COSC 130 – HW 07 Instructions

General Instructions

Create a new notebook. and complete Parts 1 – 8 described below.

Any set of instructions you see in this document with an orange bar to the left will indicate a place where you should create a markdown cell. For each new part of this assignment, create a markdown cell that indicates the title of that part as a level 2 header.

Any set of instructions you see with a blue bar to the left will provide instructions for creating a single code cell.

Read the problem instructions carefully. One of the goals of this assignment is to have you practice working with numpy. As such, you will occasionally be asked to avoid using loops, and instead use efficient tools from numpy.

Assignment Header and Import Statements

Create a markdown cell with a level 1 header that reads: "COSC 130 - Homework 07". Add your name as a level 3 header.

Import the following packages: **`numpy`**, **`math`**, and **`matplotlib.pyplot`**. No other packages should be used in this project.

Part 1: Sample Mean and Variance

In this part, you will be asked to use NumPy to calculate sample mean and variance of a data set. You will do this in two ways:

1. First using numpy to perform the calculations indicated by the formulas for these values, and
2. Then using built-in NumPy functions for the mean and variance.

We start with a discussion of the necessary formulas.

Given a collection of n numerical observations $x_1, x_2, \dots, x_n$, the sample mean $\bar{x}$ and sample variance s^2 of the observations are given by the following formulas:

- Sample mean: $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{1}{n}(x_1 + x_2 + \cdots + x_n)$
- Sample variance: $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 = \frac{1}{n-1}[(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \cdots + (x_n - \bar{x})^2]$

Create a code cell to perform the following steps:

1. Create an array **`x`** containing the following integers: `10, 16, 26, 12, 17, 22, 14, 12, 21, 16`
2. Store the length of this array in a variable **`n`**.
3. Calculate the sample mean, storing the result in **`mean`**. You may use **`np.sum()`** in your calculation.
4. Calculate an array named **`diff`** that stores the differences between each value in **`x`** and the mean. That is to that that **`diff`** should contain values $(x_i - \bar{x})$ for each $i = 1, \dots, n$.
5. Use **`diff`**, **`n`**, and **`np.sum()`** to find the sample variance. Store the result in a variable named **`var`**.
6. Print your results in the format shown below. Make sure that the numerical outputs are aligned with each other on the left.

```
Sample Mean:     xxxxx
Sample Variance: xxxxx
```

Your code in Part 1 should not include any loops.

The example above shows how to use numpy to quickly perform complex calculations that would typically require loops. However, numpy actually has built-in functions that we can use to calculate the mean and standard deviation.

Use the functions **`np.mean()`** and **`np.var()`** to calculate the sample mean and sample variance of **`x`**, storing the result in variables named **`mean_np`** and **`var_np`**. In order to get the sample variance (as opposed to the population variance) you will need to set the **ddof** parameter of **`np.var()`** to 1. Print the results in the same format as above

Part 2: Scoring a Regression Model

Assume that we wish to predict estimate the value of a variable y based on the values of other input variables. If y is a continuous numerical variable, then a function used to generate such estimations is called a **regression model**. When we know the true y values for a set of observations, we can evaluate the performance of our model by comparing the estimated y values to the actual values. There are several metrics that can be used for scoring a model in this way, but one of the most common is to calculate the model's **sum of squared errors (SSE)** score on the dataset. We will explain this metric in more detail below.

Assume that $y_1, y_2, \ldots, y_n$ are actual y values, and that $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n$ are estimated y values generated by some regression model (we will see exactly how such estimations might be generated later on in this course). The model's **SSE** score on the data set is calculated as follows:

$$SSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

The SSE score provides use with a tool for measuring the performance of a regression model. Models that produce estimates with smaller errors will have smaller SSE scores.

Write a function named **`find_sse()`** that accepts two parameters, **`true_y`** and **`pred_y`**. The parameter **`true_y`** is expected to be an array of observed y values while **`pred_y`** is expected to be an array of predicted y values generated by a regression model. The function should return the SSE score for the regression model, as calculated on this set of observations. Do not call the function in this cell. **Your function should use numpy operations, and should NOT involve any loops.**

When we have several different regression models that we want to compare, we will often use the SSE score to compare the models. We generally prefer models with lower SSE scores.

