Libraries

In this pset we will be using:

- `numpy` library to store and manipulate data
- `matplotlib.pyplot` to plot data

1) Problem 1: Linear Regression

The Dataset Class

In `ps5.py`, we have provided you with the `Dataset` class. You will be using this class to access the data in `data.csv` for use throughout this pset. Read through the docstrings, and **make sure you understand what each function does and how to use them** (you do not need to understand the implementation details).

Open up `data.csv` and look at how the raw data is formatted. Each row specifies

- the city,
- the average daily temperature in Celsius, and
- the date for the years 1961-2016.

Minimizing the Squared Error

In this part, we will write code to fit a **simple linear regression** to our data set.

For each data point,

- the **x-coordinate** (the independent variable) is an `int` representing the year of the sample (e.g. 1997), and
- the **y-coordinate** (the dependent variable) is a `float` representing the temperature observed that year.

Our goal is to find the line of the form $y = mx + b$ that best fits the dataset. In other words, we want to find the line that best predicts the temperature values as a function of the year.

The best-fit line **minimizes the total squared error across all data points**.

The **squared error (SE)** for a given point (x, y) is a function of the **actual value** y and the **regression estimate** y'.

$SE(y, y') = (y – y')^2$

Our goal is to minimize $\sum_{i=1}^{n} SE(y_i, y_i')$ (total squared error) for the n points in our data set.

We can do this by taking the derivative and finding the critical points, which gives us the following equations:

$$m = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$b = \bar{y} - (m * \bar{x})$$

where $\bar{x}$ and $\bar{y}$ are average of all the x and y values, respectively.

Implement the **linear_regression** function according to its docstring. Please use the equations above to implement this function. Do NOT use **numpy.polyfit** for this problem. Run the **ps5_tester.py** file. Your code should pass **test_linear_regression**.

Evaluating the Fit

We want to assess how well our best-fit line actually fits our data set by calculating the total squared error across all the data points.

Implement the **squared_error** function according to its docstring. Your code should pass **test_squared_error**.

2) Problem 2: Curve Fitting

In this section, you will write code to fit several **polynomial regression models** to our data set.

Implement the **generate_polynomial_models** function according to its docstring. This function should return a list of best-fit polynomial models for a given data set. A model is defined here as a one-dimensional numpy array containing the coefficients of the polynomial. Therefore, the value returned by this function is a **list of numpy arrays**.

Note: The models should appear in the list in the same order as their corresponding integers in the **degrees** parameter. Your code should pass **test_generate_polynomial_models**.

Example:

```
>>> print(generate_polynomial_models(np.array([1961, 1962, 1963]),
                          np.array([-4.4, -5.5, -6.6]),  [1, 2]))
[array([ -1.10000000e+00,   2.15270000e+03]),
 array([  8.86320195e-14,  -1.10000000e+00,   2.15270000e+03])]
```

- Note that due to numerical errors, it is fine if you do not get this exact output, but it should be very close.

Some helpful hints:

- Check out the documentation for `numpy.polyfit`.
- **Note that the inputs x and y are one-dimensional numpy arrays, NOT Python lists**. Although they are similar, do not expect them to always behave like Python lists. Some functions you will be calling only work with numpy arrays, so be careful!
- The documentation for numpy N-dimensional arrays can be found here.

3) Problem 3: Evaluating the Models

In this problem, we will write a function that uses numerical and graphical metrics to evaluate the regression models generated by `generate_polynomial_models`.

More specifically, we will use the following:

- The **model's R^2 value** (i.e. its coefficient of determination). You are welcome to use the r2_score function that we have imported for you in `ps5.py`.
- A **plot** of the data samples and the polynomial curves.
- The **standard error over slope** (only for linear regression models).

Standard Error Over Slope
The ratio of the standard error of the fitted curve's slope to the slope measures how likely it is that you would see the upward or downward trend in your data and fitting curve purely by chance. The larger the absolute value of this ratio is, the more likely it is that the trend is due to chance. If you are interested in learning more, check out: Hypothesis Test for Regression Slope.

In our case, if the absolute value of the ratio is less than 0.5, the trend is significant (i.e. not by chance). We have written a helper function `standard_error_over_slope` that calculates this value for you.

Implement the function `evaluate_models` according to its docstrings.

Hint: You might find numpy.poly1d or numpy.polyval helpful for calculating the predicted y values based on the model. Refer to Lecture 9 for detailed explanations on plotting using matplotlib.pyplot.

