Homework 11: Regression Inference and Classification

Reading:

- Inference for Regression (https://www.inferentialthinking.com/chapters/16/Inference_for_Regression.html)
- Classification (https://www.inferentialthinking.com/chapters/17/Classification.html)

Please complete this notebook by filling in the cells provided. Before you begin, execute the following cell to load the provided tests. Each time you start your server, you will need to execute this cell again to load the tests.

Homework 11 is due **Thursday, 5/6 at 11:59pm**.

Start early so that you can come to office hours if you're stuck. Late work will not be accepted as per the course policies.

In [ ]:
```
# Don't change this cell; just run it.

import numpy as np
from datascience import *

# These lines do some fancy plotting magic.
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
import warnings
warnings.simplefilter('ignore', FutureWarning)
from matplotlib import patches
from ipywidgets import interact, interactive, fixed
import ipywidgets as widgets
```

1. Visual Diagnostics for Linear Regression

Regression Model Diagnostics

Linear regression isn't always the best way to describe the relationship between two variables. We'd like to develop techniques that will help us decide whether or not to use a linear model to predict one variable based on another.

We will use the insight that if a regression fits a set of points well, then the residuals from that regression line will show no pattern when plotted against the predictor variable.

The table below contains information on waiting time between eruptions and the duration of the eruption for the Old Faithful geyser in Yellowstone National Park. Run the next cell to load the data and see a scatter plot.

In [ ]:
```
old_faithful = Table.read_table('faithful.csv')
old_faithful.scatter('duration')
old_faithful
```

Question 1.1. Write a function called `residuals`. It should take a single argument, a table. It should first compute the slope and intercept of the regression line that predicts the second column of that table (accessible as `tbl.column(1)`) using the first column (`tbl.column(0)`). The function should return an array containing the *residuals* for that regression line. Recall that residuals are given by

$$residual = observed\ value - regression\ estimate$$

Hint: If your code is getting long, think about how you can split the problem up into multiple smaller, simpler functions.

In [ ]:
```
def residuals(tbl):
    ...
```

Question 1.2. Make a scatter plot of the residuals for the Old Faithful dataset for predicting wait from duration; duration should be on the horizontal axis.

In [ ]:
```
...
```

Question 1.3. Does the plot of residuals look roughly like a formless cloud, or is there some kind of pattern? Are they centered around 0?

Write your answer here, replacing this text.

Question 1.4. Does it seem like a linear model is appropriate for describing the relationship between waiting time and duration?

Assign `linear` to `True` if a linear model is appropriate for describing the relationship, and `False` if it is not.

In [ ]:
```
linear = ...
```

Section 15.6 (https://www.inferentialthinking.com/chapters/15/6/numerical-diagnostics.html) of the textbook describes some mathematical facts that hold for all regression estimates, regardless of goodness of fit. One fact is that there is a relationship between the standard deviation of the residuals, the standard deviation of the response variable, and the correlation. Let us test this.

Question 1.5. Directly compute the standard deviation of the residuals from the Old Faithful data. Then compute the same quantity without using the residuals, using the formula described in section 15.6 here (https://www.inferentialthinking.com/chapters/15/6/Numerical_Diagnostics.html#sd-of-the-residuals) instead.

In [ ]:
```
faithful_residual_sd = ...
faithful_residual_sd_from_formula = ...

print("Residual SD: {0}".format(faithful_residual_sd))
print("Residual SD from the formula: {0}".format(faithful_residual_sd_fr
om_formula))
```

2. Finding the Least Squares Regression Line

In this exercise, you'll work with a small invented data set. Run the next cell to generate the dataset `d` and see a scatter plot.

In [ ]:
```
d = Table().with_columns(
    'x', make_array(0,  1,  2,  3,  4),
    'y', make_array(1, .5, -1,  2, -3))
d.scatter('x')
```

Question 2.1. (Ungraded, but you'll need the result later) Running the cell below will generate sliders that control the slope and intercept of a line through the scatter plot. When you adjust a slider, the line will move.

By moving the line around, make your best guess at the least-squares regression line. (It's okay if your line isn't exactly right, as long as it's reasonable.)

Note: Python will probably take about a second to redraw the plot each time you adjust the slider. We suggest clicking the place on the slider you want to try and waiting for the plot to be drawn; dragging the slider handle around will cause a long lag.

