lab0_student_version

August 30, 2021

1 Lab 0: Review and warm-up

Welcome to the first Data 102 lab!

The goal of this lab is to review some basic probability and programming. We will also learn more about binary decision making.

The code (and answers) that you need to write is commented out with a message **"TODO: …"**. There is additional documentation for each part as you go along.

1.1 Collaboration Policy

Data science is a collaborative activity. While you may talk with 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 cell below.

1.2 Submission

To submit this assignment, rerun the notebook from scratch (by selecting Kernel > Restart & Run all), and then print as a pdf (File > download as > pdf) and submit it to Gradescope.

For full credit, this assignment should be completed and submitted before Wednesday, September 1, 2021 at 11:59 PM. PST

1.3 Collaborators

Write the names of your collaborators in this cell.

`<Collaborator Name> <Collaborator e-mail>`

2 Setup

Let's begin by importing the libraries we will use. You can find the documentation for the libraries here: * matplotlib: https://matplotlib.org/3.1.1/contents.html * numpy: https://docs.scipy.org/doc/ * pandas: https://pandas.pydata.org/pandas-docs/stable/ * seaborn: https://seaborn.pydata.org/

[135]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression

import timeit
import hashlib
%matplotlib inline

sns.set(style="dark")
plt.style.use("ggplot")

def get_hash(num):  # <- helper function for assessing correctness
    return hashlib.md5(str(num).encode()).hexdigest()
```

3 Question 0: Vectorized operations in NumPy

We'll start this lab by reviewing the concept of **vectorization**. Many of the functions in NumPy (and pandas) are optimized to be much faster than the equivalent code using a for loop. This is because NumPy uses optimized and pre-compiled code written in a low-level language (in this case, C) to carry out mathematical operations. By using NumPy's vectorized operations instead of iterating explicitly (e.g., writing `for` loops), we can make our code run much faster. In some cases, this difference is small, but you'll see in future labs and homework assignments that sometimes it has a big impact.

Let's see vectorization in action, and measure the time it takes to perform some vectorized and non-vectorized tasks on NumPy arrays. We'll start by summing numbers from 0 to 14,999.

[136]:
```python
%%timeit

sum_nonvect = 0
for item in range(0, 15000):
     sum_nonvect += item
```

```
777 µs ± 20 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[137]:
```python
%%timeit

sum_vect = np.sum(np.arange(15000))
```

```
33.2 µs ± 627 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

3.0.1 a) How much faster is the vectorized version? Your answer should be a multiplicative factor (e.g., "it takes half as long").

The vectorized version is about 20 times faster.

Now, consider the following array:

[138]:
```
arr = np.arange(0, 60, 3).reshape(4, 5)
arr
```

[138]:
```
array([[ 0,  3,  6,  9, 12],
       [15, 18, 21, 24, 27],
       [30, 33, 36, 39, 42],
       [45, 48, 51, 54, 57]])
```

Suppose we want to compute a new array where each entry is the average of two neighboring entries in the original array. So, the first row would look like `[1.5, 4.5, 7.5, 10.5]` (we'll see examples of operations like this later in the class). Let's try doing this two different ways:

[139]:
```
%%timeit
# Using nested for loops

new_array_slow = np.zeros([arr.shape[0], arr.shape[1] - 1])
for i in range(arr.shape[0]):
    for j in range (arr.shape[1] - 1):
        new_array_slow[i,j] = (arr[i,j] + arr[i, j+1])/2
```