Suppose that we have 10 observations in a dataset, and that we now the true y values for these observations, as well as the values for some other variables that we will use to estimate the y values. Suppose we have two different regression models and we use both of them to generate estimated y values for our 10 observations. The true and estimated y values are as follows:

• **True y values:**	`22.1, 17.9, 16.5, 14.3, 19.8, 23.7, 22.0, 18.4, 25.7, 19.2`
• **Model 1 Predictions:**	`21.4, 16.7, 17.9, 12.1, 22.1, 25.1, 21.7, 19.3, 23.4, 19.9`
• **Model 2 Predictions:**	`20.7, 18.1, 16.9, 13.6, 21.9, 24.8, 20.3, 21.1, 24.8, 18.4`

Use a code cell to define numpy arrays **`true_y`**, **`pred_1`**, and **`pred_2`** to store the values provided above. Use **`find_sse()`** to calculate the SSE score for each model, storing the results in variables named **`sse_1`** and **`sse_2`**. Print your results in the format shown below. Round your numerical outputs to 2 decimal places.

```
Model 1 SSE: xxxx
Model 2 SSE: xxxx
```

Part 3: Scoring a Classification Model

Suppose that y is a **categorical variable** that takes on values selected from a finite collection of **classes** or **labels**. A **classification model** is a model that attempts to predict the class stored in y based on the values of other input variables. Assume that we know the actual labels for a set of observations. Then we can evaluate the performance of a classification model by calculating the model's **accuracy** on the data set. The accuracy score is the proportion of observations in the data set for which the model generated a correct prediction. In other words:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of observations}}$$

Write a function named **`find_accuracy()`** that accepts two parameters, **`true_y`** and **`pred_y`**. The parameter **`true_y`** is expected to be an array of observed classes while **`pred_y`** is expected to be an array of predicted classes generated by a classification model. The function should return the accuracy score for the classification model, as calculated on this set of observations. Do not call the function from this cell. **Your function should use numpy operations and should not involve any loops**.

We will now apply our function to two different classification problems.

Suppose that a classification model is developed for the purposes of detecting the presence of a disease in patients based on the results of blood work and other medical information. To test the performance of the model, it is applied to 20 individuals for which the correct diagnosis is already known. We will use **`'P'`** to indicate a positive diagnosis (presence of the disease) and **`'N'`** to denote a negative diagnosis (absence of the disease).

The correct diagnoses for these individuals are given as follows:

```
   'P', 'P', 'N', 'N', 'P', 'N', 'N', 'N', 'P', 'N', 'N', 'N', 'N', 'P', 'P', 'N', 'N',
'N', 'N', 'N'
```

The predicted diagnoses for these individuals are:

```
   'N', 'P', 'N', 'P', 'P', 'N', 'P', 'N', 'P', 'N', 'N', 'N', 'P', 'P', 'P', 'N', 'N',
'N', 'P', 'N'
```

Create arrays named **`true_diag`** and **`pred_diag`** to store the diagnosis information provided above. Use **`find_accuracy()`** to calculate the accuracy of the classification model that generated these these predictions. print the result in the following format:

```
Model Accuracy: xxxx
```

Suppose that an image classification model is created to label images of dogs and cats. The model is applied to a collection of 24 images.

The true labels for the images are as follows:

```
'dog', 'dog', 'cat', 'dog', 'cat', 'cat', 'cat', 'dog', 'cat', 'cat', 'dog', 'cat',
'cat', 'dog', 'dog', 'dog', 'dog', 'cat', 'cat', 'cat', 'dog', 'dog', 'cat', 'cat'
```

The labels predicted by the image classification model are:

```
'dog', 'dog', 'cat', 'dog', 'cat', 'dog', 'cat', 'dog', 'cat', 'cat', 'dog', 'cat',
'cat', 'dog', 'cat', 'dog', 'dog', 'cat', 'dog', 'cat', 'dog', 'dog', 'cat', 'cat'
```

Create arrays named **`true_labels`** and **`pred_labels`** to store the label information provided above. Use **`find_accuracy()`** to calculate the accuracy of the classification model that generated these these predictions. print the result in the following format:

```
Model Accuracy: xxxx
```

Part 4: Classification Report

There are metrics other than accuracy that can be used to evaluate a classification model. **Precision** and **recall** are two such metrics that can be used to convey how well the model performs on observations in specific classes. Before we can officially define these metrics, we need to introduce a few preliminary definitions. In a **binary classification** problem, there are two possible classes. We will refer to one of the classes as the **positive class** and will refer to the other as the **negative class**. This designation is often arbitrary. Assume that we have used a classification model to generate class predictions for a data set. We can group the observations using the following designations:

An observations is considered to be:

- A **true positive** if it was predicted to be in the positive class, and actually was in the positive class.
- A **false positive** if it was predicted to be in the positive class, but actually was in the negative class.
- A **true negative** if it was predicted to be in the negative class, and actually was in the negative class.
- A **false negative** if it was predicted to be in the negative class, but actually was in the positive class.

