Homework 6

Fill in your name and the names of any students who helped you below.

> I affirm that I personally wrote the text, code, and comments in this homework assignment.
>
> I received help from [Ching-Yi Wang and Yanjun Jiang] who gave me suggestions on [problems].

- [Yiting Zhang]

Comments/docstrings are required for problems 1 and 3. They are not required for problems 2.

Problem 0

It is highly recommended that you work with your group to fully complete the recent Discussion assignments related to the Palmer Penguins data set, as these will directly help with your project.

Problem 1

Gapminder is a foundation, based on Sweden, that aims to enhance basic awareness of basic facts about the socioeconomic global world. As part of their efforts, they collect detailed statistics on life expectancy, population, and GDP, sometimes going back over many years.

Here, we'll work with an excerpt of the Gapminder data. This excerpt has been packaged up and made available via Jenny Bryan's `gapminder` repo on Github.

Run the code below to retrieve the data and take a look. As usual, you can also directly download the data by pasting the url into your browswer, saving the file, and reading it in locally via `pandas.read_csv`.

```
In [1]:    import pandas as pd

url = "https://philchodrow.github.io/PIC16A/datasets/gapminder.csv"
gapminder = pd.read_csv(url)
gapminder
```

Out[1]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	Africa	1987	62.351	9216418	706.157306
1700	Zimbabwe	Africa	1992	60.377	10704340	693.420786
1701	Zimbabwe	Africa	1997	46.809	11404948	792.449960
1702	Zimbabwe	Africa	2002	39.989	11926563	672.038623
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

1704 rows × 6 columns

Use the `gapminder` data to create the folowing visualization. Here, each trendline corresponds to a distinct country, which in turn lies on the stated continent.
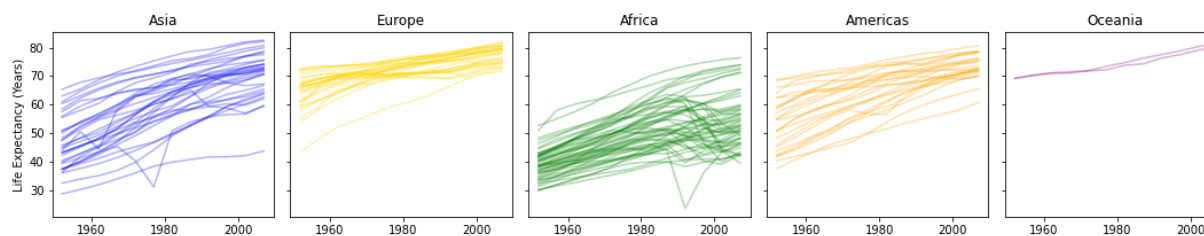
You should achieve this result **without for-loops**, and also without manually creating the plot on each axis. You may find it useful to define additional data structures, such as dictionaries, that assign colors or axis indices to each continent. Feel free to modify aesthetic details of the plots, such as the colors.

Hint: Try `df.groupby(["country"]).apply(f)`, where `f` is some very smart function that you have figured out.

In [2]:
```python
import pandas as pd
import numpy as np
import urllib
from matplotlib import pyplot as plt
```

In [3]:
```python
# your solution here

fig,ax=plt.subplots(1,5,figsize=(22,3),sharey=True)


#create a dictionary to assign axis indices
col={
    "Asia":0,
    "Europe":1,
    "Africa":2,
    "Americas":3,
    "Oceania":4
}

#create a dictionary to assign colors
colors={
    0:"Blue",
    1:"Gold",
```

```
        2:"Green",
        3:"Orange",
        4: "Purple"

    }

    def f(data, **kwargs):
        """
        This function plot each countries data point from Gapminder onto the plot, vategorized by countinents.
        By referencing the dictionaries of ais indices and colors for each continents,
        we can graph each data point using this function.
        """
        #iloc locates the specific row within the Gapminder dataset column "continent"
        i = col[data['continent'].iloc[0]]
        #plot life expectancy against year and each continents are color coded.
        ax[i].plot(data.year, data.lifeExp, color=colors[i], alpha=0.5, **kwargs)
        #set x axis title
        ax[i].set(title = data['continent'].iloc[0] )
        #set y label
        ax[0].set(ylabel = "Life Expectancy (Years)")

    gapminder.groupby(["country"]).apply(f)
```

Out[3]: —
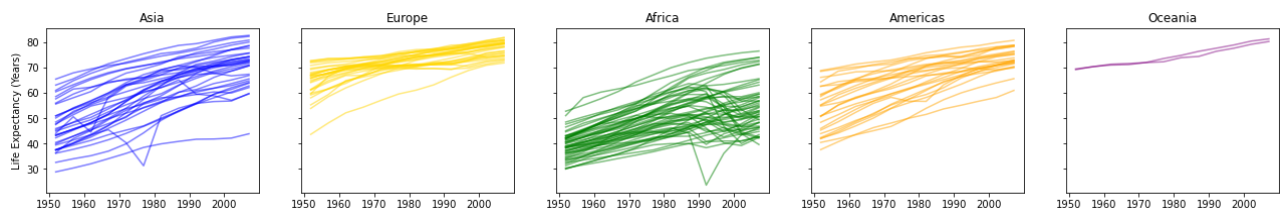
In [4]: `gapminder["continent"].iloc[500]`

Out[4]: `'Africa'`

Problem 2

In this problem, you will create several attractive visualizations of the `gapminder` data set using the Seaborn package. Seaborn is a high-level interface to Matplotlib specially designed for working with Pandas data frames. Seaborn makes it relatively easy to create complex, multifaceted data graphics. The drawback is that it can sometimes be more difficult to exercise control over figure details, such as axis labels or figure size.

Part of the learning objective for this problem is for you to **practice learning from documentation and examples**. I have given you some pointers in each problem, but it will be up to you to fill in the blanks.