```
13.7 µs ± 118 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[140]:
```
%%timeit

new_array_fast = (arr[:, :-1] + arr[:, 1:])/2
```

```
3.88 µs ± 117 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

3.0.2 b) Suppose your friend looks at this and says that while the vectorized version does run faster, both versions run fast enough that it doesn't matter. Give two reasons your friend might be wrong.

Hint: What happens if an algorithm needs to run this kind of operation many times?

Hint: try replacing `arr` with `np.random.random([2000, 2000])`. How do the results change?

Reason 1: Even though it may run at comparable speeds right now, as the size of the input array increases, the time it takes to run it may increase exponentially.

Reason 2: In the slower version, you are doing all the operations one by one, but in the faster one, you are ignoring this additional work because of how level the backend of NumPy is.

4 Question 1: The sinking of the Titanic

On April 15, 1912, during her maiden voyage, the Titanic sank after colliding with an iceberg. Unfortunately, there weren't enough lifeboats for everyone onboard, resulting in the death of 1502 out of 2224 passengers and crew.

In this question we will work with data about passengers traveling on the Titanic, and we will try to understand whether some groups of people were more likely to survive than others. We will also fit a binary classifier to the data, and check its performance (e.g. how many false positive does it produce?)

Let's start by importing the Titanic dataset using Pandas.

[141]:
```
titanic = pd.read_csv("titanic.csv")
titanic.head()
```

[141]:
```
   Survived  Pclass                                               Name  \
0         0       3                             Mr. Owen Harris Braund
1         1       1  Mrs. John Bradley (Florence Briggs Thayer) Cum…
2         1       3                              Miss. Laina Heikkinen
3         1       1        Mrs. Jacques Heath (Lily May Peel) Futrelle
4         0       3                            Mr. William Henry Allen

      Sex   Age  Siblings/Spouses Aboard  Parents/Children Aboard     Fare
0    male  22.0                        1                        0   7.2500
1  female  38.0                        1                        0  71.2833
2  female  26.0                        0                        0   7.9250
3  female  35.0                        1                        0  53.1000
4    male  35.0                        0                        0   8.0500
```

Here is a brief description of the variables included in the dataframe: * **Survived**: binary variable taking value 1 if the person survived the shipwreck, 0 otherwise; * **Pclass**: whether the passenger was traveling in 1st, 2nd, or 3rd class; * **Name**: passenger's name; * **Sex**: passenger's gender; * **Age**: passenger's age; * **Siblings/Spouses Aboard**: how many siblings or spouses the passenger is traveling with; * **Parents/Children Aboard**: how many parents or children the passenger is traveling with; * **Fare**: what was the fare that the passenger paid.

4.1 Part 1.a: dataset check

We will first check some general properties of the dataframe. For example, we will see how many rows and columns the dataframe has, and if there are missing values.

[142]:
```
# TODO: find the number of rows and columns in the dataframe
rows = titanic.shape[0]
cols = titanic.shape[1]
print(rows, cols)
```

```
887 8
```

[143]:
```
# TODO: check how many missing values there are per column
titanic.isna().sum()
```

[143]:
```
Survived                   0
Pclass                     0
Name                       0
Sex                        0
Age                        0
```