In [ ]:
```
def plot_line(slope, intercept):
    plt.figure(figsize=(5,5))

    endpoints = make_array(-2, 7)
    p = plt.plot(endpoints, slope*endpoints + intercept, color='orange',
label='Proposed line')

    plt.scatter(d.column('x'), d.column('y'), color='blue', label='Point
s')

    plt.xlim(-4, 8)
    plt.ylim(-6, 6)
    plt.gca().set_aspect('equal', adjustable='box')

    plt.legend(bbox_to_anchor=(1.8, .8))
    plt.show()

interact(plot_line, slope=widgets.FloatSlider(min=-4, max=4, step=.1), i
ntercept=widgets.FloatSlider(min=-4, max=4, step=.1));
```

You can probably find a reasonable-looking line by just eyeballing it. But remember: the least-squares regression line minimizes the mean of the squared errors made by the line for each point. Your eye might not be able to judge squared errors very well.

A note on mean and total squared error

It is common to think of the least-squares line as the line with the least *mean* squared error (or the square root of the mean squared error), as the textbook does.

But it turns out that it doesn't matter whether you minimize the mean squared error or the *total* squared error. You'll get the same best line in either case.

That's because the total squared error is just the mean squared error multipled by the number of points (`d.num_rows`). So if one line gets a better total squared error than another line, then it also gets a better mean squared error. In particular, the line with the smallest total squared error is also better than every other line in terms of mean squared error. That makes it the least squares line.

tl; dr: Minimizing the mean squared error minimizes the total squared error as well.

Question 2.2. (Ungraded, but you'll need the result later) The next cell produces a more useful plot. Use it to find a line that's closer to the least-squares regression line, keeping the above note in mind.

In [ ]:
```
def plot_line_and_errors(slope, intercept):
    plt.figure(figsize=(5,5))
    points = make_array(-2, 7)
    p = plt.plot(points, slope*points + intercept, color='orange', label
='Proposed line')
    ax = p[0].axes

    predicted_ys = slope*d.column('x') + intercept
    diffs = predicted_ys - d.column('y')
    for i in np.arange(d.num_rows):
        x = d.column('x').item(i)
        y = d.column('y').item(i)
        diff = diffs.item(i)

        if diff > 0:
            bottom_left_x = x
            bottom_left_y = y
        else:
            bottom_left_x = x + diff
            bottom_left_y = y + diff

        ax.add_patch(patches.Rectangle(make_array(bottom_left_x, bottom_
left_y), abs(diff), abs(diff), color='red', alpha=.3, label=('Squared er
ror' if i == 0 else None)))
        plt.plot(make_array(x, x), make_array(y, y + diff), color='red',
alpha=.6, label=('Error' if i == 0 else None))

    plt.scatter(d.column('x'), d.column('y'), color='blue', label='Point
s')

    plt.xlim(-4, 8)
    plt.ylim(-6, 6)
    plt.gca().set_aspect('equal', adjustable='box')

    plt.legend(bbox_to_anchor=(1.8, .8))
    plt.show()

interact(plot_line_and_errors, slope=widgets.FloatSlider(min=-4, max=4,
step=.1), intercept=widgets.FloatSlider(min=-4, max=4, step=.1));
```

Question 2.3. Describe the visual criterion you used to find a line in question 2.2. How did you judge whether one line was better than another?

For example, a possible (but incorrect) answer is, "I tried to make the red line for the bottom-right point as small as possible."

Write your answer here, replacing this text.

Question 2.4. We can say that a point influences the line by how much the line would move if the point was removed from the data set. Does the outlier at (3, 2) have more or less influence than any other point on the resulting best-fit line?

Assign `more_influence` to `True` if the outlier (3,2) has more influence than any other point on the best-fit line, or `False` if it does not.

In [ ]:
```
more_influence = ...
```

Now, let's have Python find this line for us. When we use `minimize`, Python goes through a process similar to the one you might have used in question 2.2.

But Python can't look at a plot that displays errors! Instead, we tell it how to find the total squared error for a line with a given slope and intercept.

Question 2.5. Define a function called `total_squared_error`. It should take two numbers as arguments:

1. the slope of some potential line
2. the intercept of some potential line

It should return the total squared error when we use that line to make predictions for the dataset `d`.

Recall that `d` has two columns: x and y.

In [ ]:
```
def total_squared_error(slope, intercept):
    predictions = ...
    errors = ...
    ...
```

Question 2.6. What is the total squared error for the line you found by "eyeballing" the errors in question 2.1? What about question 2.2, where you made a guess that was "aided" by a visualization of the squared error? (It's okay if the error went up, but for many students, the error will go down when using the visual aid.)

In [ ]:
```
eyeballed_error = ...
aided_error = ...
print("Eyeballed error:", eyeballed_error, "\nAided error:", aided_error
)
```

Question 2.7. Use `minimize` to find the slope and intercept for the line that minimizes the total squared error. This is the definition of a least-squares regression line.