Import the `seaborn` package with the line `import seaborn as sns`. Then, run the code block below to create a scatterplot of `gdpPercap` (gross domestic product per capita) against `lifeExp` (life expectancy), with different colors for each continent.

In [5]:
```python
from matplotlib import pyplot as plt
import seaborn as sns

fgrid = sns.relplot(x = "gdpPercap",
                    y = "lifeExp",
                    hue = "continent",
                    data = gapminder)
```
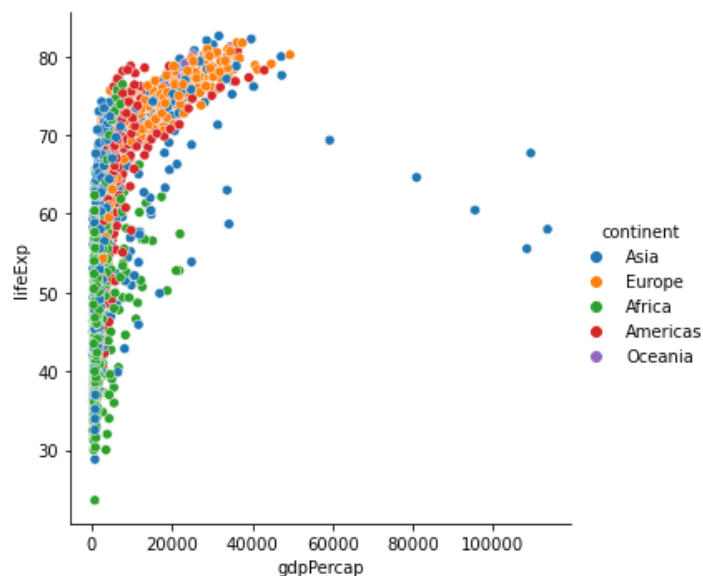
The result of the above code is that `fgrid` is now a `FacetGrid` object (we'll see why it's called that in a minute). However, under the hood, we are still dealing with Matplotlib. To retrieve the individual plotting axis, we can access it from the `fgrid.axes` attribute, which is a 2d Numpy array containing the axes. For example, to make the horizontal axis logarithmic and labels, we can do this:

```
In [6]:  fgrid = sns.relplot(x = "gdpPercap",
                             y = "lifeExp",
                             hue = "continent",
                             data = gapminder)

         fgrid.axes[0][0].semilogx()

         labels = fgrid.axes[0][0].set(xlabel = "GDP per capita",
                                       ylabel = "Life Expectancy",
                                       title  = "Impact of Economic Output on Life Expectancy")
```
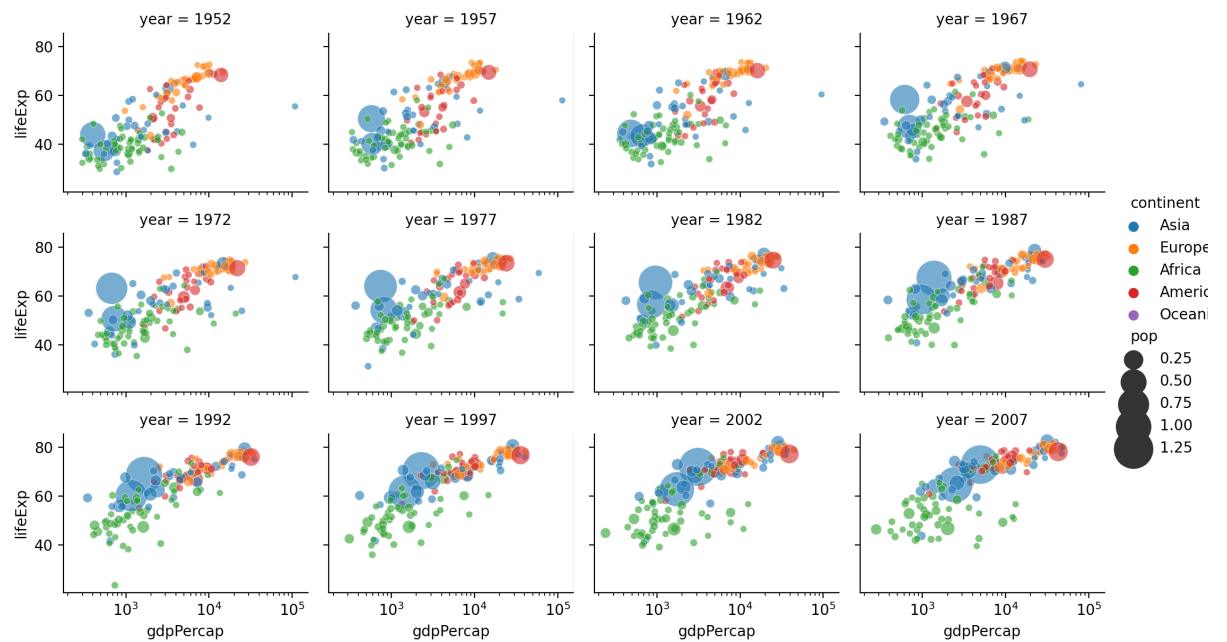

Not bad! Note that seaborn has effectively looped over the different continents, without us needing to use `for` -loops or `apply` .

Part A

Create the following plot using Seaborn:

In this figure, each point represents a country, and the size of the point corresponds to the population.

In this part and the next, it is not necessary that your plot *precisely* match mine -- feel free to modify the sizes of labels, change the color scheme, etc.

Hints

- You should use `sns.relplot` to create the figure.
- Look in the Seaborn Gallery for examples illustrating how to create a multi-panel figure without using `plt.subplots()` .
- The size of the figure is controlled by the `height` and `aspect` ratio arguments to `relplot` . I used `height = 2` and `aspect = 1.3` , but other choices are also fine.
- When using column wrapping to create the `3 x 4` axis array, the resulting `fgrid.axes` array is no longer 2-dimensional, but 1-dimensional.

