_squared_error(data[“prediction”], data["tip"])

The mean squared error between the prediction of your 1D model and the tip is 1.178.

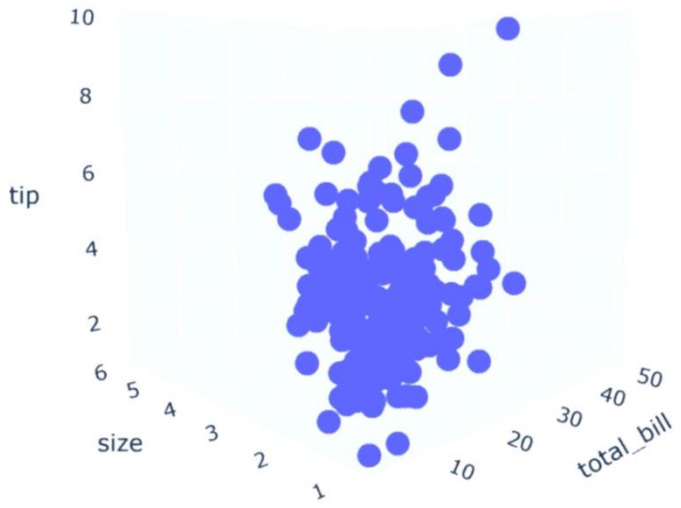
mean_squared_error(data[“prediction_2d”], data["tip"])

The prediction for the 2D model is 1.065. What does that tell you? Well, recall that minimizing the mean squared error is the definition of quality. So that tells you the 2D model is better, in the sense that it gets a lower mean squared error. So the machine learning model gave higher-quality predictions on this dataset when you gave it more information.

To understand this other model, you need to think in three dimensions:

px.scatter_3d(data, x = “total_bill”, y = “size”, z = “tip”)

So this is a 3D plot of the raw data:

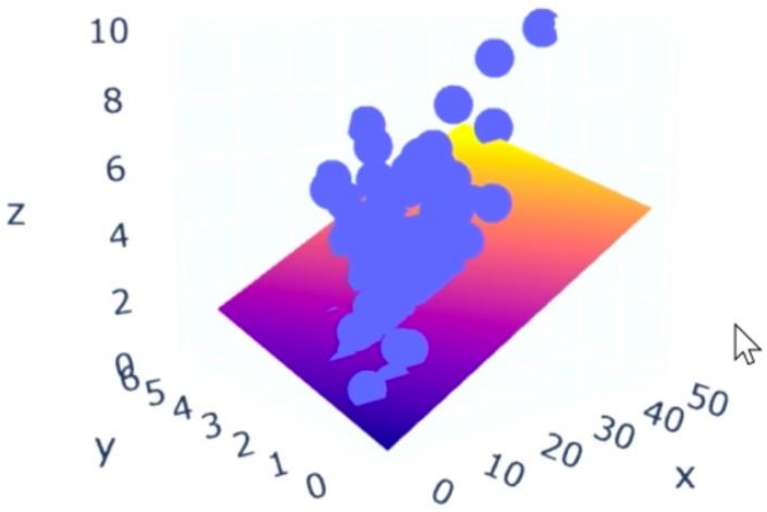


Looking at this 3D plot, you can tell that:

- As the total bill goes up, the tip goes up
- As the size goes up, the tip goes up

The linear regression model is going to fit something to this.

This could be illustrated using a meshgrid of x, y values with the model plotted on top of the data.



Again, you have the total bill on the x-axis, size on the y-axis, and tip on the z-axis. But instead of the model being a one-dimensional line, it is a two-dimensional plane.

Looking at the 2D plot, you can tell that:

- The plane increases with both the size and the tip
- The model is finding the plane of best fit

1D and 2D as equations

Consider what the actual equations are that represent the output of the model. Recall that f's coefficients are 0.1437 and f2's are 0.1 and 0.36.

So if you think of your models as equations:

- Model one says the tip is 0.14 times the bill
- Model two says the tip is 0.1 times the bill, plus an extra $0.36 for every person seated at the table

Although model two has a lower mean squared error, model one is probably a better model of the actual reality. What is happening here is that model two is **overfitting** in some sense. So keep in mind that the model is not just as good as the mean squared error it produces, but also needs to make some sense.

