AI 534: Machine Learning HW1: Feature Map and k-NN (20%)

Instructions:

1. This HW, like all other programming HWs, should be done in Python 3 and numpy only. See the course homepage for a numpy tutorial. If you don't have a Python+numpy installation yourself, you can use the College of Engineering servers by

   `ssh username@access.engr.oregonstate.edu`

   replacing `username` with your ENGR user name (should be identical to your ONID login). See the following for instructions on using SSH from Windows:

   `https://it.engineering.oregonstate.edu/accessing-unix-server-using-putty-ssh`

   If you don't have an ENGR account, see `https://it.engineering.oregonstate.edu/get-engr-account`.

   You're **highly recommended** to set up SSH keys to bypass Duo authentication and password:

   `https://it.engineering.oregonstate.edu/ssh-keygen`

   The default `python3` on ENGR servers have `numpy`, `matplotlib`, `sklearn`, and `pandas` installed.

2. Besides machine learning, this HW also teaches you data (pre-)processing skills and Unix (Linux or Mac OS X) command lines tools. These skills are even more important than machine learning itself for a software engineer or data scientist. As stated in the syllabus, Windows is **not** recommended. The instructor and the TA do not have access to Windows computers and will not be able to provide technical assistance for Windows. Use **cmder, cygwin**, or ssh (see above) if you don't have Linux or Mac OS X on your own computer.

   If you prefer to use Jupyter notebook but have difficulty setting it up on your own computer, the TAs have written a tutorial on running jupyter notebook remotely on the server:

   `https://classes.engr.oregonstate.edu/eecs/spring2022/cs513-400/extra/TAs_jupyter_tutorial.pdf`

3. Ask questions on **Ed Discussion**. You're encouraged to answer other students' questions as well.

4. Download HW1 data from the Canvas. Unzip it by running `tar -xzvf hw1-data.tgz` (here `-x` means "extract" and `z` means "unzip"; you might also be able to open it using WinZip or 7-zip on Windows). It contains:

`income.train.txt.5k`	training data (5,000 examples, with labels)
`income.dev.txt`	dev set (1,000 examples, with labels)
`income.test.blind`	test set (1,000 examples, without labels)
`toy.txt` & `binarize_example.py`	toy data and binarization code example
`validate.py` & `random_output.py`	tools to validate your test result

   The semi-blind test set does not contain the target labels (`>50K` or `<=50K`), which you will need to predict using your best model. Part of your grade is based on your prediction accuracy on test. Please make heavy use of the three python programs supplied, so that you don't need to start from scratch. For example, you should base your Part 1 on `binarize_example.py` and use `validate.py` to verify your test result in Part 4.

5. You should submit a single `.zip` file containing `hw1-report.pdf`, `income.test.predicted`, and all your code. LaTeX'ing is recommended but not required. **Do not forget the debrief section (in each HW).**

1 Hands-on Exploration of the Income Dataset (3 pts)

1. Take a look at the data. A training example looks like this:

   `37, Private, Bachelors, Separated, Other-service, White, Male, 70, England, <=50K`

   which includes the following 9 input fields plus one output field (y):

   age, sector, education, marital-status, occupation, race, sex, hours-per-week, country-of-origin, target

   Q: What are the positive % of training data? What about the dev set? Does it make sense given your knowledge of the average per capita income in the US? (0.5 pts)

2. Q: What are the youngest and oldest ages in the training set? What are the least and most amounts of hours per week do people in this set work? (0.5 pts) Hint:

   ```
   $ cat income.train.txt.5k | sort -nk1 | head -1
   ```

   Note the `$` is the prompt, not part of the command. `cat` lists all contents of a file, `sort -nk1` means sort by the first column numerically, and `head -1` lists the first line. Here the vertical bar `|` means "pipe", i.e., feeding the output of the previous command as input to the next command.