In [7]:
```python
fgrid=sns.relplot(data=gapminder,
                  x="gdpPercap",
                  y="lifeExp",
                  hue="continent",
                  size="pop",
                  alpha=0.6,
                  col="year",
                  col_wrap=4,
                  height=2,
                  aspect=1.3,
                  sizes=(20,800)) #determines how sizes are chosen when size is used.

for ax in fgrid.axes.flatten():  #flatten() reshape everything into a one dimensional array
    ax.semilogx() #put the x axis in log scale
```
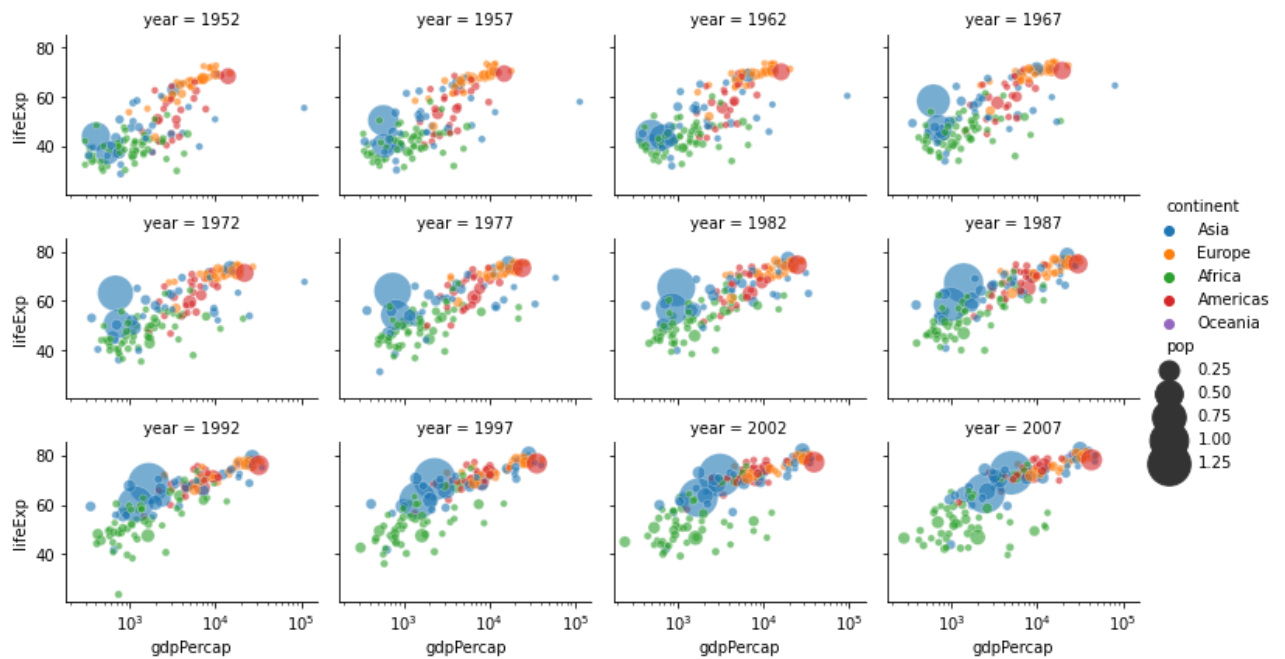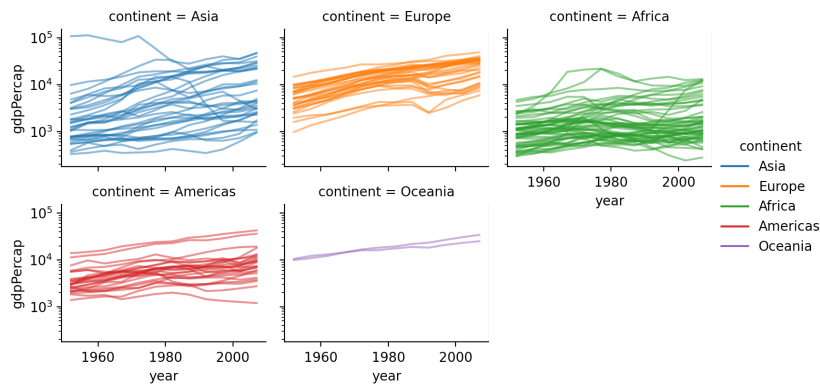
Part B

Create the following plot using Seaborn:

Hints

- The easiest way is to again use `relplot`. This function defaults to producing scatterplots, but if you check the gallery you'll find examples of how to specify keywords that create line charts instead.

```
In [8]: fgrid=sns.relplot(data=gapminder,
                          y="gdpPercap",
                          x="year",
                          hue="continent",
                          alpha=0.6,
                          col="continent",
                          col_wrap=3,
                          height=2,
                          aspect=1.3,
                          kind="line",
                          units="country", estimator=None)

        for ax in fgrid.axes.flatten():  #flatten() reshape everything into a one dimensional array
            ax.semilogy() #put the y axis in log scale
```
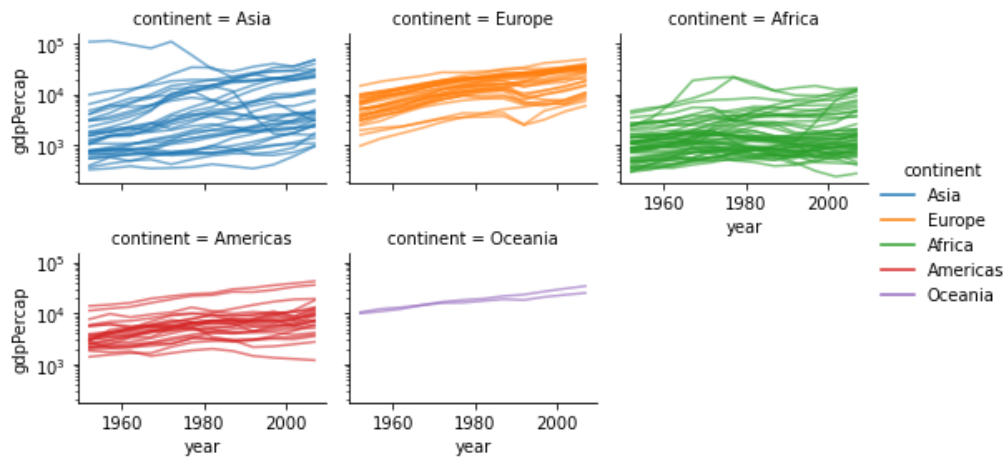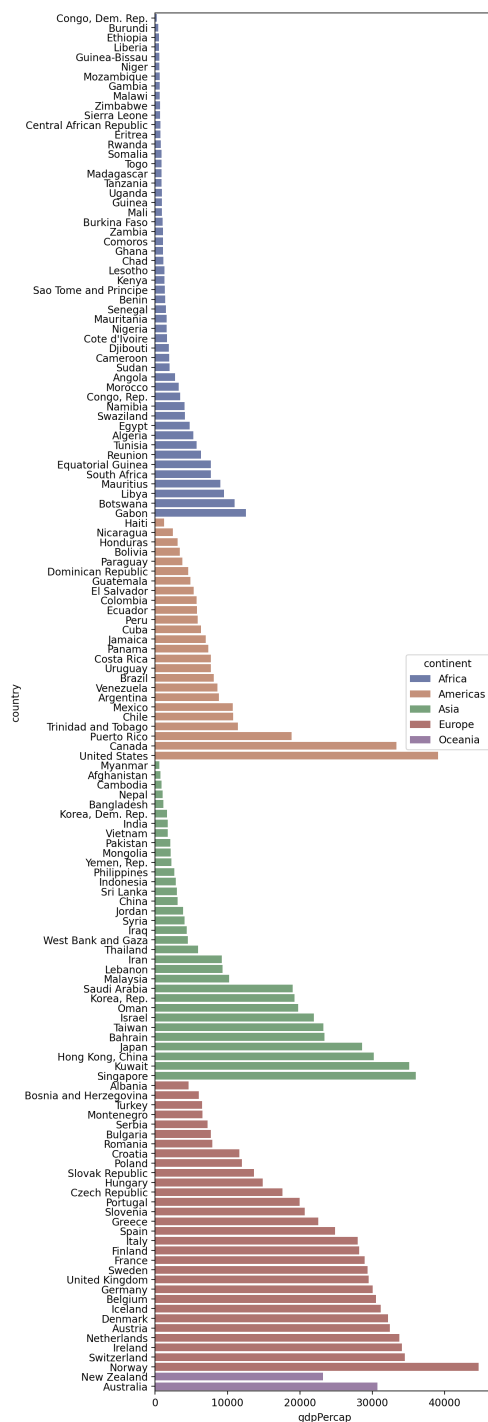
Part C

Create the following (very long) plot using Seaborn:

Hints

- Use `sns.barplot`.
- Pick a single year of the `gapminder` data frame to show. In the example, I showed data from 2002 only.
- To sort the bars in ascending order by continent, you should sort the data frame that you pass to `sns.barplot`.
- The argument `dodge = False` is a helpful keyword that will keep your bars centered.
- I used a different color palette just to shake things up a little.

In [9]:
```python
#create a subset of gaominder tqt have only a single year of data
sub_gapminder=gapminder[gapminder["year"]==1987]

#create an empty plot
fig,ax=plt.subplots(figsize=(6,25))
#sort gdpPercap by continent in ascending order
```
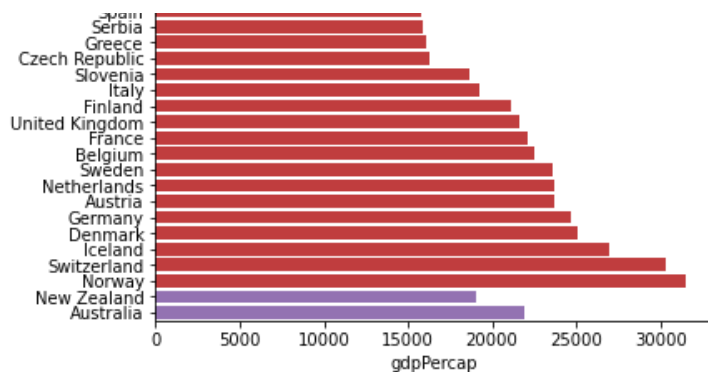
```python
df=sub_gapminder.sort_values(["continent","gdpPercap"],ascending=True)
#plot using sns.barplot
sns.barplot(data=df,x="gdpPercap",y="country", hue="continent", dodge=False)
```