```
Siblings/Spouses Aboard    0
Parents/Children Aboard    0
Fare                       0
dtype: int64
```

Running the cell below, we can also check how many different values each variable takes.

[144]:
```
titanic.nunique()
```

[144]:
```
Survived                     2
Pclass                       3
Name                       887
Sex                          2
Age                         89
Siblings/Spouses Aboard      7
Parents/Children Aboard      7
Fare                       248
dtype: int64
```

4.2 Part 1.b: exploring the dataset

We'll now look into more detail at some descriptive statistics and plots. We would like to know what is the percentage of people who survived the shipwreck, and what is the passengers' distribution in terms of sex, age, class, etc.

For a refresher on data visualization using pandas and seaborn, you can refer to Chapter 10 of the Data 100 textbook.

Let's start by computing the percentage of people in the dataset who survived the shipwreck.

[145]:
```
# TODO: compute the percentage of people who survived
survival_prob = titanic[titanic['Survived'] == 1].shape[0] / titanic.shape[0]
```

[146]:
```
# Validation tests: do not modify!
assert(get_hash(survival_prob)== '4fe6b7a204127acbec520ac2997133dc')
print("Test passed!")
```

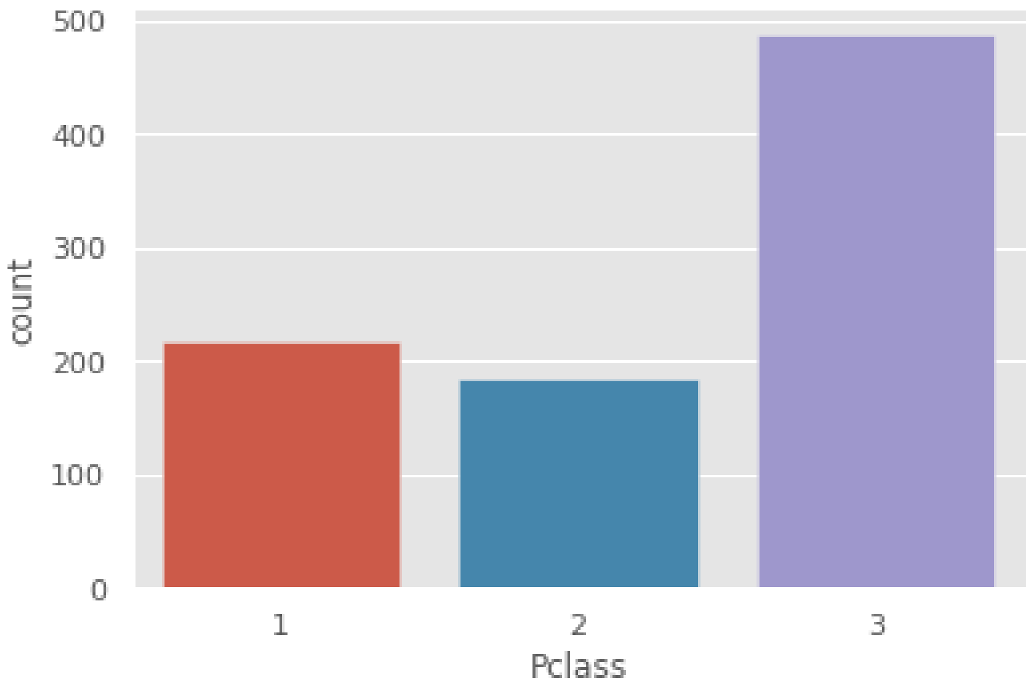
```
Test passed!
```

The cell below uses the `countplot` function from seaborn to generate a plot displaying how many passengers were traveling in each of the three classes.

[147]:
```
sns.countplot(titanic["Pclass"])
```

```
/opt/conda/lib/python3.9/site-packages/seaborn/_decorators.py:36: FutureWarning:
Pass the following variable as a keyword arg: x. From version 0.12, the only
valid positional argument will be `data`, and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```

[147]: <AxesSubplot:xlabel='Pclass', ylabel='count'>



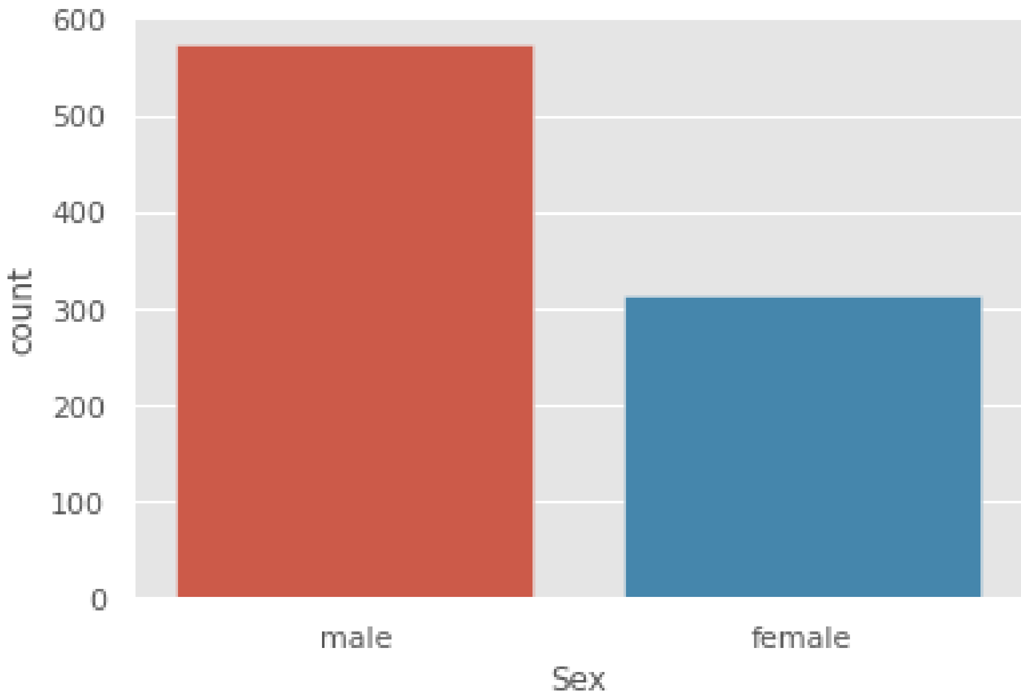
We can see that the distribution is not uniform, and that the majority of the passengers were traveling in third class.