You should make a separate plot for each model. Your figure should adhere to the following guidelines:

- **Plot the data points** as individual blue dots.
- **Plot the model** with a red solid line color.
- **Include a title**. Your title should include the R^2 value of the model and the degree. If the model is a linear curve (i.e. its degree is one), the title should also include the ratio of the standard error of this fitted curve's slope to the slope.
- **Label the axes**. You may assume this function will only be used in the case where the x-axis represents years and the y-axis represents temperature in degrees Celsius.

Please round your R^2 and SE/slope values to 4 decimal places.
The R^2 values will be tested by `ps5_tester.py` but the graphs will not. However, you will use the graphs from this function for discussion questions 4A, 4B, 5B, 5C, and 6B.

4) Problem 4: Sampling the Data

Now that we have all the components we need, we can select data samples from the raw temperature records and investigate trends for those samples. In this problem, we will try out two different methods of sampling data.

Part 4A: Daily Temperature

For our first sampling method, we will pick an arbitrary day of the year and see whether we can find any trends in the temperature on this day across multiple years.

Write your code for Problem 4A under `if __name__ == '__main__'`.
Use the `Dataset` class to create a data set where

- the **x-coordinates** are the years from 1961 to 2016 and
- the **y-coordinates** are the temperature measurements in Boston on December 1st for each year

Remember to represent the x and y coordinates as numpy arrays when passing them as arguments to functions expecting numpy arrays. Next, fit the data to a degree-one polynomial with `generate_polynomial_models` and plot the regression results using `evaluate_models`.

Part 4B: Annual Temperature

Let's try another way to sample data points. Instead of looking at the change in temperature for a single day, we will look at the change in the average annual temperature. Repeat the steps in 4A, but now let the **y-coordinates** be the **average annual temperature in Boston**. We will use the same year range as 4A.

1. **Implement the function** `calculate_annual_temp_averages` in the Dataset class **according to its docstrings**.

   Example:
   ```
   >>> print(dataset.calculate_annual_temp_averages(['TAMPA', 'DALLAS'], range(2008, 2015)))
   [21.76502732 21.24561644 20.8040411 22.03910959 22.27206284 21.31136986 20.88123288]
   ```

   The first element in the returned numpy array corresponds to the average of the annual temperatures for Tampa and Dallas in the year 2008.

   Your code should pass `test_calculate_annual_temp_averages`.

2. Use `calculate_annual_temp_averages` **to generate the y-coordinates for your new data set under** `if __name__ == '__main__'`.

   Fit your dataset to a degree-one polynomial with `generate_polynomial_models` and plot the regression results using `evaluate_models`.

5) Problem 5: Long-Term vs. Short-Term Trends

Looking at trends within smaller time intervals may lead us to make different conclusions from the data. In this problem, you will make use of linear regression models to construct conflicting narratives for the change in the average annual temperature in Seattle.

Part 5A: Finding Extreme Trends

Implement the function `get_max_trend` **according to its docstrings.**

This function takes as arguments the x and y numpy arrays of samples, an interval length, and a specified trend. It returns a tuple that represents the start and end **indices** of the x and y numpy arrays whose linear regression model produces the most positive or most negative slope (as specified by the `positive_slope` parameter).

Note: Due to floating point precision errors, use a tolerance of 1e-8 to compare slope values (i.e. a float x and a float y are considered equal if $abs(x-y) <= 1e-8$). If there are any ties, we resolve ties by returning the interval which occured first.

Your code should pass `test_get_max_trend`.

Part 5B: Increasing

Suppose you are the mayor of Seattle, and you are taking a data-driven approach to policy. You want to call your citizens to action against climate change by finding data supporting the claim that temperatures are increasing in Seattle.

Write your code for Problem 5B under `if __name__ == '__main__'`.

Use `get_max_trend` to identify a window of **30 years** that demonstrates that the average annual temperature in Seattle is **rising**. Plot the corresponding model with `evaluate_models`.

Part 5C: Decreasing

The political group "Turn Down the AC" has promised to donate 1 trillion dollars to your campaign, as long as you amend your previous statements about temperature change to a more agnostic opinion.

Write your code for Problem 5C under `if __name__ == '__main__'`.

Use `get_max_trend` to identify a window of **15 years** that demonstrates that the average annual temperature in Seattle is **decreasing**. Plot the corresponding model with `evaluate_models`.

Part 5D: Finding Extreme Trends for all Possible Intervals

Implement the function `get_all_max_trends` **according to its docstrings.**

This function takes as arguments the x and y numpy arrays of samples. It returns a a list of tuples that represents the most extreme trend (positive OR negative) of every possible interval length (2 <= interval length <= len(x)).

Note: Due to floating point precision errors, use a tolerance of 1e-8 to compare slope values (i.e. a float x and a float y are considered equal if $abs(x - y) <= 1e - 8$). If there are any ties, we resolve ties by returning the interval which occured first.