Out[9]:   `<AxesSubplot:xlabel='gdpPercap', ylabel='country'>`

```python
df=sub_gapminder.sort_values(["continent","gdpPercap"],ascending=True)
#plot using sns.barplot
sns.barplot(data=df,x="gdpPercap",y="country", hue="continent", dodge=False)
```
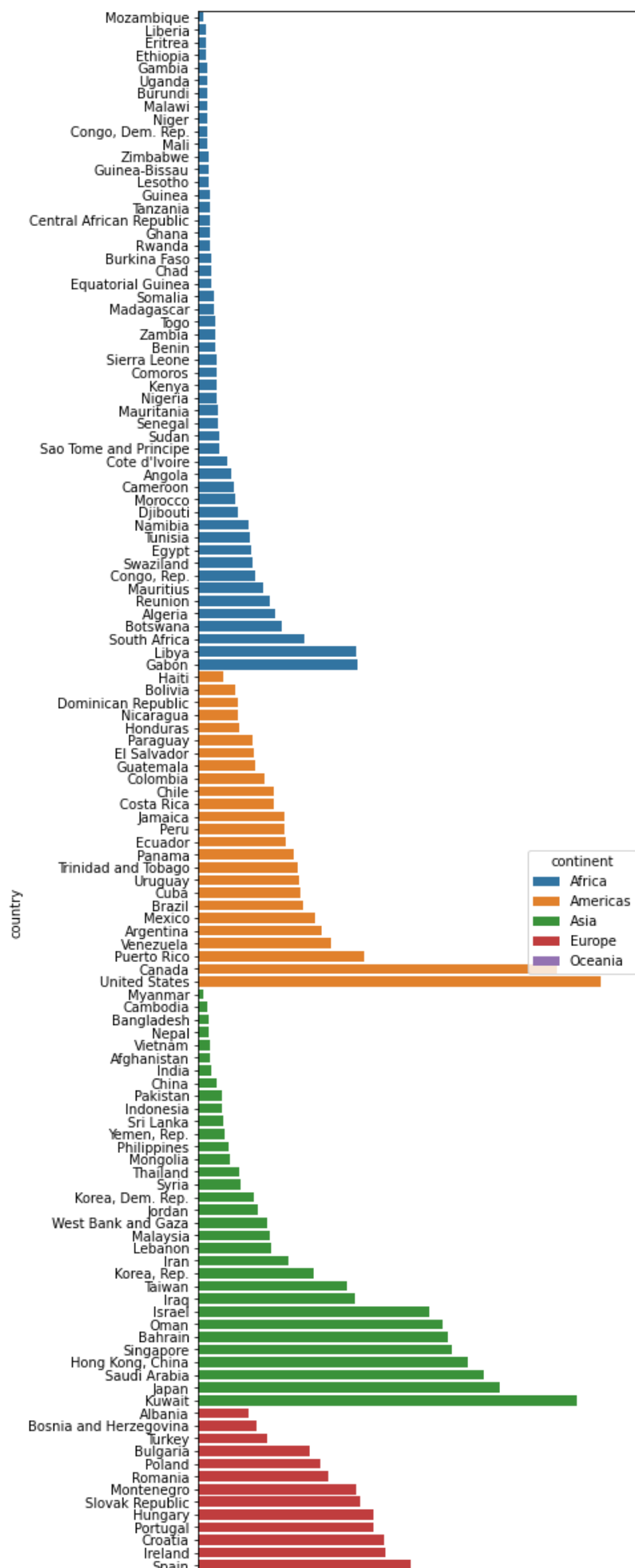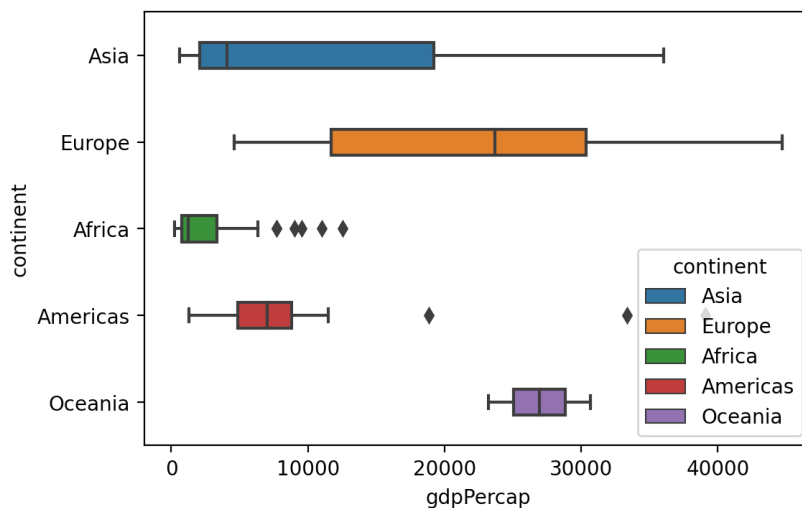
Part D

Create the one last plot using Seaborn:

Hints

- This is a boxplot.
- You should again pick a single year of the `gapminder` data frame to show. In the example, I show data from 2002 only.
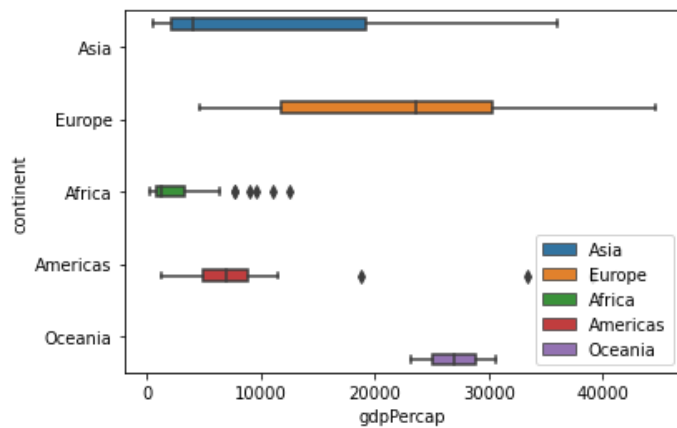
In [10]:
```python
#create a subset of gaominder tqt have only a single year of data
sub_gapminder=gapminder[gapminder["year"]==2002]

#create an empty plot
fig,ax=plt.subplots()

df=sub_gapminder
#plot using sns.barplot
sns.boxplot(data=df,x="gdpPercap",y="continent", hue="continent")

#change the lgend to lower right
plt.legend(loc="lower right")
```

Out[10]:   `<matplotlib.legend.Legend at 0x1f5848a58e0>`

Problem 3

In our first lecture on machine learning, we did linear regression "by hand." In this problem, we will similarly perform logistic regression "by hand." This homework problem is closely parallel to the lecture, and so you might want to have the notes handy.

Although logistic regression is a relatively simple model, it is a foundation for many modern deep neural nets. Additioanlly, logistic regression itself is still often used in applied contexts even now.