Using Non-Numeric Features

Next, consider the problem of using non-numeric features.

total_bill	tip	sex	smoker	day	time	size
28.97		Male	Yes	Fri	Dinner	2
17.81		Male	No	Sat	Dinner	4
13.37		Male	No	Sat	Dinner	2
15.69		Male	Yes	Sun	Dinner	2
15.48		Male	Yes	Thur	Dinner	2

For each table, you know not just the bill amount and the size, but also the sex of the pair, whether they were a smoker or not, the day of the week, and the time of day.

So you now want to find a way to use all of that other information to inform the prediction of a tip.

One problem is that your definition of a linear model is the **weighted sum of the features of a given sample**. There is no way for the model to do something like; if it is Friday, then use this θ, but if it is Saturday, then use this other θ.

In fact, linear models can:

- Use non-numeric features
- Capture non-linear behavior

Going back to the table, you use the following equation to determine the tip:

$$\hat{y} = \theta_1 \times bill + \theta_2 \times size + \theta_3 \times day$$

Now at first glance, it is not totally clear how you would use, for example, the day of the week. A naive approach might be to decide that Thursday is zero, Friday is one, Saturday is two, and Sunday is three. But that does not really make sense, because it implies that Sunday is somehow three times as large as Friday. Instead, there are **two different approaches** that will both work much better than this naive approach.

One-hot encoding

In this **first approach**, you create a new set of K features, where K is the number of unique values for the non-numeric feature of interest. So if you have four different days of the week that appear, K is four. These features are sometimes called **dummy features**. For a given observation, then, exactly one of your four dummy features will be 1 and the rest will be 0.

total_bill	size	day	Thur	Fri	Sat	Sun
15.48	2	Thur	1	0	0	0
28.97	2	Fri	0	1	0	0
17.81	4	Sat	0	0	1	0

13.37	2	Sat	0	0	1	0
15.69	2	Sun	0	0	0	1

For this, you can create a new Thursday feature, which is 1, and a new Friday, Saturday, and Sunday feature – all of which are 0. Since only one of these dummy features can be 1, this idea is often called one-hot encoding because only one of the columns is hot, or 1, at any given time.

Creating dummy features in pandas

Recall loading the tips dataset in its original form:

data = sns.load_dataset("tips")
data.head(5)

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

So now you can add those dummy variables. But before you do, you can focus only on the columns that you care about – the total_bill, size, and day – by creating a new dataframe that only has these three features.

three_features = ['total_bill', 'size', 'day']
three_feature_data = pd.DataFrame(data[three_features])
three_feature_data.iloc[[193, 90, 25, 26, 190], :]

So if you do this, you get back just the total bill, the size, and the day.

	total_bill	size	day
193	15.48	2	Thur
90	28.97	2	Fri
25	17.81	4	Sat
26	13.37	2	Sat
190	15.69	2	Sun

You may notice they are coming back in a different order. That is because you used the **iloc** command to include all four days in the table.

Next, you can use the following syntax to create the dummy variables:

```
dummies = pd.get_dummies(three_feature_data['day'])
dummies.iloc[[193, 90, 25, 26, 190], :]
```

The dummies table that comes back from pandas has one new column for every possible value of the day value that you provided.

	Thur	Fri	Sat	Sun
193	1	0	0	0
90	0	1	0	0
25	0	0	1	0
26	0	0	1	0
190	0	0	0	1

If you want to use this for linear regression, you need to combine these two tables. To do that, you can use a new piece of syntax called **pd.concat** that can either add rows or columns to a dataframe:

```
data_w_dummies = pd.concat([three_feature_data, dummies], axis=1)
data_w_dummies.iloc[[193, 90, 25, 26, 190], :]
```

To add columns, you can use the argument **axis=1**. The other argument is just the two tables that you wanted to combine.

	total_bill	size	day	Thur	Fri	Sat	Sun
193	15.48	2	Thur	1	0	0	0
90	28.97	2	Fri	0	1	0	0
25	17.81	4	Sat	0	0	1	0
26	13.37	2	Sat	0	0	1	0
190	15.69	2	Sun	0	0	0	1

Then you can just fit it with the whole table:

f_with_day = linear_model.LinearRegression(fit_intercept=False)
f_with_day.fit(data_w_dummies, tip)

Once you have done so, you have a model that has been trained, not on just these two numeric features, but also the non-numeric feature – the day. It yields six coefficients: 0.09299361, 0.18713231, 0.66829361, 0.74578683, 0.62112858, and 0.73228865.

