lab05

July 5, 2022

[ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook("lab05.ipynb")
```

1 Lab 5: Modeling, Loss Functions, and Summary Statistics

In this assignment, you will be performing modeling on a dataset containing restaurant tips, exploring loss functions and summary statistics in the process.

Due Date: Saturday, July 9, 11:59 PM PT.

1.0.1 Collaboration Policy

Data science is a collaborative activity. While you may talk to others about the labs, we ask that you **write your solutions individually**. If you do discuss the assignments with others, please **include their names** in the following cell:

Collaborators: *List names here*

1.1 Predicting Restaurant Tips

In this lab, you will try to predict restaurant tips from a set of data in several ways:

A. Without given any additional information, use a **constant model with L2 loss** to predict the tip $\hat{y}$ as a summary statistic, θ.

B. Given one piece of information—the total bill x—**use a linear model with L2 loss** to predict the tip $\hat{y}$ as a linear function of x.

C. See if a **constant model with L1 loss** changes our predictions.

First, let's load in the data.

[1]:
```
# just run this cell
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
sns.set()
sns.set_context("talk")
```

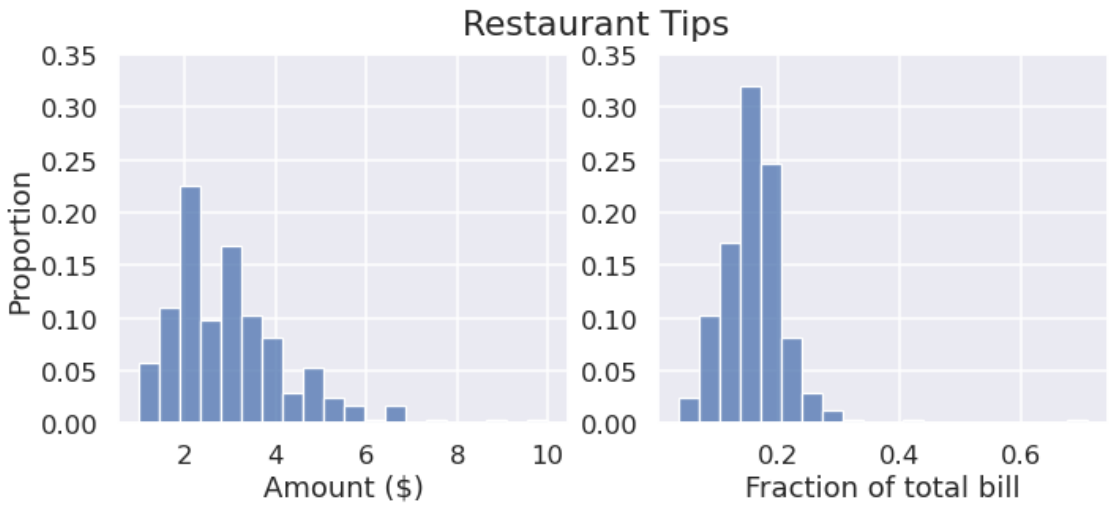
[2]:
```
tips = sns.load_dataset("tips")
tips.head(5)
```

[2]:

	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

Quick EDA: Note that this dataset is likely from the United States. The below plot graphs the distribution of tips in this dataset, both in absolute amounts ($) and as a fraction of the total bill (post-tax, but pre-tip).

[3]:
```
# just run this cell
fig, ax = plt.subplots(ncols=2, figsize=(10, 4))
sns.histplot(tips['tip'], bins=20, stat="proportion", ax=ax[0])
sns.histplot(tips['tip']/tips['total_bill'], bins=20, stat="proportion",␣
 ↪ax=ax[1])
ax[0].set_xlabel("Amount ($)")
ax[1].set_xlabel("Fraction of total bill")
ax[0].set_ylim((0, 0.35))
ax[1].set_ylim((0, 0.35))
ax[1].set_ylabel("") # for cleaner visualization
fig.suptitle("Restaurant Tips")
plt.show()
```



In this lab we'll estimate the tip in **absolute amounts ($)**. The above plot is just to confirm your expectations about the `tips` dataset.

1.2 Part A: Tips as a Summary Statistic

Let's first predict any restaurant tip using one single number: in other words, let's try to find the best statistic $\hat{\theta}$ to represent (i.e., **summarize**) the tips from our dataset.

Each actual tip in our dataset is y, which is what we call the **observed value**. We want to predict each observed value as $\hat{y}$. We'll save the observed tip values in a NumPy array `y_tips`:

[4]:
```
# just run this cell
y_tips = np.array(tips['tip']) # array of observed tips
y_tips.shape
```

[4]:
```
(244,)
```

Recall the three-step process for modeling as covered in lecture:

1. Define a **model.**
2. Define a **loss function** and the associated **risk** on our training dataset (i.e., average loss).
3. Find the best value of θ, known as $\hat{\theta}$, that **minimizes** risk.

We'll go through each step of this process next.

1.3 A.1: Define the model

We will define our model as the **constant model**:

$$\hat{y} = \theta$$

In other words, regardless of any other details (i.e., features) about their meal, we will always predict our tip $\hat{y}$ as one single value: θ.

θ is what we call a **parameter**. Our modeling goal is to find the value of our parameter(s) that **best fit our data**. We have choice over which θ we pick (using the data at hand), but ultimately we can only pick one to report, so we want to find the optimal parameter(s) $\hat{\theta}$.

We call the constant model a **summary statistic**, as we are determining one number that best "summarizes" a set of values.

No code to write here!

1.4 A.2: Define the loss function and risk

Next, in order to pick our θ, we need to define a **loss function**, which is a measure of how well a model is able to predict the expected outcome. In other words, it measures the deviation of a predicted value $\hat{y}$ from the observed value y.

We will use **squared loss** (also known as the L_2 loss, pronounced "ell-two"). For an observed tip value y (i.e., the real tip), our prediction of the tip $\hat{y}$ would give an L_2 loss of:

$$L_2(y, \hat{y}) = (y - \hat{y})^2$$

1.5 Question 1

In our constant model $\hat{y} = \theta$, we always predict the tip as θ. Therefore our L_2 loss for some actual, observed value y can be rewritten as:

$$L_2(y, \theta) = (y - \theta)^2$$

Use the function description below to implement the squared loss function for this single datapoint, assuming the constant model. Your answer should not use any loops.

[5]:
```
def squared_loss(y_obs, theta):
    """
    Calculate the squared loss of the observed data and a summary statistic.