Your code should pass `test_get_max_trend`.

6) Problem 6: Predicting the Future

Now, we are curious whether we can predict future temperatures based on models created from historical data.

We will divide our original 1961-2016 dataset into two sets of data: 1961-1999 will serve as our **training data**, and 2000-2016 will serve as our **testing data**. We will use the training data to generate our models, and then we will evaluate the accuracy of our models at predicting the temperatures during the testing interval.

Use the provided variables `TRAIN_INTERVAL` and `TEST_INTERVAL` to represent these ranges of years in your code.

Part 6A: RMSE

Before we use our models to predict "future" data points (i.e. temperatures for years later than 1999), we should think about how to evaluate a model's performance on this task. Recall that R^2 measures how closely a model matches the training data, which does not give us any indication of how well the model fits the testing data. One way to evaluate a model's performance on test data is with the **Root Mean Square Error (RMSE)**. This measures the deviation between a model's predicted values and the actual values across all the data samples.

RMSE can be found as follows:

$$RMSE = \sqrt{\frac{\sum^{n}(y_{obs,i} - y_{pred,i})^2}{n}}$$

- $y_{pred,i}$ is the predicted y-value for the ith data point based on the regression model
- $y_{obs,i}$ is the actual y-value for the ith data point based on the raw data
- n is the number of data points

For this part you should:

1. Implement the function `calculate_rmse` according to its docstrings.

   Your code should pass `test_calculate_rmse`.

2. Implement the function `evaluate_rmse` according to its docstrings.

   This function is very similar to `evaluate_models`, except that you should report the RMSE in the title rather than the R2 value. You also do not need to include SE/slope.

Part 6B: Predicting

Now that we have a method for evaluating model performance on test data, we are going to use our models to make predictions.

(i) Generate models based on training data

First, we want to generate regression models based on the training data.

Write your code for Problem 6B under `if __name__ == '__main__'`.

1. Use the `Dataset` class to generate a **training set** of the **national annual average temperature** for the years in `TRAIN_INTERVAL`.
2. Fit the training set to polynomials of degree 2 and 10 with `generate_polynomial_models` and plot the results with `evaluate_models`.

(ii) Predict the results

Now, let's make some predictions, and compare our predictions to the real average temperatures from 2000-2016.

Continue writing your code for Problem 6B under `if __name__ == '__main__'`.

1. Use the `Dataset` class to generate a **test set** of the **national annual average temperature** for the years in `TEST_INTERVAL`.
2. Evaluate the predictions of each model obtained in the previous section and plot the results with `evaluate_calculate_rmse`. Read the docstring carefully to make sure you use the function correctly.

7) Problem 7: Clustering Climates

This final portion of the pset involves k-means clustering. Part 7 has no tests associated with it. Your code and outputs for Part 7 will be entirely evaluated in the pdf submission.

For Part 7, you will be using k-means clustering to group the climates of each city in our dataset. Within the `cluster_cities` function:

1. convert the daily temperature data for each city in **cities** into a numpy array feature vector to be used for clustering. Each city feature vector should be of length 365, with each element in the array being the average temperature for that day across all years in **years**. Note that in the case of leap years, with 366 days, we will only use the first 365 days. Your final array should have shape (`len(cities)`, 365)
2. cluster the feature vectors using the KMeans class from the sklearn package, with **n_clusters** clusters. This class has already been imported for your convenience. Feel free to check out the docs for KMeans at https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.
3. take note of the cities in each cluster. You will be asked about what types of cities you find in each cluster

Lastly, you will plot each city feature vector. Your line plot should have the following properties:

- x-axis: days of the year, 1-365
- y-axis: degrees celcius
- label x and y axis appropriately
- each city feature vector should be colored according to the cluster it was associated with
- title the plot appropriately
- include a legend for the plot, mapping line colors to cluster numbers

Cluster and plot the average daily temperature for all cities in **CITIES**, for all years [1961,2016], using **n_clusters=4**. **IMPORTANT: Save this plot to be submitted in a pdf in Part 8B.**

Hand-In Procedure

1. Save

Save your solution as **ps5.py**. Maks sure to save your plot from Part 7 in a pdf.

2. Time and Collaboration Info

At the start of each file, in a comment, write down the names of your collaborators. For example:

```
# Problem Set 5
# Name: Jane Lee
# Collaborators: John Doe
```

Please estimate the number of hours you spent on the Problem Set in the question box below.

12

Submit Thank you!

You have infinitely many submissions remaining.

3. Sanity checks

After you are done with the problem set, run your files and make sure they run without errors.

Note: Passing all of the tests in the local tester does not necessarily mean you will receive a perfect score. The staff will run additional tests on your code to check for correctness.