In the next question, write a line of code to generate a bar graph for the distribution of passenger sex.

[148]:
```
# TODO: use countplot to show how the distribution of passenger sex
sns.countplot(titanic["Sex"])
```

```
/opt/conda/lib/python3.9/site-packages/seaborn/_decorators.py:36: FutureWarning:
Pass the following variable as a keyword arg: x. From version 0.12, the only
valid positional argument will be `data`, and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```

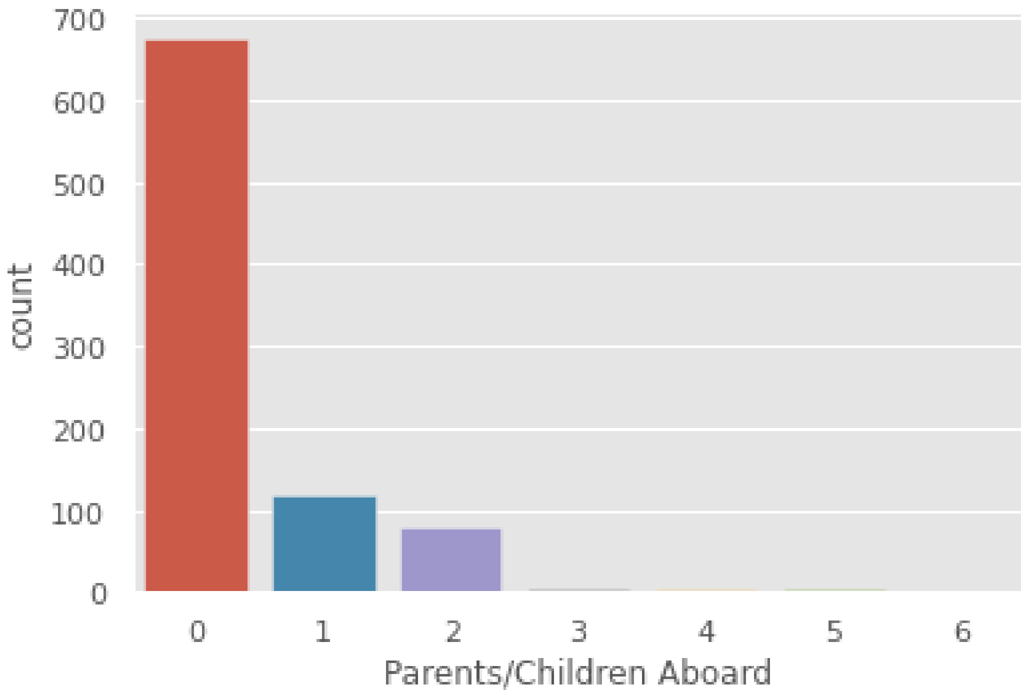
[148]: <AxesSubplot:xlabel='Sex', ylabel='count'>



[149]:
```
# TODO: use countplot to show the distribution of the number of parents/
↪children aboard
sns.countplot(titanic["Parents/Children Aboard"])
```

```
/opt/conda/lib/python3.9/site-packages/seaborn/_decorators.py:36: FutureWarning:
Pass the following variable as a keyword arg: x. From version 0.12, the only
valid positional argument will be `data`, and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```