The resulting model is a six-dimensional model. In other words, each observation has six features. The model can be represented as:

$$\hat{y} = \theta_1\phi_1 + \theta_2\phi_2 + \theta_3\phi_3 + \theta_4\phi_4 + \theta_5\phi_5 + \theta_6\phi_6$$

For each of these six features there is a corresponding weight:

- θ_1 = 0.093: How much to weight the total bill
- θ_2 = 0.187: How much to weight the party size
- θ_3= 0.668: How much to weight the fact that it is Thursday
- θ_4= 0.746: How much to weight the fact that it is Friday
- θ_5= 0.621: How much to weight the fact that it is Saturday
- θ_6= 0.732: How much to weight the fact that it is Sunday

These θs tell you how much each of your six features affect the prediction of your model.

What did AI learn about human behavior and tipping?

AI thinks that the way a group of dining humans compute a tip is to start with a basic tip value, depending on the day, throw in another $0.19 for each person at the table, and then tip 9.3% on top of the value they have so far. Why is AI coming up with this really convoluted rule? Well, it is because of the chosen loss function.

If you compute the mean squared error for this six-dimensional model:

- It does slightly better than the 2D and 1D models
- Due to overfitting, it might perform worse on new observations

An alternate approach for using non-numeric data

In this approach, rather than building a model with a fixed offset in cents for each day, you can build a different model for each day. To keep things simple, assume you are using only the total bill and day. In that case, you would have a different slope and, if you choose to have one, a different y-intercept each day.

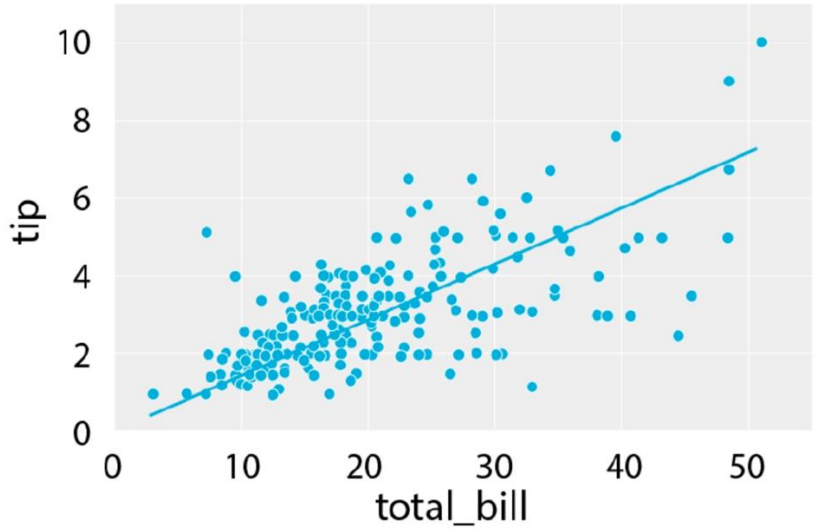
How do you do this?

- Create different dataframes for each day
- Use scikit-learn to train the four separate models

However, this can be done in Plotly very simply, without making separate dataframes, by setting the **color** argument.