> "When we raise money it's AI, when we hire it's machine learning, and when we do the work it's logistic regression."
>
> (I'm not sure who came up with this but it's a gem 💎)
>
> — Dr. Daniela Witten (@daniela_witten) September 26, 2019

Whereas linear regression is often used to measure arbitrary quantities like GDP or happiness scores, logistic regression is primarily used to estimate *probabilities*. For example, we might use logistic regression to estimate the probability of a passenger surviving the Titanic crash, a borrower defaulting on a loan, or an email being spam.

For concreteness, let's say that we are considering the latter case. Suppose that we wish to model the probability that an email is spam as a function of the proportion of flag words (like "investment", "capital", "bank", "account", etc.) in the email's body text. Call this proportion x. x is then a variable between 0 and 1.

In logistic regression, we suppose that the probability p that an email is spam has the form

$$p = \frac{1}{1 + e^{-ax-b}} \, ,$$

where a and b are again parameters. Let's see how this looks.

```
# run this block

import numpy as np
from matplotlib import pyplot as plt

n_points = 1000

a = 10
b = -5

x = np.sort(np.random.rand(n_points))
p = 1/(1+np.exp(-a*x - b))

fig, ax = plt.subplots(1)
ax.plot(x, p, color = "black")
```
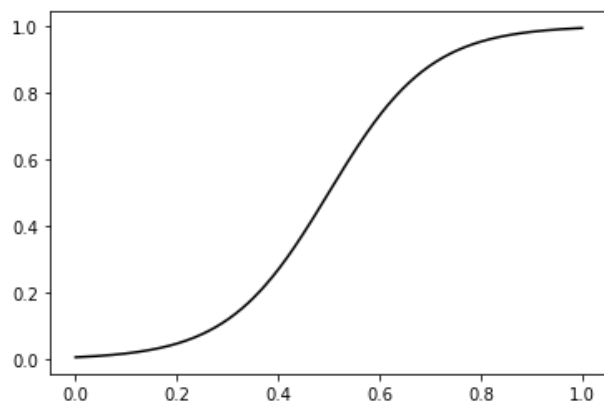
Out[119]:   `[<matplotlib.lines.Line2D at 0x1f58fe57fd0>]`

As usual, in practice we don't have access to the true function telling us the probability that an email is spam. Instead, we have access to data telling us whether or not the email really IS spam. We can model this situation by flipping a biased coin for each email, with the probability of heads determined by the modeled probability.

In [120...
```python
# run this block
y = 1.0*(np.random.rand(n_points) < p)
```

A value of 1 indicates that the email is indeed spam, while a value of 0 indicates that the email is not spam.

In [121...
```python
# run this block
ax.scatter(x, y,  alpha = 0.5)
fig
```

Out[121]:
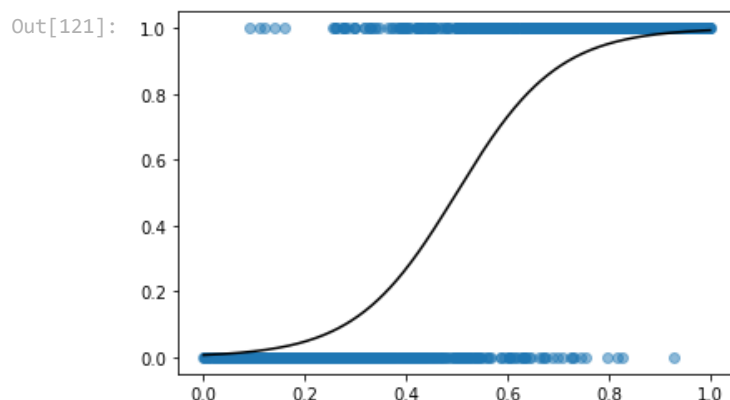
Notice that there are more spam emails where the model gives a high probability, and fewer where the model gives a lower probability. However, there may be some non-spam emails with even high probability -- sometimes we get legitimate emails about bank accounts, investments, etc.

Of course, we don't have access to the true model, so our practical situation looks more like this:

In [122...
```python
# run this block
fig, ax = plt.subplots(1)
ax.scatter(x, y, alpha = 0.5)
```

Out[122]:   `<matplotlib.collections.PathCollection at 0x1f58fead070>`
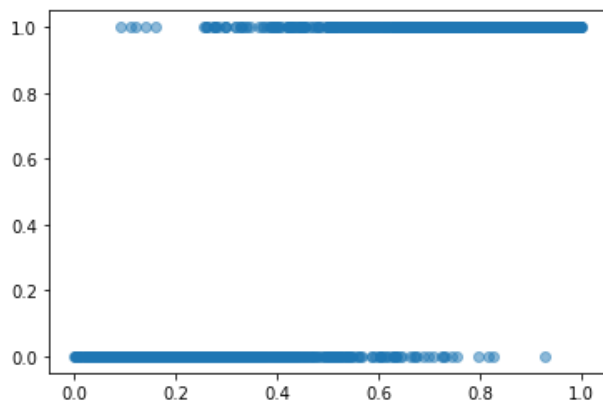
We would like to use logistic regression to try to recover something close to the true model.

Part A

Write the model function `f` . The arguments of `f` should be the predictor variables `x` and the parameters `a` and `b` . The output of `f` should be the spam probabilities under the logistic model (see equation above) for these data. Use `numpy` tools, without `for` -loops. If you scan the above code carefully, you'll observe that most of this code is already written for you.

This is a simple function, but **please add a short docstring indicating** what kinds of input it accepts and how to interpret the output.