    Parameters
    ------------
    y_obs: an observed value
    theta : some constant representing a summary statistic

    Returns
    ------------
    The squared loss between the observation and the summary statistic.
    """
    return (y_obs - theta)**2 # SOLUTION
```

[ ]:
```
grader.check("q1")
```

We just defined loss for a single datapoint. Let's extend the above loss function to our entire dataset by taking the **average loss** across the dataset.

Let the dataset $\mathcal{D}$ be the set of observations: $\mathcal{D} = \{y_1, \ldots, y_n\}$, where y_i is the i^{th} tip (this is the `y_tips` array defined at the beginning of Part A).

We can define the average loss (aka **risk**) over the dataset as:

$$R\left(\theta\right) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \hat{y}_i)$$

If we use L_2 loss per datapoint ($L = L_2$), then the risk is also known as **mean squared error** (MSE). For the constant model $\hat{y} = \theta$:

$$R\left(\theta\right) = \frac{1}{n}\sum_{i=1}^{n} (y_i - \theta)^2$$

1.6 Question 2

Define the `mse_tips_constant` function which computes $R(\theta)$ as the **mean squared error** on the tips data for a constant model with parameter θ.

Notes/Hints: * This function takes in one parameter, `theta`; `data` is defined for you as a NumPy array that contains the observed tips values in the data. * Use the `squared_loss` function you defined in the previous question.

[10]:
```
def mse_tips_constant(theta):
    data = y_tips
    return np.mean([squared_loss(theta, d) for d in data]) # SOLUTION

mse_tips_constant(5.3) # arbitrarily pick theta = 5.3
```

[10]:
```