[149]: `<AxesSubplot:xlabel='Parents/Children Aboard', ylabel='count'>`

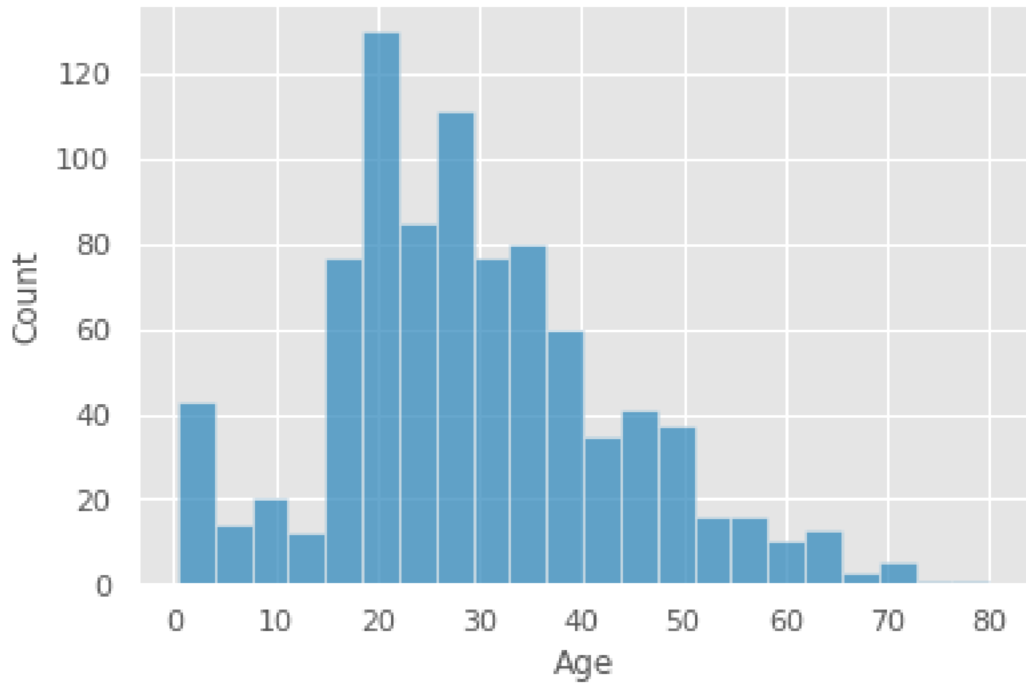


Summing up: male passengers were almost twice as female passengers, and the majority of people was traveling alone.

Let's conclude by looking at the distribution of age and fare paid. In the next cell, write a line of code to visualize the distribution of passenger ages. *Hint: since the age column contains numerical data, you shouldn't use `sns.countplot`, which is meant for categorical data. Is there another seaborn function you can use?*

[150]:
```
# TODO: visualize the distribution of passenger ages
sns.histplot(titanic['Age'])
```

[150]: <AxesSubplot:xlabel='Age', ylabel='Count'>



You should see that most travelers were in their 20s and 30s, and there were also many babies. What is the age of the youngest passenger? And the oldest?

[151]:
```python
min(titanic["Age"])
```

[151]: 0.42

[152]:
```python
# TODO: print the age of the oldest passenger
max_age = max(titanic["Age"])
```

[153]:
```python
# Validation tests: do not modify!
assert(get_hash(max_age) == '8ee5d21b272d43a875504f3e5845e141')
print("Test passed!")
```

```
Test passed!
```

[154]:
```python
# TODO: use distplot to show the fare distribution
sns.distplot(titanic['Fare'])
```

```
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:2619:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```

[154]: <AxesSubplot:xlabel='Fare', ylabel='Density'>

The fare distribution is really skewed: most people spent very little, but there is someone who spent even more than 500! Who are these people?

[155]:
```
# TODO: display the rows corresponding to passengers who spent more than 500
titanic[titanic['Fare'] > 500]
```

[155]:
```
     Survived  Pclass                               Name     Sex   Age  \
257         1       1                    Miss. Anna Ward  female  35.0
676         1       1  Mr. Thomas Drake Martinez Cardeza    male  36.0
733         1       1               Mr. Gustave J Lesurer    male  35.0