For a set of predictions, let **TP**, **FP**, **TN**, and **FN** denote the number of true positives, false positives, true negatives, and false negatives, respectively.

The model's **positive precision**, **positive recall**, **negative precision**, and **negative recall** scores are defined as follows:

- **Positive Precision:** $\frac{\text{Number of True Positives}}{\text{Number of Positive Predictions}} = \frac{TP}{TP + FP}$
- **Positive Recall:** $\frac{\text{Number of True Positives}}{\text{Number of Positive Observations}} = \frac{TP}{TP + FN}$
- **Negative Precision:** $\frac{\text{Number of True Negatives}}{\text{Number of Negative Predictions}} = \frac{TN}{TN + FN}$
- **Negative Recall:** $\frac{\text{Number of True Negatives}}{\text{Number of Negative Observations}} = \frac{TN}{TN + FP}$

The precision for a particular class is an estimate of the probability of a correct classification, given that the model has classified an observation as that class. The recall for a particular class is an estimate of the probability of a correct classification, given that the observation is actually a member of that class.

Write a function called `classification_report()` that accepts two parameters: `true_y` and `pred_y`. This function will print several metrics used to evaluation the performance of a classification model based on the supplied values of `true_y` and `pred_y`. The function should perform the following steps:

1. Create a local variable called `classes` that stores the unique values that appear in `true_y`. You may use `np.unique()` for this. Going forward, treat the value in `classes[0]` as the "negative class" and the value in `classes[1]` as the "positive class".
2. Use `find_accuracy()` to calculate and store the model's accuracy.
3. Use NumPy (and no loops) to calculate **TP**, **FP**, **TN**, and **FN**.
4. Calculate the positive precision, positive recall, negative precision, and negative recall.
5. Print several lines displaying the results of these calculations, as shown below. The first two lines of this output should display the names of the positive and negative classes.
6. Format your results as shown below. Include a blank line between "Negative Class" and "Accuracy". Make sure that the values use to replace the xxxx symbols are left-aligned. All numeric output should be rounded to four decimal places.

```
Positive Class:     xxxx
Negative Class:     xxxx
```

```
    Accuracy:            xxxx
    Positive Precision:  xxxx
    Positive Recall:     xxxx
    Negative Precision:  xxxx
    Negative Recall:     xxxx
```

This function should not return any value.

We will now call apply this function to the examples in Part 3.

Use the **`classification_report()`** function to display a report for the medical diagnosis model from Part 3.

Use the **`classification_report()`** function to display a report for the image classification model from Part 3.

Part 5: Transformation of Random Variables

A **random variable** is a value associated with a person, object, or event that is assumed to be driven by a random process, and that will vary from one observation to the next. Every random variable has a **probability distribution** which can be used to calculate probabilities associated with that random variable. We do not always know (or even have a good guess about) the distribution of a random variable that we wish to work with.

Suppose that X is a random variable whose distribution we know, but we are interested in working with a different random variable Y that is defined as a function of X. That is, $Y = f(X)$. In this case, we can sometimes use calculus and ideas from probability theory to determine the distribution of Y, but that requires a lot of background knowledge and can be difficult to do.

As an alternative, we can estimate the probability that a transformed variable Y falls within a certain range by programmatically sampling several observations of X, transforming these into values of Y, and then determining the proportion of values that fall within the range in which we are interested. We can also use this technique to estimate quantities like the mean and standard deviation of Y. As long as we take a large enough sample from X, our estimates should be fairly accurate.

Use numpy to set a seed of 1. Use **`np.random.normal`** to sample 25,000 observations from a normal distribution with a mean of 0 and a standard deviation of 0.4. Name the resulting array **X** (not that this is a capital **X** to avoid conflict with the variable **x** created in Part 1). Assume that $Y = e^X$. Use a numpy function to create an array **Y** based on the sampled values in **X**.

Use **`np.mean()`** and **`np.std()`** (with **ddof=1**) to calculate the sample mean and sample standard deviation of both **X** and **Y**. Print the results in the format shown below, with the numerical outputs left-aligned and rounded to four decimal places.

```
    Sample Mean of X:     xxxx
    Sample Std Dev of X:  xxxx
    Sample Mean of Y:     xxxx
    Sample Std Dev of Y:  xxxx
```

We will now create histograms to get a visual sense of the distributions of **X** and **Y**.