7.20452950819672
```

[ ]:
```
grader.check("q2")
```

1.7 A.3: Find the θ that minimizes risk

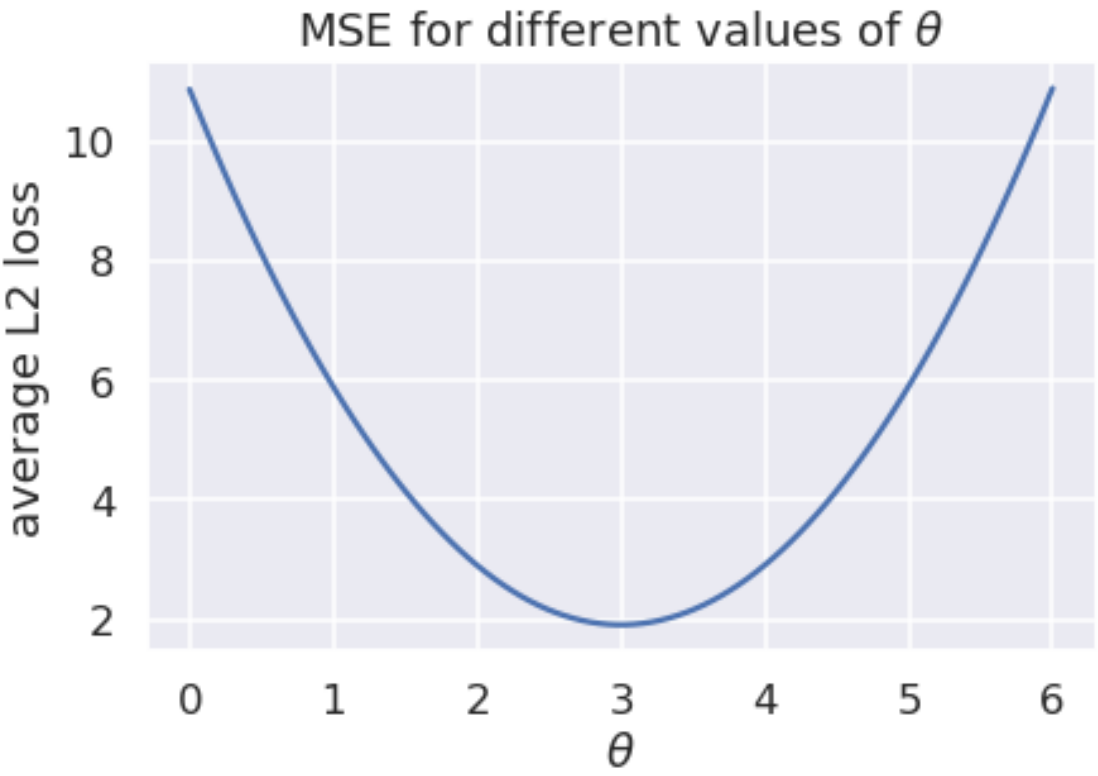
1.8 Question 3

Now we can go about choosing our "best" value of θ, which we call $\hat{\theta}$, that minimizes our defined risk (which we defined as mean squared error). There are several approaches to computing $\hat{\theta}$ that we'll explore in this problem.

1.8.1 Question 3a: Visual Solution

In the cell below we plot the mean squared error for different thetas:

[14]:
```
# just run this cell
theta_values = np.linspace(0, 6, 100)
mse = [mse_tips_constant(theta) for theta in theta_values]
plt.plot(theta_values, mse)
plt.xlabel(r'$\theta$')
plt.ylabel('average L2 loss')
plt.title(r'MSE for different values of $\theta$');
```



Find the value of `theta` that minimizes the mean squared error via observation of the plot above. Round your answer to the nearest integer.

[15]:
```
min_observed_mse = 3 # SOLUTION
min_observed_mse
```

[15]: 3

[ ]:
```
grader.check("q3a")
```

1.8.2 Numerically computing $\hat{\theta}$

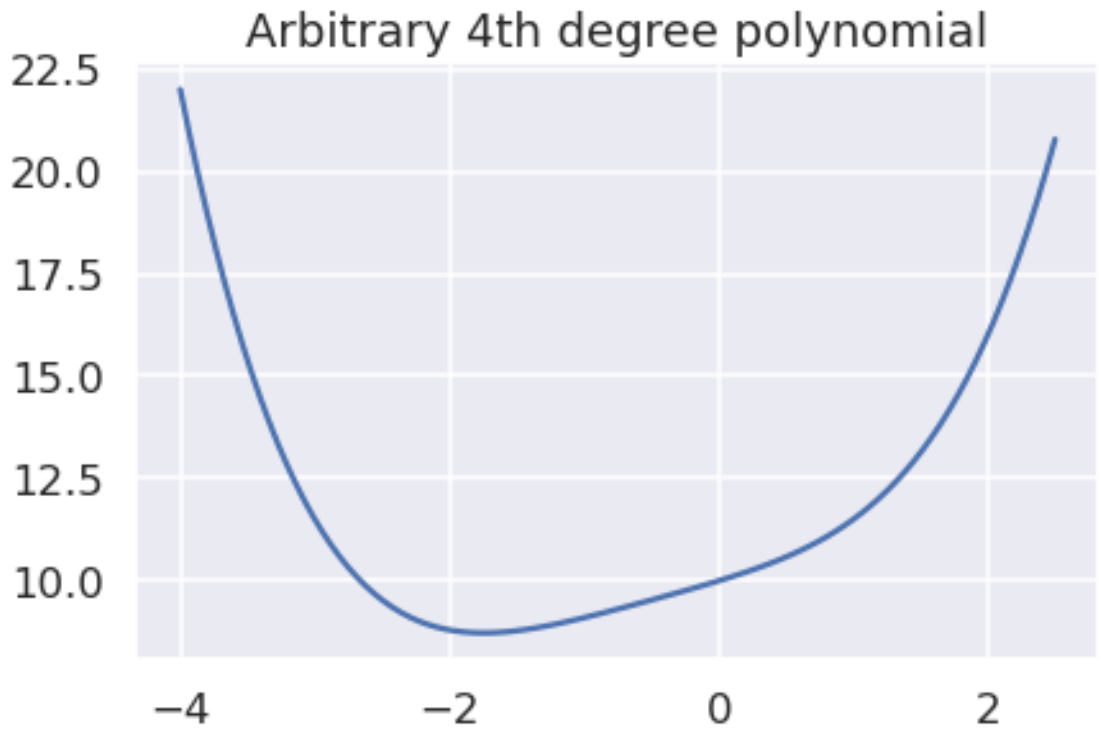
`scipy.optimize.minimize` is a powerful method that can determine the optimal value of a variety of different functions. In practice, it is used to minimize functions that have no (or difficult to obtain) analytical solutions (it is a **numerical method**).

It is overkill for our simple example, but nonetheless, we will show you how to use it, as it will become useful in the near future.

The cell below plots some arbitrary 4th degree polynomial function.

[17]:
```
# just run this cell
x_values = np.linspace(-4, 2.5, 100)
```