     Siblings/Spouses Aboard  Parents/Children Aboard      Fare
257                        0                        0  512.3292
676                        0                        1  512.3292
733                        0                        0  512.3292
```

4.3 Part 1.c: conditional probabilites

So far, we have looked at the variables separately, considering each marginal distribution. This tells us about them individually, but we're also interested in the relationships between variables. In particular, we want to find connections between survival and the other variables. For example, we might wonder whether people in first class had a higher probability of surviving than people in third class, or if females survived more or less often than males. To answer this type of question, we'll need to look at **conditional probabilities**. For a refresher on conditional probabilities, you can refer to Section 4.4 of the Data 140 textbook.

Let's start by looking at the conditional probabilities of survival given class type. If we denote the survival variable by S and the class variable by C, we are comparing:

$$\mathbb{P}(S = 1|C = 1) \quad \text{vs} \quad \mathbb{P}(S = 1|C = 2) \quad \text{vs} \quad \mathbb{P}(S = 1|C = 3)$$

We'll first compute $\mathbb{P}(S = 1|C = 1)$, the probability of surviving given that the passenger has a first-class ticket. To do so, remember the definition of conditional probability: for two events A and B,

$$P(A|B) = \frac{P(A \text{ and } B)}{P(B)}$$

In this case, we don't have to use Bayes' rule since we can compute the conditional probably directly using the definition.

[156]:
```python
# TODO: complete the following code to compute P(S=1|C=1)
firstclass_survival_prob = np.mean((titanic["Survived"] == 1) &␣
↪(titanic["Pclass"] == 1)) / np.mean(titanic["Pclass"] == 1)
firstclass_survival_prob
```

[156]: 0.6296296296296297

[157]:
```python
# Validation tests: do not modify!
assert get_hash(np.round(firstclass_survival_prob, 2)) ==␣
↪'b9c48c2d04160ef1ff72dba569292058'
print('Test passed!')
```

```
Test passed!
```

So, almost 63% of passengers traveling in first class survived: not bad!

What about people in the other two classes?

[158]:
```python
# TODO: compute the conditional probability of survival given that the␣
↪passenger is in class 2
secondclass_survival_prob = np.mean((titanic["Survived"] == 1) &␣
↪(titanic["Pclass"] == 2)) / np.mean(titanic["Pclass"] == 2)
secondclass_survival_prob
```

[158]: 0.47282608695652173

[159]:
```python
# Validation tests: do not modify!
assert get_hash(np.round(secondclass_survival_prob, 2)) ==␣
↪'7a7763d4618eb5666515c04dd53a3c08'
print('Test passed!')
```

```
Test passed!
```

[160]:
```
# TODO: compute the conditional probability of survival given that the␣
↪passenger is in class 3
thirdclass_survival_prob = np.mean((titanic["Survived"] == 1) &␣
↪(titanic["Pclass"] == 3)) / np.mean(titanic["Pclass"] == 3)
thirdclass_survival_prob
```

[160]:
```
0.24435318275154008
```

[161]:
```
# Validation tests: do not modify!
assert get_hash(np.round(thirdclass_survival_prob, 2)) ==␣
↪'5d6182b8169f820c3e247e91131138ea'
print('Test passed!')
```

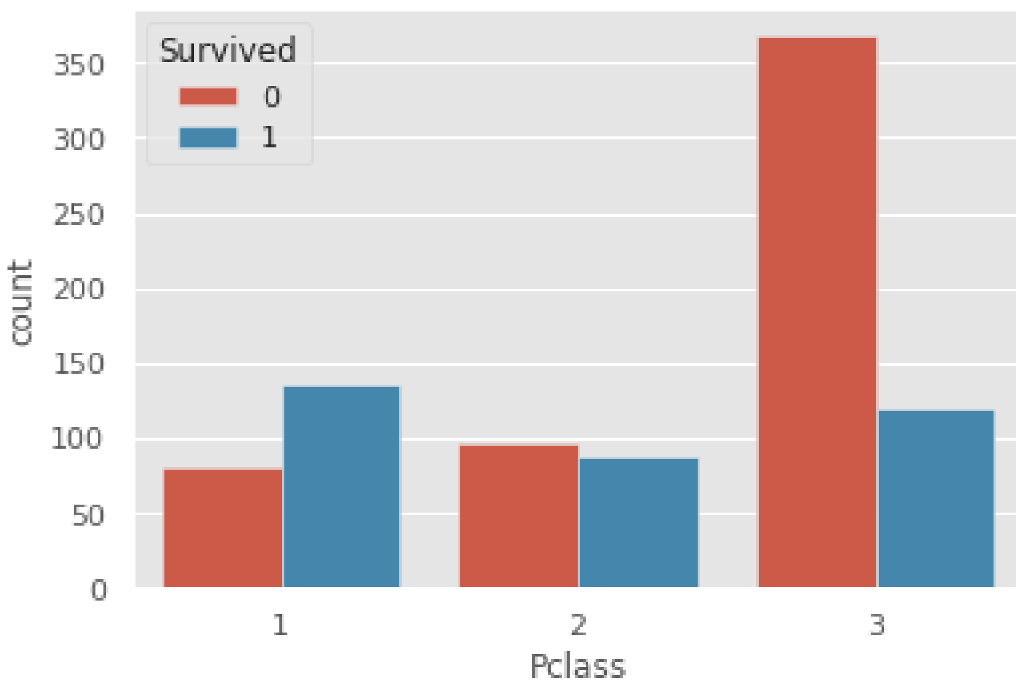
```
Test passed!
```

Survival probabilities were very different for the three class types! High class passenger had higher chances to survive.

We can also visualize survivals by class running the following cell.

[162]:
```
sns.countplot(x ='Pclass', hue = "Survived", data = titanic)
```

[162]:
```
<AxesSubplot:xlabel='Pclass', ylabel='count'>
```



You can keep on exploring and play with the other variables too, checking if there are groups with high or low probability of surviving. Try to condition on more than one variable (e.g. condition on both class type and gender).

4.4 Part 1.d: binary classification

In this last part, we're going to use logistic regression to predict passenger survival from the other variables. Here are the steps we'll follow:

1. Split the dataset into a training and test set
2. Fit a logistic regression model to the training set
3. Evaluate the performance on the test set using the language of binary decision-making that you saw in last week's lecture

We've done most of steps 1 and 2 for you, but we **strongly** encourage you to read through and understand all the code, since you'll need to do it yourself in future labs and homework assignments. Some resources that might be helpful:

- Data 100 textbook: logistic regression
- Data 100 textbook: one-hot encoding
- scikit-learn's documentation for `LabelEncoder` and `train_test_split` (you'll have to find these on your own).

Running the following cell will prepare the data and split it into training and test sets.

[163]:
```
titanic["Sex"] = LabelEncoder().fit_transform(titanic["Sex"]) # encode gender␣
 ↪variable (from string to integer)
titanic = titanic.drop("Name", axis = 1) # drop Name column