3. There are two types of fields, *numerical* (*age* and *hours-per-week*), and *categorical* (everything else).[1] The default preprocessing method is to *binarize* all categorical fields, e.g., *race* becomes many *binary* features such as `race=White`, `race=Asian-Pac-Islander`, etc. These resulting features are all binary, meaning their values can only be 0 or 1, and for each example, in each field, there is one and only one positive feature (this is the so-called "one-hot" representation, widely used in ML and NLP).

   Q: Why do we need to binarize all categorical fields? (0.5 pts)

   Q: Why we do **not** want to binarize the two numerical fields, *age* and *hours*? (0.5 pts)

4. Q: How should we deal with the two numerical fields? Just as is, or normalize them (say, age / 100)? (0.5 pts)

   Hint: Note that the max distance between two people on a categorial field is 2. If we simply "normalize" a numerical field by, say, age / 100, what's the max distance on age? The idea is to treat all fields equally.

5. Q: How many features do you have in total (i.e., the dimensionality)? Hint: should be around **90**. **Note:** the target label is not a feature! How many features do you allocate for each of the 9 fields? (0.5 pts) **Hint:**

   ```
   $ for i in `seq 1 9`; do cat income.train.txt.5k | cut -f $i -d ',' | sort | uniq | wc -l; done
   ```

   Here `` `seq 1 9` `` returns `1 2 ... 9`, `cut` extracts specific columns of each line (e.g., `cut -f 2` extracts the second column), and `-d ','` means using comma as the column separator. `uniq` filters out duplicate consecutive rows, and `wc -l` counts the number of lines. So `sort | uniq | wc -l` returns the number of unique rows.

2 Data Preprocessing and Feature Extraction I: Naive Binarization (3 pts)

In this section, we'll delve into a simple method of data processing, naive binarization, and examine its implications and utility when applied to the Income dataset. Given the structure of our dataset, we have both numerical and categorical data. For the purpose of this exploration, we'll treat the numerical data (*age* and *hours-per-week*) equivalently to the categorical data. This means that an *age* of `37` will be treated similarly to a label such as `Private` in the *sector* field.

1. **Pandas and Data Loading.** Before we proceed with feature extraction, let's understand how to load our dataset using the `pandas` library. The `read_csv` function facilitates this, and here we showcase loading from the toy dataset `toy.txt`:

   ```
   import pandas as pd

   # Load the toy dataset
   data = pd.read_csv("toy.txt", sep=", ", names=["age", "sector"])
   ```

   Here's a breakdown of the parameters:

   (a) `sep`: The separator between each data point. In this case, it's a comma followed by a space. We specify this to ensure the data is read correctly.

   (b) `names`: This specifies the column headers or names.

[1] In principle, we could also convert *education* to a numerical feature, but we choose **not** to do it to keep it simple.

If you print `data`, it should output:

```
   age       sector
0   45  Federal-gov
1   33      Private
2   19      Private
3   47      Private
4   30    Local-gov
5   37      Private
6   35      Private
7   33      Private
8   40      Private
9   54  Federal-gov
```

Masking in `pandas` is useful for filtering data based on specific conditions. For instance, to view rows where *age* is `33` and *sector* is `Private`, use:

```
filtered_data = data[(data['age'] == 33) & (data['sector'] == 'Private')]
```

This returns a DataFrame containing only the rows that satisfy both conditions.

Hint: for simplicity reasons, you can use the following column names in your implementation:

`["age", "sector", "edu", "marriage", "occupation", "race", "sex", "hours", "country", "target"]`

2. **Binarization in Pandas.** Let's now introduce the concept of one-hot encoding with a practical example from our toy dataset. One-hot encoding is a technique to convert categorical data into a format that could be fed into machine learning algorithms. Essentially, for each unique value in a category, a new binary column is created. If the original value of that row matches the unique value associated with a column, the new column gets a `1`, otherwise, it gets a `0`. Using `pandas`, the one-hot encoding can be done as:

```
encoded_data = pd.get_dummies(data, columns=["age", "sector"])
print(encoded_data)
```