Note: `minimize` will return a single array containing the slope as the first element and intercept as the second. Read more of its documentation here (http://data8.org/datascience/util.html?highlight=minimize#datascience.util.minimize) or an example of its use here (https://www.inferentialthinking.com/chapters/15/3/method-of-least-squares.html).

In [ ]:
```
# The staff solution used 1 line of code above here.
slope_from_minimize = ...
intercept_from_minimize = ...
print("Least-squares regression line: predicted_y =",
      slope_from_minimize,
      "* x + ",
      intercept_from_minimize)
```

Question 2.8. What is the total squared error for the least-squares regression line that you found?

In [ ]:
```
best_total_squared_error = ...
best_total_squared_error
```

Finally, run the following cell to plot this "best fit" line and its errors:

In [ ]:
```
plot_line_and_errors(slope_from_minimize, intercept_from_minimize)
```

3. Reading Sign Language with Classification

Brazilian Sign Language is a visual language used primarily by Brazilians who are deaf. It is more commonly called Libras. People who communicate with visual language are called *signers*. Here is a video of someone signing in Libras:

In [ ]:
```
from IPython.lib.display import YouTubeVideo
YouTubeVideo("SiJUggsJ3e8")
```

Programs like Siri or Google Now begin the process of understanding human speech by classifying short clips of raw sound into basic categories called *phones*. For example, the recorded sound of someone saying the word "robot" might be broken down into several phones: "rrr", "oh", "buh", "aah", and "tuh". Phones are then grouped together into further categories like words ("robot") and sentences ("I, for one, welcome our new robot overlords") that carry more meaning.

A visual language like Libras has an analogous structure. Instead of phones, each word is made up of several *hand movements*. As a first step in interpreting Libras, we can break down a video clip into small segments, each containing a single hand movement. The task is then to figure out what hand movement each segment represents.

We can do that with classification!

The data (https://archive.ics.uci.edu/ml/machine-learning-databases/libras/movement_libras.names) *(you can use Notepad or TextEdit to open it)* in this exercise come from Dias, Peres, and Biscaro, researchers at the University of Sao Paulo in Brazil. They identified 15 distinct hand movements in Libras (probably an oversimplification, but a useful one) and captured short videos of signers making those hand movements. (You can read more about their work here (http://ieeexplore.ieee.org/Xplore/login.jsp?url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F5161636%2F5178557%2F05178917.pdf&authDecision=-2(The paper is gated, so you will need to use your institution's Wi-Fi or VPN to access it.)

For each video, they chose 45 still frames from the video and identified the location (in horizontal and vertical coordinates) of the signer's hand in each frame. Since there are two coordinates for each frame, this gives us a total of 90 numbers summarizing how a hand moved in each video. Those 90 numbers will be our *attributes*.

Each video is *labeled* with the kind of hand movement the signer was making in it. Each label is one of 15 strings like "horizontal swing" or "vertical zigzag".

For simplicity, we're going to focus on distinguishing between just two kinds of movements: "horizontal straight-line" and "vertical straight-line". We took the Sao Paulo researchers' original dataset, which was quite small, and used some simple techniques to create a much larger synthetic dataset.

These data are in the file `movements.csv` . Run the next cell to load it.

In [ ]:
```
movements = Table.read_table("movements.csv")
movements.take(np.arange(5))
```

The cell below displays movements graphically. Run it and use the slider to answer the next question.

In [ ]:
```
# Just run this cell and use the slider it produces.
def display_whole_movement(row_idx):
    num_frames = int((movements.num_columns-1)/2)
    row = np.array(movements.drop("Movement type").row(row_idx))
    xs = row[np.arange(0, 2*num_frames, 2)]
    ys = row[np.arange(1, 2*num_frames, 2)]
    plt.figure(figsize=(5,5))
    plt.plot(xs, ys, c="gold")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim(-.5, 1.5)
    plt.ylim(-.5, 1.5)
    plt.gca().set_aspect('equal', adjustable='box')

def display_hand(example, frame, display_truth):
        time_idx = frame-1
        display_whole_movement(example)
        x = movements.column(2*time_idx).item(example)
        y = movements.column(2*time_idx+1).item(example)
        plt.annotate(
            "frame {:d}".format(frame),
            xy=(x, y), xytext=(-20, 20),
            textcoords = 'offset points', ha = 'right', va = 'bottom',
            color='white',
            bbox = {'boxstyle': 'round,pad=0.5', 'fc': 'black', 'alpha':
.4},
            arrowprops = {'arrowstyle': '->', 'connectionstyle':'arc3,ra
d=0', 'color': 'black'})
        plt.scatter(x, y, c="black", zorder=10)
        plt.title("Hand positions for movement {:d}{}".format(example, "
\n(True class: {})".format(movements.column("Movement type").item(exampl
e)) if display_truth else ""))

def animate_movement():
    interact(
        display_hand,
        example=widgets.BoundedIntText(min=0, max=movements.num_rows-1,
value=0, msg_throttle=1),
        frame=widgets.IntSlider(min=1, max=int((movements.num_columns-1)
/2), step=1, value=1, msg_throttle=1),
        display_truth=fixed(False))

animate_movement()
```