X = titanic.drop("Survived", axis = 1) # independent variables
y = titanic["Survived"] # dependent variable

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.30,␣
 ↪random_state = 3)
```

Now, we are ready to fit our logistic regression classifier, and to predict which passengers in the test data survived.

[164]:
```
lr = LogisticRegression(random_state = 0)
lr.fit(X_train, y_train) # fit the classifier to the training data
yhat = lr.predict(X_test) # predict survival for test data
```

Complete the code in the following cell to generate a new dataframe with two columns: the first containing our **decisions**, and the second containing the values in **reality**.

[165]:
```
output = pd.DataFrame()
output["Decisions"] = yhat
output["Reality"] = y_test.values
```

[166]:
```
output
```

[166]:
```
     Decisions  Reality
0            0        0
1            0        0
2            0        0
3            0        1
4            0        0
..         ...      ...
262          0        0
263          1        1
264          1        1
265          0        0
266          0        0

[267 rows x 2 columns]
```

[167]:
```
# Validation test: do not modify
assert get_hash(output['Decisions'].sum()) == 'e2ef524fbf3d9fe611d5a8e90fefdc9c'
assert get_hash((output['Decisions'] & output['Reality']).sum()) ==␣
↪'ad61ab143223efbc24c7d2583be69251'
print('Test passed!')
```

```
Test passed!
```

We're now interested in evaluating the performance of the logistic regression classifier: let's check how many false positive, false negative, true positive and true negative we have obtained.

[168]:
```
# TODO: compute the number of false positives (FP), false negatives (FN), true␣
↪positives (TP), and true negatives (TN)
FP = output[(output['Decisions'] == 1) & (output['Reality'] == 0)].shape[0]
FN = output[(output['Decisions'] == 0) & (output['Reality'] == 1)].shape[0]
TP = output[(output['Decisions'] == 1) & (output['Reality'] == 1)].shape[0]
TN = output[(output['Decisions'] == 0) & (output['Reality'] == 0)].shape[0]
print(FP, FN, TP, TN)
```