```
def fx(x):
    return 0.1 * x**4 + 0.2*x**3 + 0.2 * x **2 + 1 * x + 10

plt.plot(x_values, fx(x_values));
plt.title("Arbitrary 4th degree polynomial");
```



By looking at the plot, we see that the x that minimizes the function is slightly larger than -2. What if we want the exact value? We will demonstrate how to grab the minimum value and the optimal `x` in the following cell.

The function `minimize` from `scipy.optimize` will attempt to minimize any function you throw at it. Try running the cell below, and you will see that `minimize` seems to get the answer correct.

Note: For today, we'll let `minimize` work as if by magic. We'll discuss how `minimize` works later in the course.

[18]:
```
# just run this cell
from scipy.optimize import minimize
minimize(fx, x0 = 1.1)
```

[18]:
```
      fun: 8.728505719866614
 hess_inv: array([[0.50884886]])
      jac: array([1.1920929e-07])
```

```
  message: 'Optimization terminated successfully.'
     nfev: 16
      nit: 6
     njev: 8
   status: 0
  success: True
        x: array([-1.74682779])
```

Notes: [1] `fun`: the minimum value of the function. [2] `x`: the x which minimizes the function. We can index into the object returned by `minimize` to get these values. We have to add the additional `[0]` at the end because the minimizing x is returned as an array, but this is not necessarily the case for other attributes (i.e. `fun`), shown in the cell below.

Note [2] means that `minimize` can also minimize multivariable functions, which we'll see in the second half of this lab.

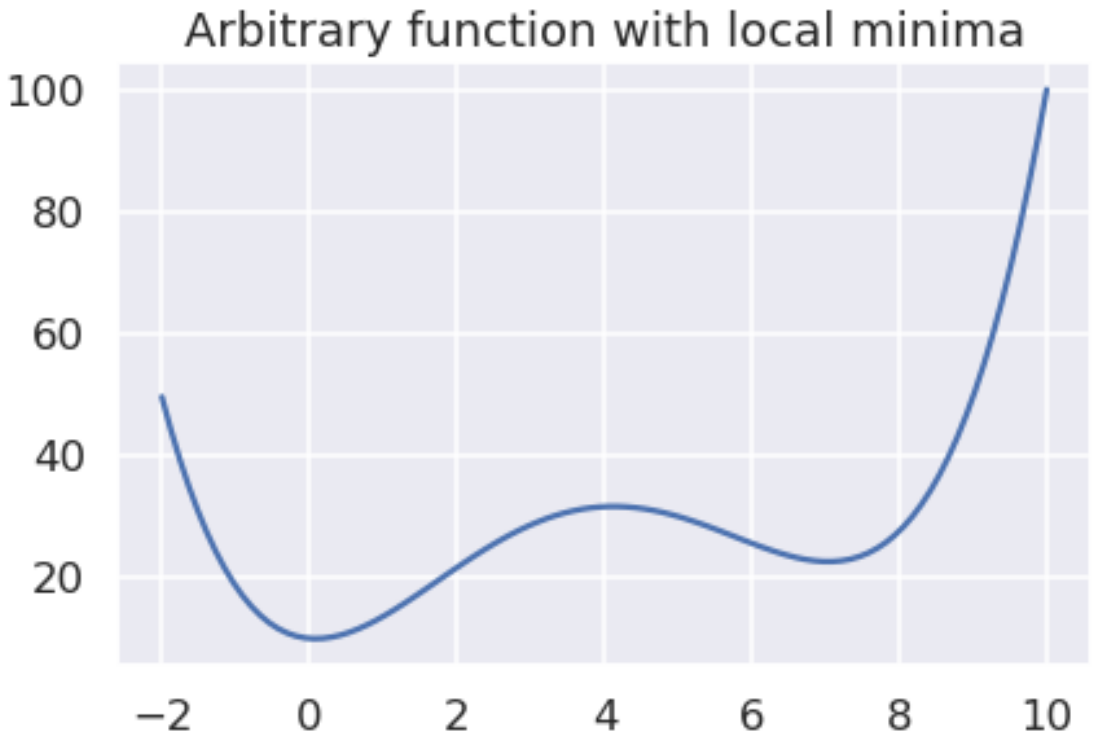
[19]:
```
# just run this cell
min_result = minimize(fx, x0 = 1.1)

min_of_fx = min_result['fun']
x_which_minimizes_fx = min_result['x'][0]
min_of_fx, x_which_minimizes_fx
```

[19]:
```
(8.728505719866614, -1.7468277863801782)
```

Initial guess: The parameter `x0` that we passed to the `minimize` function is where the `minimize` function starts looking as it tries to find the minimum. For example, above, `minimize` started its search at $x = 1.1$ because that's where we told it to start. For the function above, it doesn't really matter what x we start at because the function is nice and has only a single local minimum. More technically, the function is nice because it is convex, a property of functions that we will discuss later in the course.

Local minima: `minimize` isn't perfect. For example, if we give it a function with many valleys (also known as local minima) it can get stuck. For example, consider the function below:

[20]:
```
# just run this cell
w_values = np.linspace(-2, 10, 100)

def fw(w):
    return 0.1 * w**4 - 1.5*w**3 + 6 * w **2 - 1 * w + 10

plt.plot(w_values, fw(w_values));
plt.title("Arbitrary function with local minima");
```



If we start the minimization at `w = 6.5`, we'll get stuck in the local minimum at w = `7.03`. Note that no matter what your actual variable is called in your function (`w` in this case), the `minimize` routine still expects a starting point parameter called `x0`.

[21]:
```
# just run this cell
minimize(fw, x0 = 6.5)     # initial w is 6.5
```

[21]:
```
      fun: 22.594302881719713
 hess_inv: array([[0.12308018]])
      jac: array([-3.81469727e-06])
  message: 'Optimization terminated successfully.'
     nfev: 12
      nit: 4
     njev: 6
   status: 0
  success: True
        x: array([7.03774624])
```

1.8.3 Question 3b: Numerical Solution

Using the `minimize` function, find the value of `theta` that minimizes the mean squared error for our `tips` dataset. In other words, you want to find the exact minimum of the plot that you saw in

the previous part.

Notes: * You should use the function you defined earlier: `mse_tips_constant`. * For autograding purposes, assign `min_scipy` to the value of `theta` that minimizes the MSE according to the `minimize` function, called with initial `x0=0.0`.

[22]:
```
# call minimize with initial x0 = 0.0
min_scipy = minimize(mse_tips_constant, x0=0.0)['x'][0] # SOLUTION
min_scipy
```

[22]:
```
2.9982787346405537
```

[ ]:
```
grader.check("q3b")
```

1.8.4 Question 3c: Analytical Solution

In lecture, we used calculus to show that the value of theta that minimizes the mean squared error for the constant model is the average (mean) of the data. Assign `min_computed` to the mean of the observed `y_tips` data, and compare this to the values you observed in questions 3a and 3b.

[24]:
```
min_computed = np.mean(y_tips) # SOLUTION
min_computed
```

[24]:
```
2.99827868852459
```

[ ]:
```
grader.check("q3c")
```

Reflecting on the lab so far, we used a 3-step approach to find the "best" summary statistic θ:

1. Define the constant model $\hat{y} = \theta$.
2. Define "best": Define loss per datapoint (L2 loss) and consequently define risk $R(\theta)$ over a given data array as the mean squared error, or MSE.
3. Find the $\theta = \hat{\theta}$ that minimizes the MSE $R(\theta)$ in several ways:
    - Visually: Create a plot of $R(\theta)$ vs. θ and eyeball the minimizing $\hat{\theta}$.
    - Numerically: Create a function that returns $R(\theta)$, the MSE for the given data array for a given θ, and use the scipy minimize function to find the minimizing $\hat{\theta}$.
    - Analytically: Simply compute $\hat{\theta}$ the mean of the given data array, since this minimizes the defined $R(\theta)$.
    - (a fourth analytical option) Use calculus to find $\hat{\theta}$ that minimizes MSE $R(\theta)$.

At this point, you've hopefully convinced yourself that the mean of the data is the summary statistic that minimizes mean squared error.

Our prediction for every meal's tip:

[26]:
```
# just run this cell
def predict_tip_constant():
```