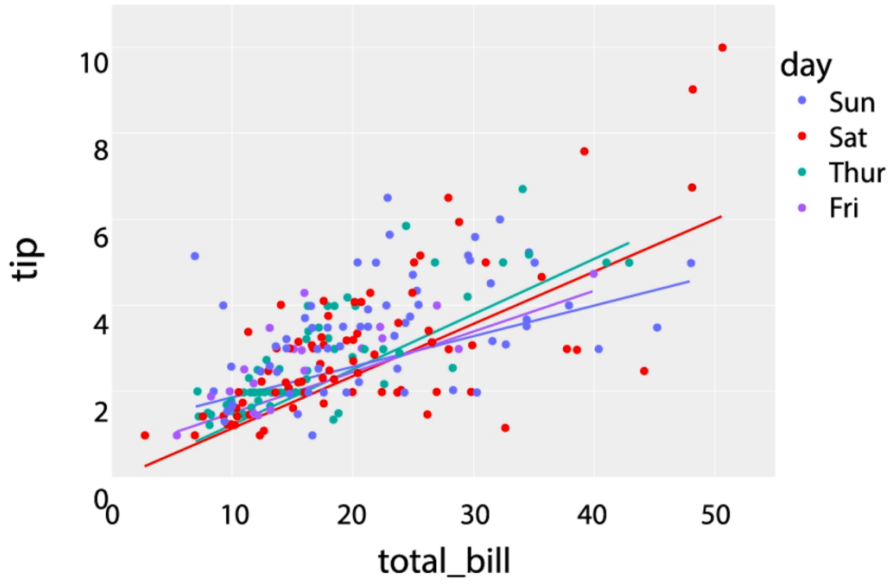
First, remember that simple linear regression looks like this in Plotly:

px.scatter(data, x="total_bill", y="tip", trendline = "ols")



But if you add another parameter and say **color = "day"**, now you get four different trendlines:

px.scatter(data, x="total_bill", y="tip", color = "day", trendline = "ols")



Each of those trendlines has its own slope and y-intercept.

The Importance of Linear Regression

Linear regression is really the building block for most applied statistics and data science applications. If the goal is to produce the most accurate forecast, or prediction, linear regression can get you very close to what the fanciest machine learning methods can produce.

An example

Consider a practical example, where you use linear regression to forecast retail sales at some candidate business locations.

In this example, you are a manager of Peet's Coffee, which is a coffee chain that was founded in Berkeley, California in 1966. You are trying to decide where to put your next store. Fortunately for you, Peet's already has over 200 retail stores in the United States and enterprise software, which tracks the annual sales from these stores. You are interested in building a model with this data that relates sales of the existing Peet's stores to local characteristics of this property where the store is located.

What do you think might matter for sales in a coffee shop?

- Foot traffic
- Local income
- Education
- Nearby competitors
- Weather
- Parking

Consider what you would do if you were a new entrant into the market entirely and you did not have a database full of sales at your other stores.

Where would other stores' sales data come from?

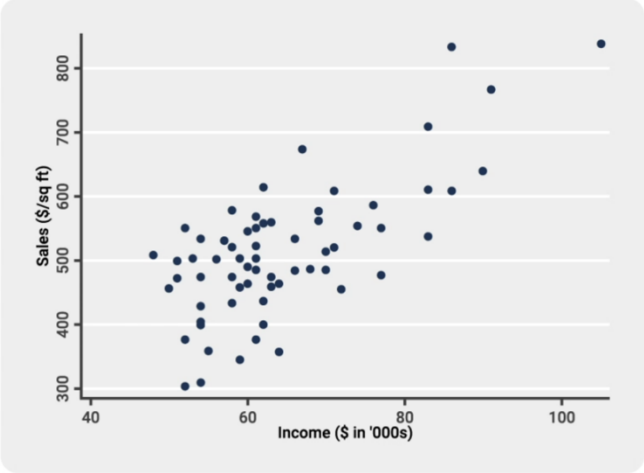
- Survey research firms

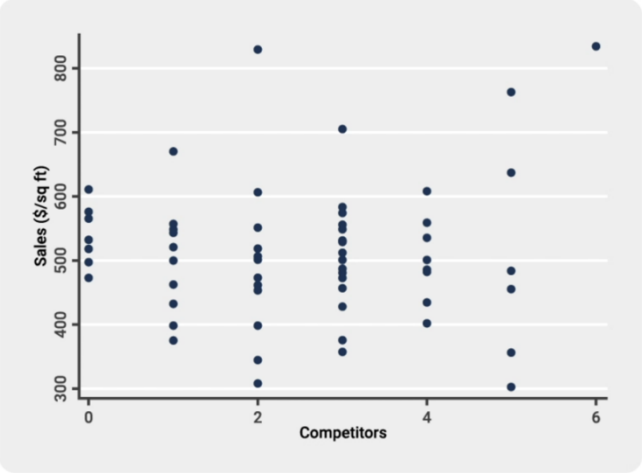
Where does the other data come from?