```
23 32 74 138
```

[169]:
```
# Validation test: do not modify
performance = [FP, FN, TP, TN]
hash_list = ['37693cfc748049e45d87b8c7d8b9aacd',
             '6364d3f0f495b6ab9dcf8d3b5c6e0b01',
             'ad61ab143223efbc24c7d2583be69251',
             '013d407166ec4fa56eb1e1f8cbe183b9']
assert all([get_hash(num) == hash_list[i] for (i, num) in␣
↪enumerate(performance)])
print("Test passed!!!")
```

```
Test passed!!!
```

Using the four numbers you found above, answer the two multiple choice questions below. *Hint: It* *might help to draw out the 2-by-2 table as in lecture.*

In which of the following two settings did the classifier perform better?

(a) People who actually survived

(b) People who actually did not survive

In which of the following two settings did the classifier perform better?

(a) People for whom the classifier predicted they survived

(b) People for whom the classifier predicted they did not survived

In the first setting, the classifier performs better amongst people who actually did NOT survive i.e. option b. This is because for the 106 people who actually did survive, 32 of them were predicted to die and 74 of them were predicted to survive i.e. the accuracy here was 69.8%. On the other hand, for the 161 people who actually did not survive, 138 of them were predicted to die while only 23 were predicted to survive i.e. the accuracy here was 85.7% which is much higher than 69.8%.

In the second setting, the classifier performs better amongst people whom the classifier predicted did not survive i.e. option b. For the 97 people for whom the classifier predicted survival, 74 of them actually survived i.e. an accuracy of 76.2%. For the 170 people for whom the classifier predicted non-survival, there was 138 of them who actually did not survive, i.e. an accuracy of 81.1% which is much higher than 76.2%.

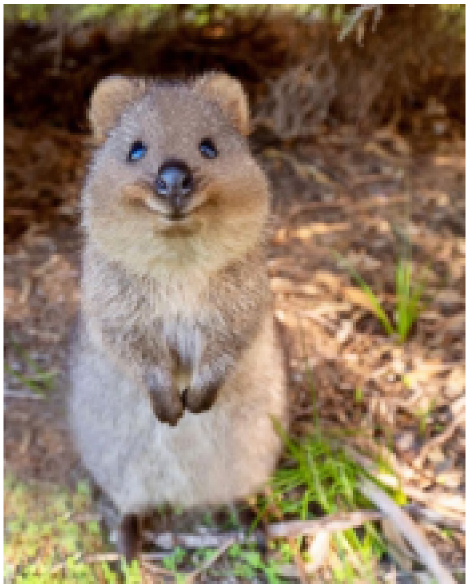
4.5 Final tests

If all the tests below pass you can assume you have successfuly completed the testable parts of the lab. Don't worry about understanding the code below; just make sure no asserts fail.

[170]:
```
tests = [survival_prob, max_age, np.round(firstclass_survival_prob, 2),
         np.round(secondclass_survival_prob, 2), np.
 ↪round(thirdclass_survival_prob, 2),
         output['Decisions'].sum(), (output['Decisions'] & output['Reality']).
 ↪sum(),
         FP, FN, TP, TN]
hash_list = ['4fe6b7a204127acbec520ac2997133dc',
             '8ee5d21b272d43a875504f3e5845e141',
             'b9c48c2d04160ef1ff72dba569292058',
             '7a7763d4618eb5666515c04dd53a3c08',
             '5d6182b8169f820c3e247e91131138ea',
             'e2ef524fbf3d9fe611d5a8e90fefdc9c',
             'ad61ab143223efbc24c7d2583be69251',
             '37693cfc748049e45d87b8c7d8b9aacd',
             '6364d3f0f495b6ab9dcf8d3b5c6e0b01',
             'ad61ab143223efbc24c7d2583be69251',
             '013d407166ec4fa56eb1e1f8cbe183b9']
assert all([get_hash(t) == hash_list[i] for (i, t) in enumerate(tests)])
print("All tests passed! You are awesome!!!")
```

```
All tests passed! You are awesome!!!
```

[171]:
```python
import matplotlib.image as mpimg
img = mpimg.imread('cute_quokka.jpg')
imgplot = plt.imshow(img)
imgplot.axes.get_xaxis().set_visible(False)
imgplot.axes.get_yaxis().set_visible(False)
plt.show()
print('Congrats! You made it to the end of the lab!!!')
```



```
Congrats! You made it to the end of the lab!!!
```