```
    return min_computed

# do not edit below this line
bill = 20
print(f"""No matter what meal you have, Part A's modeling process
    predicts that you will pay a tip of ${predict_tip_constant():.2f}.""")
```

```
No matter what meal you have, Part A's modeling process
    predicts that you will pay a tip of $3.00.
```

1.9 Part B: Tips as a Linear Function of Total Bill

In this section, you will follow the exact same modeling process but instead use total bill to predict tip.

We'll save the observed total bill values (post-tax but pre-tip) and the observed tip values in two NumPy arrays, `x_total_bills` and `y_tips`:

[27]:
```
# just run this cell
x_total_bills = np.array(tips['total_bill']) # array of total bill amounts
y_tips = np.array(tips['tip'])               # array of observed tips
print("total bills", x_total_bills.shape)
print("tips", y_tips.shape)
```

```
total bills (244,)
tips (244,)
```

1.10 B.1 Define the model

We will define our model as the **linear model** that takes a single input feature, `total_bill` (x):

$$\hat{y} = a + bx$$

Our "parameter" θ is actually two parameters: a and b. You may see this written as $\theta = (a, b)$.

Our modeling task is then to pick the best values $a = \hat{a}$ and $b = \hat{b}$ from our data. Then, given the total bill x, we can predict the tip as $\hat{y} = \hat{a} + \hat{b}x$.

No code to write here!

1.11 B.2: Define the loss function and risk

Next, we'll define our loss function $L(y, \hat{y})$ and consequently our risk function $R(\theta) = R(a, b)$.

Similar to our approach to Part A, we'll use L2 Loss and Mean Squared Error. Let the dataset $\mathcal{D}$ be the set of observations: $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where (x_i, y_i) are the i^{th} total bill and tip, respectively, in our dataset.

Our L2 Loss and Mean Squared Error are therefore:

$$L_2(y, \hat{y}) = (y - \hat{y})^2 = (y - (a + bx))^2 \tag{1}$$

$$R(a, b) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, \hat{y}_i) = \frac{1}{n} \sum_{i=1}^{n} (y_i - (a + bx_i))^2 \tag{2}$$

Notice that because our model is now the linear model $\hat{y} = a + bx$, our final expressions for Loss and MSE are different from Part A.

1.12 Question 4

Define the `mse_tips_linear` function which computes $R(a, b)$ as the **mean squared error** on the tips data for a linear model with parameters a and b.

Notes: * This function takes in two parameters `a` and `b`. * You should use the NumPy arrays `x_total_bills` and `y_tips` defined at the beginning of Part B. * We've included some skeleton code, but feel free to write your own as well.

[28]:
```
def mse_tips_linear(a, b):
    """
    Returns average L2 loss between
    predicted y_hat values (using x_total_bills and parameters a, b)
    and actual y values (y_tips)
    """
    y_hats = a + b * x_total_bills # SOLUTION
    return np.mean((y_tips - y_hats) ** 2) # SOLUTION

mse_tips_linear(5, 0) # arbitrarily pick a = 0.9, b = 0.1
```

[28]:
```
5.913496721311476
```

[ ]:
```
grader.check("q4")
```

1.13 B.3: Find the θ that minimizes risk

Similar to before, we'd like to try out different approaches to finding the optimal parameters $\hat{a}$ and $\hat{b}$ that minimize MSE.

1.14 Question 5: Analytical Solution

In lecture, we derived the following expression for the line of best fit:

$$\hat{y}_i = \bar{y} + r \frac{SD(y)}{SD(x)} (x_i - \bar{x})$$

where $\bar{x}$, $\bar{y}$, $SD(x)$, $SD(y)$ correspond to the means and standard deviations of x and y, respectively, and r is the correlation coefficient.

1.14.1 Question 5a

Assign `x_bar`, `y_bar`, `std_x`, `std_y`, and `r`, for our dataset. Note: Make sure to use `np.std`, and not `<Series name>.std()`.

- Hint: Remember, in our case, `y` is `y_tips`, and `x` is `x_total_bills`.
- Hint: You may find `np.corrcoef` (documentation) handy in computing `r`. Note that the output of `np.corrcoef` is a matrix, not a number, so you'll need to collect the correlation coefficient by indexing into the returned matrix.

[31]:
```
x_bar = np.mean(x_total_bills) # SOLUTION
y_bar = np.mean(y_tips) # SOLUTION
std_x = np.std(x_total_bills) # SOLUTION
std_y = np.std(y_tips) # SOLUTION
r = np.corrcoef(x_total_bills, y_tips)[0, 1] # SOLUTION
```

[ ]:
```
grader.check("q5a")
```

1.14.2 Question 5b

Now, set `b_hat` and `a_hat` correctly, in terms of the variables you defined above.

Hints: * Try and match the slope and intercept in $\hat{y}_i = \hat{a} + \hat{b}x_i$ to the slope and intercept in $\hat{y}_i = \bar{y} + r\frac{SD(y)}{SD(x)}(x_i - \bar{x})$. * You may want to define `a_hat` in terms of `b_hat`.

[38]:
```
b_hat = r * std_y / std_x # SOLUTION
a_hat = y_bar - b_hat * x_bar # SOLUTION
```

[ ]:
```
grader.check("q5b")
```

1.14.3 Question 5c

Now, use `a_hat` and `b_hat` to implement the `predict_tip_linear` function, which predicts the tip for a total bill amount of `bill`.

[41]:
```
def predict_tip_linear(bill):
    return a_hat + b_hat * bill # SOLUTION