Comments are necessary only if your function body exceeds one line.

```
In [72]:  # your solution

def f(x,a,b):
    """
    This function output the spam probabilities under the logistic regression model
    x: predictor variables, a list of values is acceptable as input
    a,b: parameters, a single number (int, float or double) is acceptable as input
    """
    return 1/(1+np.exp(-a*x - b))
```

Part B

Plot 10 candidate models against the data, using randomly chosen values of `a` between 5 and 15 and randomly chosen values of `b` between -2.5 and -7.5. Your plot should resemble in certain respects the third plot in these lecture notes.
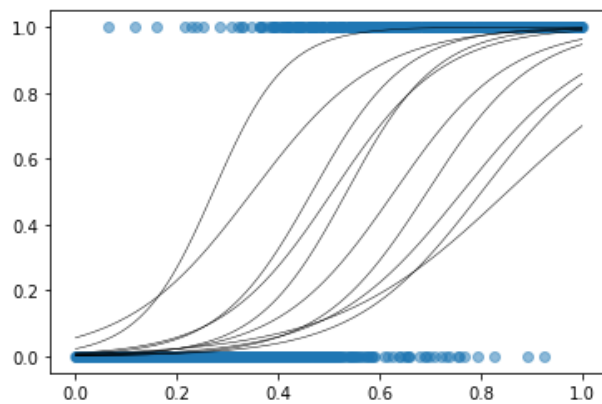
Comments are not necessary in this part.

```
In [73]:  # your solution here

fig, ax = plt.subplots(1)

#create a scatter plot of x and y
ax.scatter(x,y, alpha=0.5)

for i in range(10): #range(10) becuase there are 10 candidates model
    # a is randomly between 5 and 15
    # b is randomly between -2.5 and -7.5
    a = 10*np.random.rand()+5
    b = -5*np.random.rand()-2.5
    #plot the regression lines
    ax.plot(x, f(x, a, b), color = "black", linewidth = .5)
```

Part C

The *loss function* most commonly used in logistic regression is the *negative cross-entropy*. The negative cross-entropy of the `i` th observation is

$$- [y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i)]$$

where $y_i \in \{0, 1\}$ is the `i` th entry of the target data and $\hat{p}_i$ is the model's estimated probability that $y_i = 1$. The negative cross-entropy of the entire data set is the sum of the negative cross-entropies for each individual observation.

Write a function that computes the negative cross entropy as a function of `x`, `y`, `a`, and `b`. This can be done in no more than two lines using `numpy`, without `for`-loops. Don't forget which logarithm is #BestLogarithm.

As in Part B, please write a short docstring describing what your function does and what inputs it accepts. Comments are necessary only if your function body exceeds two lines.

```
In [74]:   # your solution here

def loss(x,y,a,b):
    """
    This function computes the negative cross entropy as a function of x, y, a, and b.

    The negative cross-entropy of the `i`th observation is
    $$-\left[y_i \log \hat{p}_i + (1-y_i)\log(1-\hat{p}_i)\right]$$
    where $y_i \in \{0,1\}$ is the `i`th entry of the target data
    and $\hat{p}_i$ is the model's estimated probability that $y_i = 1$.
    The negative cross-entropy of the entire data set is the sum of the
    negative cross-entropies for each individual observation.

    ----------------------------------------------------------------
    x: predictor variables, a list of values is acceptable as input
    y: $y_i \in \{0,1\}$ is the `i`th entry of the target data
    a,b: parameters, a single number (int, float or double) is acceptable as input
    """
    return np.sum(-(y*np.log(f(x,a,b))+(1-y)*np.log(1-f(x,a,b))))
```

Part D

On a single axis, plot 100 distinct models (using the code that you wrote in) in Part B. Highlight the one with the lowest negative cross entropy in a different color -- say, red. Compare the best values of `a` and `b` that you found to the true values, which were `a = 10` and `b = -5`. Are you close?

The plot you produce should resemble, in some respects, the fifth plot in these lecture notes.

It is not necessary to write comments in this part.

```
In [131…   # your solution here
```

```python
x = np.sort(np.random.rand(n_points))
p = 1/(1+np.exp(-a*x - b))
y = 1.0*(np.random.rand(n_points) < p)

fig, ax = plt.subplots(1)
ax.scatter(x,y)

best_a, best_b = 0, 0
best_cross_entropy = np.inf
for i in range(100):
    a = 10*np.random.rand()+5
    b = -5*np.random.rand()-2.5

    cross_entropy = loss(x, y, a, b)
    if cross_entropy < best_cross_entropy:
        best_cross_entropy = cross_entropy
        best_a, best_b = a,b
    y = f(x, a, b)
    ax.plot(x, y , color = "black", alpha = 0.5, linewidth = .2)

best_preds = f(x, best_a, best_b)
ax.plot(x, best_preds, color = "red")
```
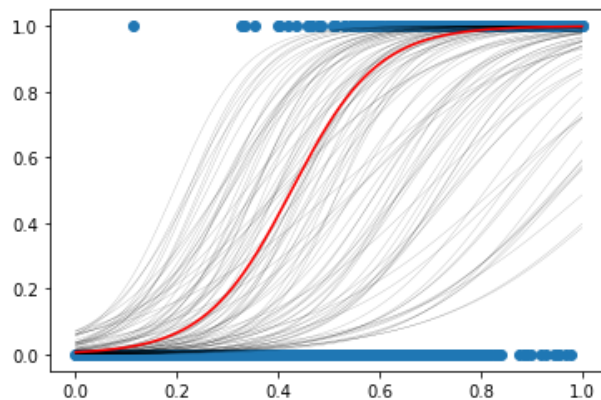
Out[131]: `[<matplotlib.lines.Line2D at 0x1f592684310>]`


```python
# true parameters were a = 10, b = -5
best_a, best_b
# the parameters are pretty close to the tru parameters
```

Out[132]: `(11.749839039396786, -5.005023652266239)`

It is not required to use `scipy.optimize` to more accurately estimate `a` and `b` for this homework assignment, but you are free to do so if you wish. You may then use the optimal estimates in the following part.

Part E

In classification tasks, we evaluate not just the standard loss function, but also the *accuracy* -- how often does the model correctly classify the data? Let's say that the model classifies an email as spam according to the following rule:

1. If $\hat{p}_i$ (the model probability plotted above) is larger than c, classify the email as spam.
2. If $\hat{p}_i$ is less than or equal to c, classify the email as not-spam.