- Survey research firms
- Publicly-available information
- Collected data

Say you have all of the data that you need on sales and the data on what you might think would be predictors of sales. Where do you go from here? Well, the first thing to do is ensure that the data you want is the data that you have. Secondly, plotting the data is incredibly helpful and often revealing in terms of outliers and other data features that may not be obvious. It is often useful, especially in cases where there are not tons of predictors or covariates.

Consider these two-way scatter plots comparing two variables.





The scatter plot on the left shows a **positive correlation between the neighborhood average income and the sales** at the retail stores in the data.

The scatter plot on the right suggests a **weak and noisy relationship between the number of nearby competitors and sales**. Is that surprising? Do you think competition is good or bad for business? What else might be going on here?

Using multiple regression

You can turn to your regression model to try to learn more. You can use multiple regression to put all the pieces together and create a linear model to forecast retail sales using the following variables.

Term		Estimate	Std Error	*t*-Statistic	*p*-Value
Intercept	b_0	60.359	49.290	1.22	0.2254
Income	b_1	7.966	0.838	9.50	< .0001
Competitors	b_2	–24.165	6.390	–3.78	0.0004

In this instance, you use neighborhood income, measured in thousands of dollars, and the number of nearby stores or competitors.

So most regression software will spit out the following set of results:

Estimated Sales = 60.36 Intercept + (7.97) Income – (24.17) Competitors

In this case, sales is denoted in dollars per square foot. Thus, you can imagine the coefficients on each of these explanatory variables is being interpreted in the following way. A one-unit increase in the explanatory variable, or independent variable, corresponds to a b or the coefficient unit increase in the outcome variable. In this case, the coefficient on competitors is −24. So an additional competitor within one mile of this store is associated with a $24 decrease in retail sales per square foot, holding local income fixed.

Recall the scatter plot showing a weak relationship between the number of competitors and sales. Now this coefficient is suggesting a negative relationship.

What is going on?

- One possible answer is that the neighborhood income and competitors are correlated with one another so as income increases, so do the number of competitors
- By holding income fixed in this multiple regression model, you can look at the independent effect of competitors that has been purged of this possible confounding

Turning to the income coefficient, the coefficient on income suggests a one-unit increase in income, in this case the unit is $1000.

Term		Estimate	Std Error	*t*-Statistic	*p*-Value
Intercept	b_0	60.359	49.290	1.22	0.2254
Income	b_1	7.966	0.838	9.50	< .0001
Competitors	b_2	–24.165	6.390	–3.78	0.0004

So a $1,000 increase in average local income is associated with a $7.90 increase in retail sales per square foot, holding the number of competitors fixed. The intercept in this case measures the average forecasted sales in areas where both explanatory variables are equal to zero. And in this particular case, it is not very informative. Why would you rent a retail space in a neighborhood where the average income is $0?

You can also look at the ***t*-statistics** and ***p*-values** corresponding to each explanatory variable, which provide you with the test statistics from the null hypothesis that each of these coefficients is equal to zero. The *p*-values for each explanatory variable are small enough to reject the null hypothesis at

conventional levels and say that these are statistically significant relationships.

You can also look at the overall predictive capacity of this model and how much of the variation in the dependent variable – in this case, sales – is explained by just these two predictor- or independent variables.

This is given by the R^2 in the model, which is 0.59 in this case. Thus nearly 60% of the variation in retail sales in all of the Peet's coffee shops in your data can be explained by just these two variables.

Now how do you use this model to generate a forecast for a few different sites that you are considering? You simply figure out the number of nearby competitors at a prospective site and the average household income at that site and plug it into this formula here:

Estimated Sales = 60.36 Intercept + (7.97) Income – (24.17) Competitors

That will give you the best prediction of retail sales at any given future location.

Now there are two things you want to be careful about when forming predictions using regression models to forecast. One, is that you want to make sure that the forecast is not too far out of sample. What that means is if all of the data that you are using to form predictive models only come from neighborhoods with one or two competitor coffee shops, you would be very reluctant to then try to extrapolate those conclusions to areas with 50 coffee shops. Two, make sure that you are not forecasting too far outside of the range of the data that the model is being built on.