# do not edit below this line
bill = 20
print(f"""If you have a ${bill} bill, Part B's modeling process
    predicts that you will pay a tip of ${predict_tip_linear(bill):.2f}.""")
```

```
If you have a $20 bill, Part B's modeling process
    predicts that you will pay a tip of $3.02.
```

[ ]:
```
grader.check("q5c")
```

1.15 Numerically computing $\hat{\theta}$

The `minimize` function we introduced earlier can also minimize functions of multiple variables (useful for numerically computing $\hat{a}$ and $\hat{b}$. There's one quirk, however, which is that the function has to accept its parameters as a single list.

For example, consider the multivariate $f(u, v) = u^2 - 2uv - 3v + 2v^2$. It turns out this function's minimum is at $(1.5, 1.5)$. To minimize this function, we create `f`.

[43]:
```
# just run this cell
def f(theta):
    u = theta[0]
    v = theta[1]
    return u**2 - 2 * u * v - 3 * v + 2 * v**2

minimize(f, x0 = [0.0, 0.0])
```

[43]:
```
      fun: -2.2499999999999982
 hess_inv: array([[0.99999999, 0.5       ],
       [0.5       , 0.5       ]])
      jac: array([-5.96046448e-08,  0.00000000e+00])
  message: 'Optimization terminated successfully.'
     nfev: 12
      nit: 3
     njev: 4
   status: 0
  success: True
        x: array([1.49999995, 1.49999997])
```

1.16 Question 6: Numerical Solution

1.16.1 Question 6a

Implement the `mse_tips_linear_list` function, which is exactly like `mse_tips_linear` defined in Question 4 except that it takes in a single list of 2 variables rather than two separate variables. For example `mse_tips_linear_list([2, 3])` should return the same value as `mse_tips_linear(2, 3)`.

[44]:
```
def mse_tips_linear_list(theta):
    """
    Returns average L2 loss between
    predicted y_hat values (using x_total_bills and linear params theta)
    and actual y values (y_tips)
```

```
    """
    # BEGIN SOLUTION
    a, b = theta[0], theta[1]
    y_hats = a + b * x_total_bills
    return np.mean((y_tips - y_hats) ** 2)
    # END SOLUTION
```

[ ]:
```
grader.check("q6a")
```

1.16.2 Question 6b

Now, set `min_scipy_linear` to the result of calling minimize to optimize the risk function you just implemented.

Hint: Make sure to set `x0`, say, to `[0.0, 0.0]`.

[46]:
```
# call minimize with initial x0 = [0.0, 0.0]
min_scipy_linear = minimize(mse_tips_linear_list, x0 = [0.0, 0.0]) # SOLUTION
min_scipy_linear
```

[46]:
```
      fun: 1.0360194420114932
 hess_inv: array([[ 2.9799997 , -0.1253415 ],
       [-0.1253415 ,  0.00633488]])
      jac: array([-4.47034836e-08, -2.98023224e-08])
  message: 'Optimization terminated successfully.'
     nfev: 15
      nit: 3
     njev: 5
   status: 0
  success: True
        x: array([0.92027035, 0.10502448])
```

Based on the above output from your call to `minimize`, running the following cell will set and print the values of `a_hat` and `b_hat`.

[47]:
```
# just run this cell
a_hat_scipy = min_scipy_linear['x'][0]
b_hat_scipy = min_scipy_linear['x'][1]
a_hat_scipy, b_hat_scipy
```

[47]:
```
(0.9202703450693733, 0.10502447914641164)
```

The following cell will print out the values of `a_hat` and `b_hat` computed from both methods ("manual" refers to the analytical solution in Question 5; "scipy" refers to the numerical solution in Question 6). If you've done everything correctly, these should be very close to one another.

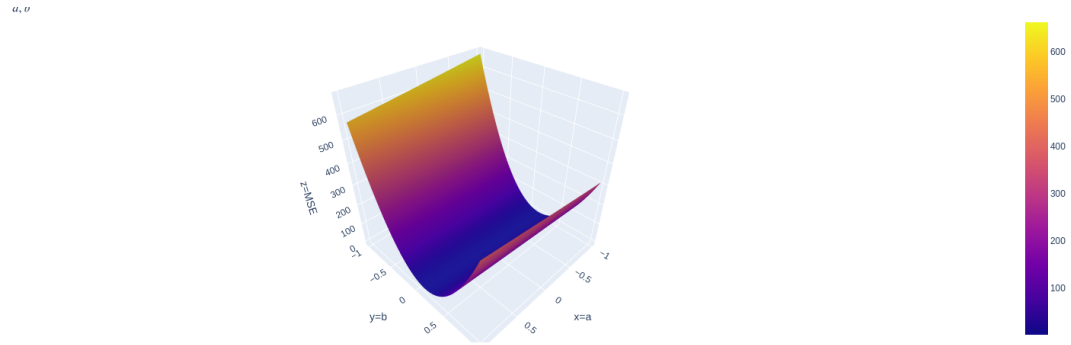
[48]:
```
# just run this cell
print('a_hat_scipy: ', a_hat_scipy)
print('a_hat_manual: ', a_hat)
print('\n')
print('b_hat_scipy: ', b_hat_scipy)
print('b_hat_manual: ', b_hat)
```