Write a function called `positive_rates` which accepts the following arguments:

1. The data, `x` and `y`.
2. The best parameters `best_a` and `best_b`.
3. A threshold `c` between 0 and 1.

This function should output two numbers. The first of these is *false positive rate*: the proportion of non-spam emails that the model incorrectly labels as spam. The second is the *true positive rate*: the proportion of spam emails that the model

correctly labels as spam.

For example:

```
 positive_rates(x, y, best_a, best_b, c = 0.5)
```

```
    (0.05058365758754864, 0.7469135802469136)
```

Note: due to randomization, your numerical output may be different.

Please write a descriptive docstring for your function. Comments are necessary only if your function body exceeds five lines.

In [135...
```python
# your solution here

def positive_rates(x, y, best_a, best_b, c):
    """
    This functin evaluates the accuracy -- how often does the model correctly classify the data?

    1. If p  (the model probability plotted above) is larger than c, classify the email as spam.
    2. If p is less than or equal to c, classify the email as not-spam.

    1. The data, `x` and `y`.
    2. The best parameters `best_a` and `best_b`.
    3. A threshold `c` between 0 and 1. Input needs to be a single number between 0 and 1.

    Function Output:
    The first of these is *false positive rate*: the proportion of non-spam emails
    that the model incorrectly labels as spam.
    The second is the *true positive rate*: the proportion of spam emails
    that the model correctly labels as spam.
    """
    p= f(x, best_a, best_b)
    y = 1.0*(np.random.rand(n_points) < p)
    #extract the values that is not spam but the model incorectly labels as spam
    false_positive= p[(p>c) & (y==0)]
    #extract the values that is spam and the model correctly labels as spam
    true_positive= p[(p>c) & (y==1)]
    #calculate the proportion of false positive data relative to non-spam values
    false_positive_prop=len(false_positive)/len(y[y==0])
    #calculate the proportion of true positive data relative to spam values
    true_positive_prop=len(true_positive)/len(y[y==1])
    return false_positive_prop,true_positive_prop
```

In [138...
```python
# demonstrate your function here

positive_rates(x, y, best_a, best_b, c = 0.5)
```

Out[138]:
```
(0.10755148741418764, 0.9005328596802842)
```

Part F

Plot the *receiver operating characteristic* (ROC) curve for the logistic model with parameters `best_a` and `best_b` . The ROC curve is the plot of the `false_positive` rate (on the horizontal axis) against the `true_positive` rate (on the vertical axis) as the threshold `c` is allowed to vary. Additionally, plot a diagonal line ("the line of equality") between the points (0,0) and (1,1). Your ROC curve should lie noticeably above the line of equality. Plot your curves in different colors and add a legend to help your reader understand the plot.

It is ok to use `for` -loops and list comprehensions in this part.

Your result should look something like this, although it won't be exactly the same due to randomness.
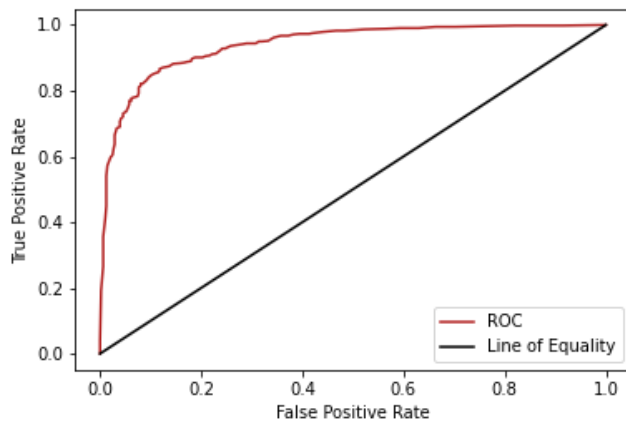
```
In [145… # plot code here

         fig, ax = plt.subplots(1)

         #create two empty lists to store values
         false_positive=[]
         true_positive=[]

         for i in range(len(y)):
             #set c equals to random values between 0 and 1
             p,q=positive_rates(x, y, best_a, best_b, c = np.random.rand())
             #append the values to the empty lists that created previously
             false_positive.append(p)
             true_positive.append(q)

         #sort the values so points can be plot accoordingly
         false_positive=np.sort(false_positive)
         true_positive=np.sort(true_positive)

         # The ROC curve is the plot of the false_positive rate (on the horizontal axis)
         #against the true_positive rate (on the vertical axis)
         ax.plot(false_positive,true_positive, color = "red", label = "ROC")
         #add a line of equality y=x
         ax.plot([0,1], [0, 1], color = "black", label = "Line of Equality")
         #set axis labels for clarification
         ax.set(xlabel="False Positive Rate", ylabel="True Positive Rate")
         #add a legend
         plt.legend()
```
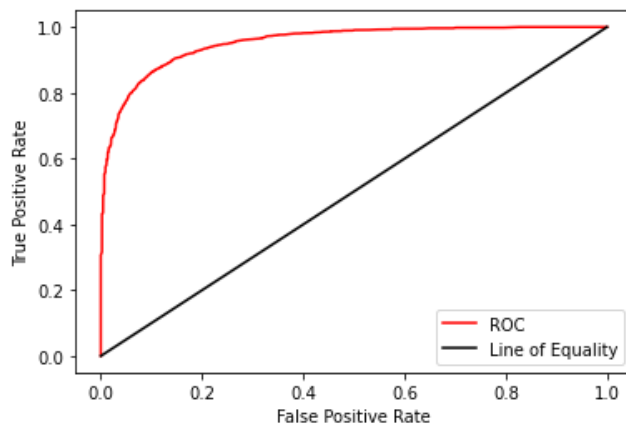
Out[145]:   <matplotlib.legend.Legend at 0x1f592d778e0>

Generally speaking, a "good" classifier is one that can reach the closest to the top-left corner of the ROC diagram. This is a classifier that can achieve a high rate of true positives, while keeping a low rate of false positives. There are various ways to measure how "good" an ROC curve is, which are beyond our present scope.