Create a single figure with two side-by-side subplots, each of which contains a histogram. The left subplot should contain a histogram of values in **X** and the right subplot should contain a histogram of values in **Y**. Your figure and subplots should be created according to the following specifications:

- The figure size should be set to `[12, 4]`.
- Both histograms should have 30 bins and an **`edgecolor`** set to black.
- Select different named colors for the bars in the two histograms.
- The titles of the histograms should be set to "Histogram of X Values" and "Histogram of Y Values", as appropriate.

Make sure to call **`plt.show()`** to display your figure.

Finally, we will use our sampled values and numpy comparisons to estimate probabilities relating to the random variable Y.

Use **`np.mean()`** and array comparisons to calculate the proportion of values in **Y** that are less than 0.5, less than 1, and less than 2. Print your results in the format shown below, with the numerical outputs left-aligned and rounded to 4 decimal places.

```
Probability that Y is less than 0.5: xxxx
Probability that Y is less than 1.0: xxxx
Probability that Y is less than 2.0: xxxx
```

Part 6: Stochastic Linear Relationships

Two random variables X and Y are said to have a **stochastic linear relationship** if they are related by an equation of the form: $Y = a + b \cdot X + \varepsilon$, where ε is a random variable that is independent from (or in other words, unrelated to) both X and Y. The random variable ε is sometimes referred to as the **noise term** or **error term**, and is often assumed to have a mean of 0.
The intuition behind this idea is that X and Y have a relationship that is *approximately* linear, but with a little bit of noise. If you plug in a value for X, you can get an estimate for Y, but you won't know the exact value of Y because of the noise term.

In this exercise, we will create arrays representing samples collected from two variables X and Y that satisfy the following stochastic linear relationship: $Y = 5.1 + 0.9 \cdot X + \varepsilon$

Use numpy to set a random seed of 1. Create an array **`x_vals`** by sampling 200 values from a normal distribution with a mean of 10 and a standard deviation of 2. Create an array named **`errors`** by sampling 200 values from a normal distribution with a mean of 0 and a standard deviation of 1.2. Create an array called **`y_vals`** according to the equation $Y = 5.1 + 0.9 \cdot X + \varepsilon$.

Create a scatter plot showing the relationship between the values in **`x_vals`** and in **`y_vals`**. Your figure should be created according to the following specifications:

- The figure size should be set to `[8, 6]`.
- Set the point size to 60 and the alpha level to 0.8.
- Set the point border to black and select a named colors for the fill color.
- Set the labels for the axes to "X Values" and "Y Values".

Make sure to call **`plt.show()`** to display your figure.

The **correlation** between two random variables, denoted by r, is a number between -1 and 1 that measures how strong of a linear relationship there is between the two variables. If r is near 1, then the variables have a strong positive linear relationship. If r is near -1, then the variables have a strong negative linear relationship. If r is near 0, then the variables likely have no linear relationship.

Suppose we have several paired observations of two variables X and Y of the form (x_1, y_1), (x_2, y_2), …, (x_n, y_n). The sample correlation between two variables is given by the following formula:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\left[\sum_{i=1}^{n}(x_i - \bar{x})^2\right] \cdot \left[\sum_{i=1}^{n}(y_i - \bar{y})^2\right]}}$$

We will now calculate the correlation between X and Y. This can be a complicated calculation, but the steps below will outline one possible approach to performing this calculation.

1. Create an array named **`diff_x`** by subtracting the mean of **`x_vals`** from each entry of **`x_vals`**.
2. Create an array named **`diff_y`** by subtracting the mean of **`y_vals`** from each entry of **`y_vals`**.
3. Use **`diff_x`**, **`diff_y`**, and **`np.sum()`** to calculate the top of the fraction in the definition of r.
4. Use **`diff_x`**, **`diff_y`**, and **`np.sum()`** to calculate the bottom of the fraction in the definition of r.
5. Calculate r.

Print the result rounding to four decimal places with an output message that reads:

```
Correlation between X and Y: xxxx
```

Part 7: Relationship between Life Expectancy and Per Capita GDP

In Parts 7 and 8, we will return to the gapminder dataset. Make sure that the file **`gapminder_data.txt`** is in the same directory as your notebook file.