```
a_hat_scipy:  0.9202703450693733
a_hat_manual:  0.9202696135546735


b_hat_scipy:  0.10502447914641164
b_hat_manual:  0.10502451738435334
```

Visual Solution (not graded): Feel free to interact with the below plot and verify that the $\hat{a}$ and $\hat{b}$ you computed using either method above minimize the MSE. In the cell below we plot the mean squared error for different parameter values. Note that now that we have two parameters, we have a 3D MSE surface plot.

Rotate the data around and zoom in and out using your trackpad or the controls at the top right of the figure. If you get an error that your browser does not support webgl, you may need to restart your kernel and/or browser.

[49]:
```
# just run this cell
import itertools
import plotly.graph_objects as go

a_values = np.linspace(-1, 1, 80)
b_values = np.linspace(-1,1, 80)
mse_values = [mse_tips_linear(a, b) \
              for a, b in itertools.product(a_values, b_values)]
mse_values = np.reshape(mse_values, (len(a_values), len(b_values)), order='F')
fig = go.Figure(data=[go.Surface(x=a_values, y=b_values, z=mse_values)])
fig.update_layout(
    title=r'MSE for different values of $a, b$',
    autosize=False,
    scene = dict(
                    xaxis_title='x=a',
                    yaxis_title='y=b',
                    zaxis_title='z=MSE'),
                    width=500,
                    margin=dict(r=20, b=10, l=10, t=10))
fig.show()
```

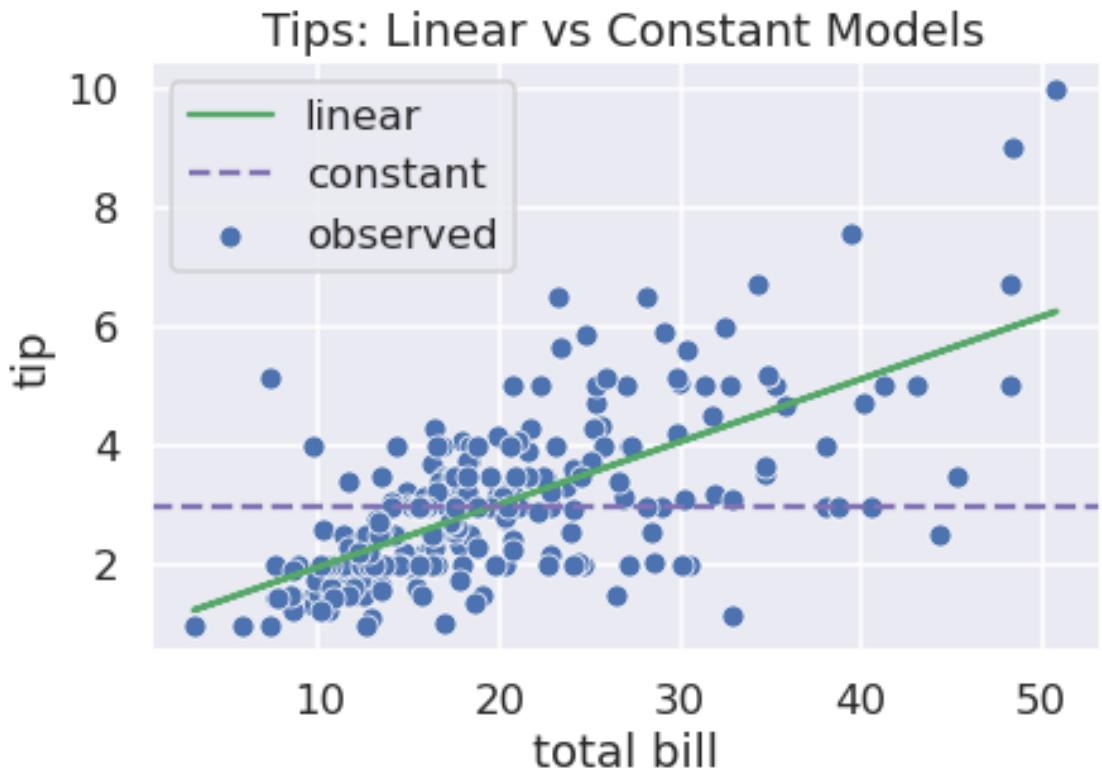


1.16.3 Comparing Constant Model vs Linear Model

At this point, we can actually compare our two models! Both the linear model and constant model were optimized using the same L2 loss function but predict different values for different tips.

Run the cell below:

[50]:
```
sns.scatterplot(x = x_total_bills, y = y_tips, label='observed');

# the below plot expects you've run all of Question 5
plt.plot(x_total_bills, predict_tip_linear(x_total_bills), label='linear',␣
 ↪color='g');

# the below function expects you've run the cell right before part B
plt.axhline(y=predict_tip_constant(), label='constant', color='m', ls='--');

plt.legend()
plt.xlabel("total bill")
plt.ylabel("tip")
plt.title("Tips: Linear vs Constant Models");
plt.show()
```



Note that while we plot tip by total bill, the constant model doesn't use the total bill in its prediction and therefore shows up as a horizontal line.

Thought question: For predicting tip on this data, would you rather use the constant model or the linear model, assuming an L2 loss function for both? This might be more fun with a partner. Note, your answer will not be graded, so don't worry about writing a detailed answer. If you want to see our answer, see the very end of this lab notebook.

Write your answer here, replacing this text.

In the not-so-distant future of this class, you will learn more quantitative metrics to compare model performance. Stay tuned!

1.17 Part C: Using a Different Loss Function

In this last (short) section, we'll consider how the optimal parameters for the **constant model** would change if we used a different loss function.

We will now use **absolute loss** (also known as the L_1 loss, pronounced "ell-one"). For an observed tip value y (i.e., the real tip), our prediction of the tip $\hat{y}$ would give an L_1 loss of:

$$L_1(y, \hat{y}) = |y - \hat{y}|$$

While we still define **risk** as **average loss**, since we now use L_1 loss per datapoint in our datset $\mathcal{D} = \{y_1, \ldots, y_n\}$, our risk is now known as **mean absolute error** (MAE).

For the constant model $\hat{y} = \theta$ (i.e., we predict our tip as a summary statistic):

$$R(\theta) = \frac{1}{n} \sum_{i=1}^{n} L_1(y_i, \hat{y}_i) \tag{3}$$

$$= \frac{1}{n} \sum_{i=1}^{n} |y_i - \theta| \tag{4}$$

Note: the last line results from using the constant model for $\hat{y}$. If we decided to use the linear model, we would have a different expression.

1.18 Question 7

1.18.1 Question 7a

Define the `mae_tips_constant` function which computes $R(\theta)$ as the **mean absolute error** (MAE) on the tips data for a constant model with parameter θ.

Hint: To most efficiently compute the MSE or MAE, try to use NumPy functions or expressions that work for each *word* in the empirical risk function name, starting from the right.

In other words, for mean squared error, we first compute the error by subtracting the data from the constant model parameter θ (**error**), then squaring those errors (**squared**), then taking the mean of those squared errors (**mean**). In implementation, this looks like `np.mean(np.square(data - theta))`.

[51]:
```
def mae_tips_constant(theta):
    data = y_tips
    return np.mean(np.abs(data - theta)) # SOLUTION