This produces:

```
   age_19  age_30  age_33  age_35  age_37  age_40  age_45  age_47  age_54  sector_Federal-gov  sector_Local-gov  sector_Private
0       0       0       0       0       0       0       1       0       0                   1                 0               0
1       0       0       1       0       0       0       0       0       0                   0                 0               1
2       1       0       0       0       0       0       0       0       0                   0                 0               1
3       0       0       0       0       0       0       0       1       0                   0                 0               1
4       0       1       0       0       0       0       0       0       0                   0                 1               0
5       0       0       0       0       1       0       0       0       0                   0                 0               1
6       0       0       0       1       0       0       0       0       0                   0                 0               1
7       0       0       1       0       0       0       0       0       0                   0                 0               1
8       0       0       0       0       0       1       0       0       0                   0                 0               1
9       0       0       0       0       0       0       0       0       1                   1                 0               0
```

This output showcases the one-hot encoded version of our data. Each unique age and sector has been turned into its own separate column. In the resulting dataframe, if an individual's *age* was `45`, the *age_45* column will have a `1`, while all other age columns will have a 0. The same logic is applied to the sector columns.

Notice how columns are named: the naming convention starts with the original column's name (e.g., "age" or "sector"), followed by an underscore and then the unique value (e.g., "19" or "Federal-gov"). This naming helps in distinguishing the source of each one-hot encoded column.

Upon completing this transformation for both age and sector columns, you can concatenate them to produce a comprehensive one-hot encoded dataset. The columns of the dataset are now in a format that's suitable for feeding into most machine learning algorithms.

Question: Although `pandas` provides excellent utilities for data processing and manipulation, such as `get_dummies`, it's not a good idea to use it in machine learning in general. Why? (0.25 pts) (Hint: It's important to think about the entire pipeline. When working with separate training, developing and testing datasets, we need to ensure consistent binarization across all datasets.)

3. **Binarization in Scikit-Learn.** To practically implement this naive binarization, we will make use of the `OneHotEncoder` class from the `sklearn.preprocessing` library instead. This tool will transform our input features into a one-hot encoded format. Here's a brief step-by-step demonstration:

   - Import the necessary libraries:

     ```
     from sklearn.preprocessing import OneHotEncoder
     ```

   - Instantiate the encoder:

     ```
     encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
     ```

   - Fit the encoder to the dataset and transform:

     ```
     encoder.fit(data)
     binary_data = encoder.transform(data)
     ```

   In practice, you can combine these two lines by one function `encoder.fit_transform(data)` but we chose to separate them for clarity. The output, `binary_data`, will now have a binary representation for each of the features in the dataset:

   ```
   [[0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0.]
    [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
    [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
    [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1.]
    [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
    [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1.]
    [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1.]
    [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
    [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1.]
    [0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0.]]
   ```

   You can check the name of each column by `encoder.get_feature_names_out()`, which gives us:

   ```
   ['age_19' 'age_30' 'age_33' 'age_35' 'age_37' 'age_40' 'age_45' 'age_47' 'age_54'
    'sector_Federal-gov' 'sector_Local-gov' 'sector_Private']
   ```

   Question: After implementing the naive binarization to the real training set (NOT the toy set), what is the feature dimension? Does it match with the result from the previous section? (0.25 pts)

4. **Fit k-NN via Scikit-Learn.** With the dataset now binarized using the `OneHotEncoder`, an intriguing exploration is to employ the k-nearest neighbors (k-NN) algorithm on this transformed data. The k-NN algorithm works by classifying a data point based on how its neighbors are classified. The number of neighbors, denoted as k, is a parameter that can greatly affect the performance of the k-NN algorithm.

   To begin with, you would utilize the `KNeighborsClassifier` from `sklearn.neighbors`. Given the binary representation of the dataset, you can follow these general steps:

   - Prediction: Predict the labels for both the training set (to get training accuracy) and the test set.
   - Evaluation: Calculate and compare the accuracy scores for the predictions on the training and test sets.