Create a cell to run the following code, exactly as written:

```
import pandas as pd
df = pd.read_csv('gapminder_data.txt', sep='\t')
country = df.country.values
year = df.year.values
continent = df.continent.values
population = df.population.values
life_exp = df.life_exp.values
pcgdp = df.gdp_per_cap.values
gini = df.gini.values
df = None
```

This code imports the gapminder dataset, and will then create arrays named **`year`**, **`country`**, **`continent`**, **`population`**, **`life_exp`**, **`pcgdp`**, and **`gini`**. The contents of these arrays will be the same as the lists from Project 01 with the same names.

The code above uses a package named **`pandas`**. We have not covered that package yet, but we will very soon.

Create a list named **`continent_list`** with the following contents: **`'africa'`**, **`'americas'`**, **`'asia'`**, and **`'europe'`**. Create a second list named **color_list** containing four strings representing four named colors. Selected colors that do not look very similar, are not too dark, and are not too light.

In the lecture covering matplotlib, we explored the relationship between per capita gdp and life expectancy. The relationship was very nonlinear. It is often easier to work with linear relationships that with nonlinear ones. We can sometimes create a linear relation from a nonlinear one by transforming one of the variables. We will now consider the relationship between life expectancy and the **natural log** of per capita gdp.

Create a scatter plot displaying the relationship between the the natural log of per capita gdp and life expectancy for 2018 data, with the color of points in the plot determined by the continent information. Instructions for creating this plot as provided below.

1. Set a figure size of `[8,6]`.
2. Use a loop to add the scatter plots for each continent one at a time. Each time the loop executes, perform the following steps:
   a. Use **Boolean masking** to create a Boolean selection array named **`sel`**. The array should indicate which elements correspond to the current continent and the year 2018.
   b. Add a scatter plot with **`x`** set to the natural log of per capita gdp of the selected countries and with **`y`** set to the life expectancy of the selected countries.
      - Set the point size to 100 and the alpha level to 0.7.
      - Set the point border to black and set the fill color to be one of the colors from **`color_list`**.
      - Set the label to be the name of the current continent. Use the **`title()`** string method to change the first character to uppercase.
3. Set the labels for the x and y axes to **`'Natural Log of Per Capita GDP'`** and **`'Life Expectancy'`**.
4. Set the title of the figure to **`'Life Expectency vs Per Capita GDP (2018)'`**.
5. Add a legend to the plot.
6. Use **`plt.show()`** to display the plot.

We will now generate a figure that separates the points for each of the continents into its own subplot.

Use **`plt.subplot()`** to split the previous plot into a 2x2 grid of subplots. The plot should be similar to the previous one, with the following changes:

1. Set a figure size of `[10,8]`.
2. You should call **`plt.subplot()`** within the loop to create a new subplot rather than adding points to an existing plot.
3. Set the limits for the x-axis of each subplot to be `[6,12]`. Set the limits for the y-axis to be `[45,90]`.
4. Each subplot should have the same labels for the x and y axes as in the previous plot.
5. Remove the label from each scatter plot, but set the title of each subplot to be the name of the continent represented by that plot, with the first letter capitalized.
6. Remove the legend from the plot.
7. Call **`plt.tight_layout()`** before **`plt.show()`** to prevent subplot elements from overlapping.
8. Use **`plt.show()`** to display the plot.

Part 8: Trends by Country

In this exercise, we will create line plots representing the change in population and life expectancy for selected countries over time.

Select 5 countries represented in the gapminder dataset. For each country, select the entries of **`population`** that correspond to this country. Keep in mind that the elements of **`population`** are already sorted in increasing order by year. For each country, add a line plot showing how the populations for those countries have changed over time. All 5 plots should appear on the same figure. The x coordinates for the points in each of your line plots should be set to the range of years from 1800 to 2018. Further specifications for your figure are as follows:

1. The figure size should be set to `[8,4]`.
2. Each line should have a label indicating the name of the country that line represents. These labels should be display in a legend.
3. For this example, let matplotlib assign the line colors automatically.
4. The x-axis should be labeled "Years" and the y-axis should be labeled "Population".
5. The title of the figure should be "Population by Year".

Use **`plt.show()`** to display the figure.

Repeat the steps from the previous code cell, but replacing **`population`** with **`life_exp`**. You may use the same 5 countries, or select different countries. Update the axis labels and titles appropriately. Otherwise keep everything the same.

Submission Instructions

When you are done, click **Kernel > Restart and Run All**. Save your notebook, and then export the notebook as an HTML file. Upload the HTML file to Canvas and upload the IPYNB file to CoCalc.