mae_tips_constant(5.3) # arbitrarily pick theta = 5.3
```

[51]:
```
2.4527868852459016
```

[ ]:
```
grader.check("q7")
```

1.18.2 Question 7b

In lecture, we saw that the value of theta that minimizes mean *absolute* error for the constant model is the median of the data. Assign `min_computed_mae` to the median of the observed `y_tips` data.

[55]:
```
min_computed_mae = np.median(y_tips) # SOLUTION
min_computed_mae
```

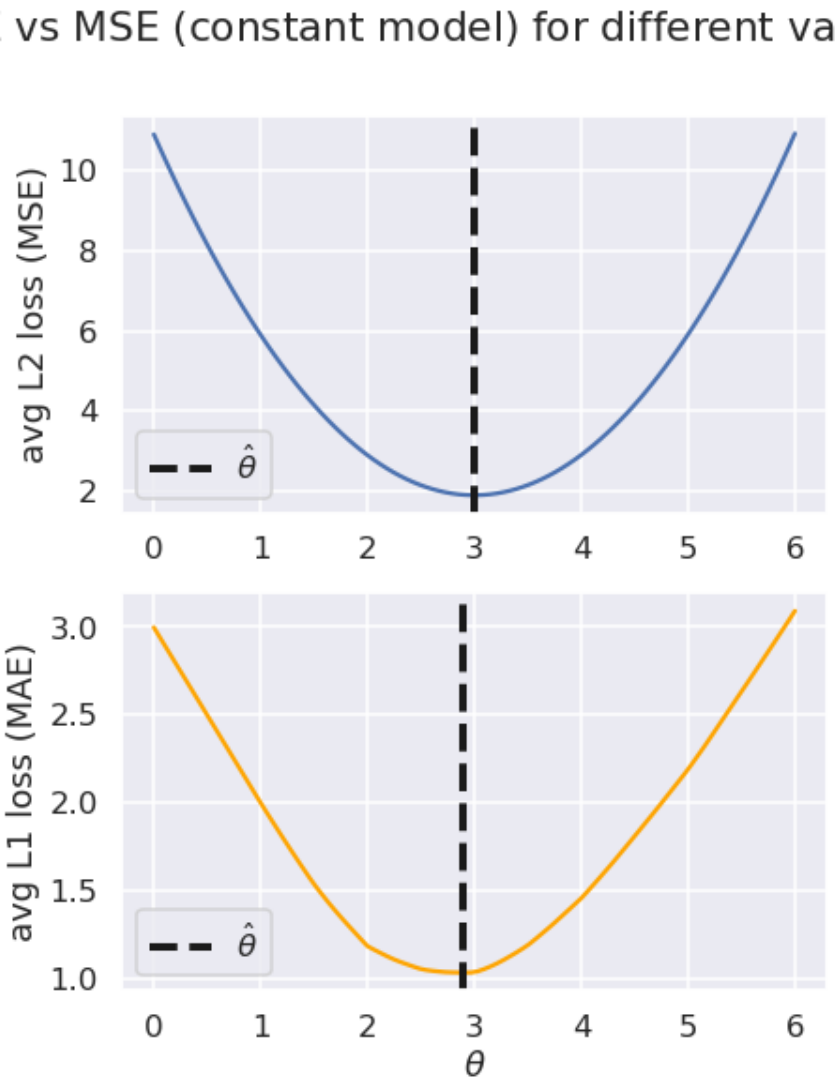
[55]: 2.9

[ ]:
```
grader.check("q7b")
```

1.18.3 Comparing MAE to MSE

Now run the below cell to compare MAE with MSE on the constant model.

[57]:
```
# just run this cell
fig, ax = plt.subplots(nrows=2, figsize=((6, 8)))
theta_values = np.linspace(0, 6, 100)
mse = [mse_tips_constant(theta) for theta in theta_values]
ax[0].plot(theta_values, mse)
ax[0].axvline(x=min_computed, linewidth=4, color='k', ls='--',
              label=r'$\hat{\theta}$')
ax[0].legend()
ax[0].set_ylabel("avg L2 loss (MSE)")

mae = [mae_tips_constant(theta) for theta in theta_values]
ax[1].plot(theta_values, mae, color='orange')
ax[1].axvline(x=min_computed_mae, linewidth=4, color='k', ls='--',
              label=r'$\hat{\theta}$')
ax[1].legend()
ax[1].set_ylabel("avg L1 loss (MAE)")

ax[1].set_xlabel(r'$\theta$');
fig.suptitle(r"MAE vs MSE (constant model) for different values of $\theta$");
```



Thought question You should see that the MAE plot (below) looks somewhat similar the MSE plot (above). Try to identify any key differences you observe and write them down below. This might be more fun with a partner. Note, your answer will not be graded, so don't worry about writing a detailed answer. If you want to see our answer, see the very end of this lab notebook.

Write your answer here, replacing this text.

2 Congratulations! You finished the lab!

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```

2.1 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

[ ]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export(pdf=False)
```