Question 3.1. Before we move on, check your understanding of the dataset. Judging by the plot, is the first movement example (movement 0) a vertical motion, or a horizontal motion? If it is hard to tell, does it seem more likely to be vertical or horizontal? This is the kind of question a classifier has to answer. Find out the right answer by looking at the `"Movement type"` column.

Assign `first_movement` to `1` if the movement was vertical, or `2` if the movement was horizontal.

In [ ]:
```
first_movement = ...
```

Splitting the dataset

We'll do 2 different kinds of things with the `movements` dataset:

1. We'll build a classifier that uses the movements with known labels as examples to classify similar movements. This is called *training*.
2. We'll evaluate or *test* the accuracy of the classifier we build.

For reasons discussed in lecture and the textbook, we want to use separate datasets for these two purposes. So we split up our one dataset into two.

Question 3.2. Create a table called `train_movements` and another table called `test_movements`. `train_movements` should include the first $\frac{11}{16}$th of the rows in `movements` (rounded to the nearest integer), and `test_movements` should include the remaining $\frac{5}{16}$th.

Note that we do **not** mean the first 11 rows for the training test and rows 12-16 for the test set. We mean the first $\frac{11}{16} = 68.75\%$ of the table should be for the the trianing set, and the rest should be for the test set.

Hint: Use the table method `take`.

In [ ]:
```
training_proportion = 11/16
num_movements = movements.num_rows
num_train = int(round(num_movements * training_proportion))

train_movements = ...
test_movements = ...

print("Training set:\t",   train_movements.num_rows, "examples")
print("Test set:\t",       test_movements.num_rows, "examples")
```

Using only 2 features

First let's see how well we can distinguish two movements (a vertical line and a horizontal line) using the hand position from just a single frame (without the other 44).

Question 3.3. Make a table called `train_two_features` with only 3 columns: the first frame's x coordinate and first frame's y coordinate are our chosen features, as well as the movement type; only the examples in `train_movements`.

In [ ]:
```
train_two_features = ...
train_two_features
```

Now we want to make a scatter plot of the frame coordinates, where the dots for horizontal straight-line movements have one color and the dots for vertical straight-line movements have another color. Here is a scatter plot without colors:

In [ ]:
```
train_two_features.scatter("Frame 1 x", "Frame 1 y")
```

This isn't useful because we don't know which dots are which movement type. We need to tell Python how to color the dots. Let's use gold for vertical and blue for horizontal movements.

`scatter` takes an extra argument called `colors` that's the name of an extra column in the table that contains colors (strings like "red" or "orange") for each row. So we need to create a table like this:

Frame 1 x	Frame 1 y	Movement type	Color
0.522768	0.769731	vertical straight-line	gold
0.179546	0.658986	horizontal straight-line	blue
...	...	...	...

Question 3.4. In the cell below, create a table named `with_colors`. It should have the same columns as the example table above, but with a row for each row in `train_two_features`. Then, create a scatter plot of your data.

In [ ]:
```
# You should find the following table useful.
type_to_color = Table().with_columns(
    "Movement type", make_array("vertical straight-line", "horizontal st
raight-line"),
    "Color",         make_array("gold",                   "blue"))

with_colors = ...
with_colors.scatter("Frame 1 x", "Frame 1 y", colors="Color")
```

Question 3.5. Based on the scatter plot, how well will a nearest-neighbor classifier based on only these 2 features (the x- and y-coordinates of the hand position in the first frame) work? Will it:

1. distinguish almost perfectly between vertical and horizontal movements;
2. distinguish somewhat well between vertical and horizontal movements, getting some correct but missing a substantial proportion; or
3. be basically useless in distinguishing between vertical and horizontal movements?

Why?

Write your answer here, replacing this text.

4. Submission

Once you're finished, submit your assignment as a .pdf (download as .html, then print to save as a .pdf) on Gradescope.