   Questions:

   (a) Evaluate k-NN on both the training and dev sets and report the error rates and predicted positive rates for k from 1 to 100 (odd numbers only, for tie-breaking), e.g., something like:

```
k=1    train_err xx.x% (+:xx.x%)  dev_err xx.x% (+:xx.x%)
k=3    ...
...
k=99  ...
```

Q: what's your best error rate on dev? Which k achieves this best error rate? (Hint: 1-NN dev error should be ∼23% and the best dev error should be ∼16%). (1 pt)

(b) Q: When $k = 1$, is training error 0%? Why or why not? Look at the training data to confirm your answer. (0.5 pts)

(c) Q: What trends (train and dev error rates and positive ratios, and running speed) do you observe with increasing k? Do they relate to underfitting and overfitting? (0.5 pts)

(d) Q: What does $k = \infty$ actually do? Is it extreme overfitting or underfitting? What about $k = 1$? (0.5 pts)

3 Data Preprocessing and Feature Extraction II: Smart Binarization (3 pts)

In the previous section, we discussed the naive binarization of the Income dataset. While this method can be effective for certain machine learning models, there are more refined data processing techniques that can yield better results, especially when dealing with a mix of numerical and categorical data. This section delves into smarter binarization methods, particularly focusing on differential handling of numerical and categorical features and scaling the numerical data for better algorithmic performance.

1. For effective machine learning model training, especially with algorithms sensitive to distance metrics such as k-NN, it's pivotal to process data intelligently. Given our dataset, we treat numerical features (*age* and *hours-per-week*) and categorical features differently.

   For categorical data, we continue to use `OneHotEncoder` for binarization. However, for numerical data, instead of binarizing them directly, we let them remain in their original format. Here's how you'd typically proceed:

   - Differentiate between numerical and categorical columns in your dataset.
   - Apply `OneHotEncoder` only to the categorical columns. For our dataset, all columns except *age* and *hours-per-week* will undergo this transformation.
   - Combine the processed categorical data with the numerical data to form the modified dataset.

   Here, we provide an example for the toy dataset:

   - Import the necessary libraries:

     ```
     from sklearn.compose import ColumnTransformer
     from sklearn.preprocessing import OneHotEncoder
     ```

   - Instantiate the preprocessing methods for numerical and categorical data:

     ```
     num_processor = 'passthrough' # i.e., no transformation
     cat_processor = OneHotEncoder(sparse=False, handle_unknown='ignore')
     ```

     Note that here `handle_unknown='ignore'` is important, otherwise when applied to dev/test data, it will throw an error on new features that it has not seen on the training set (e.g., a new country).

   - Fit the preprocessor to the dataset and transform:

     ```
     preprocessor = ColumnTransformer([
         ('num', num_processor, ['age']),
         ('cat', cat_processor, ['sector'])
     ])
     preprocessor.fit(data)
     processed_data = preprocessor.transform(data)
     ```

And the `processed_data` will give us:

```
[[45.  1.  0.  0.]
 [33.  0.  0.  1.]
 [19.  0.  0.  1.]
 [47.  0.  0.  1.]
 [30.  0.  1.  0.]
 [37.  0.  0.  1.]
 [35.  0.  0.  1.]
 [33.  0.  0.  1.]
 [40.  0.  0.  1.]
 [54.  1.  0.  0.]]
```

After this preprocessing step, rerun the k-NN experiment on this data. With the numerical data kept in its original form and the categorical data binarized, there's potential for improved classification performance.

Question: Re-execute all experiments with varying values of k (Part 2, Question 4a) and report the new results. Do you notice any performance improvements compared to the initial results? If so, why? If not, why do you think that is? (1.5 pts) (Hint: 1-NN dev error should be ∼27% and the best dev error should be ∼21%)

2. Simply keeping numerical data unchanged may not always yield optimal results. Algorithms like k-NN, which rely on distance metrics, can be influenced by the scale of the features. A variable that spans a large range can unduly influence the distance computation. For instance, *age* and *hours-per-week* typically has a much larger range than other one-hot columns, which could result in *age* and *hours-per-week* dominating the distance computations.

   To tackle this, we can perform rescaling on the numerical data, ensuring they lie within a similar range (e.g., make all column distances up to 2). `MinMaxScaler` from `sklearn.preprocessing` is a tool designed for this task, scaling features to lie between a given minimum and maximum value, often between zero and one.

   Here's an improved version of the previous implementation:

   - Import the necessary libraries:

     ```
     from sklearn.compose import ColumnTransformer
     from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
     ```

   - Instantiate the processing methods for numerical and categorical data:

     ```
     num_processor = MinMaxScaler(feature_range=(0, 2))
     cat_processor = OneHotEncoder(sparse=False, handle_unknown='ignore')
     ```

   - Fit the processor to the dataset and transform:

     ```
     preprocessor = ColumnTransformer([
         ('num', num_processor, ['age']),
         ('cat', cat_processor, ['sector'])
     ])
     preprocessor.fit(data)
     processed_data = preprocessor.transform(data)
     ```

   This new version will produce more reasonable features for numerical data:

```
[[1.48571429 1.         0.         0.        ]
 [0.8        0.         0.         1.        ]
 [0.         0.         0.         1.        ]
 [1.6        0.         0.         1.        ]
 [0.62857143 0.         1.         0.        ]
 [1.02857143 0.         0.         1.        ]
 [0.91428571 0.         0.         1.        ]
 [0.8        0.         0.         1.        ]
 [1.2        0.         0.         1.        ]
 [2.         1.         0.         0.        ]]
```

You can also check the name of each column by `preprocessor.get_feature_names_out()`, which gives us:

```
['num__age' 'cat__sector_Federal-gov' 'cat__sector_Local-gov' 'cat__sector_Private']
```

Questions: Again, rerun all experiments with varying values of k and report the results (Part 2, Question 4a). Do you notice any performance improvements? If so, why? If not, why do you think that is? (1.5 pts) (Hint: 1-NN dev error should be ∼24% and the best dev error should be ∼15%)

4 Implement your own k-Nearest Neighbor Classifiers (5 pts)

While leveraging libraries like `sklearn` can be incredibly convenient for employing algorithms like k-NN, there's significant educational value in implementing the algorithm from scratch by yourself. By doing so, you can gain a deeper understanding of the intricacies and operations that occur under the hood.

Note: you can use the Matlab style "broadcasting" notations in numpy (such as matrix - vector) to calculate many distances in one shot. For example, if `A` is an $n \times m$ matrix (n rows, m columns, where n is the number of people and m is the number of features), and `p` is an m-dimensional vector (1 row, m columns) representing the query person, then `A - p` returns the difference vectors from each person in `A` to the query person `p`, from which you can compute the distances:

```
>>> A = np.array([[1,2], [2,3], [4,5]]); p = np.array([3,2])
>>> A - p
array([[-2,  0],        [-1,  1],        [ 1,  3]])
>>> np.linalg.norm(A-p, axis=1)
array([2.        , 1.41421356, 3.16227766])
```

This is Euclidean distance (what does `axis=1` mean?). You need to figure out Manhattan distance yourself.

1. **Distance Verification (sanity check).** Once you have prepared the dataset, use the `sklearn` k-NN predictor (with rescaled smart binarization) to validate your results for the first person in the `dev` dataset. Particularly, identify the three closest individuals from the `train` dataset.

   The first person in the `dev` set:

   ```
   $ head -1 income.dev.txt
   45, Federal-gov, Bachelors, Married-civ-spouse, Adm-clerical, White, Male, 45, United-States, <=50K
   ```

   Its binarized features should look like:

   ```
        age  sector_Federal-gov  sector_Local-gov  ...  country_United-States  country_Vietnam  country_Yugoslavia
   0.767123                 1.0               0.0  ...                    1.0              0.0                 0.0
   ```

   Using the `kneighbors` method from `KNeighborsClassifier`, according to the Euclidean distance, you can find the following three closest individuals from the `train` dataset:

   ```
   Three closest neighbors from the train dataset:
   Neighbor 1 at index 4872:
        age  sector_Federal-gov  sector_Local-gov  ...  country_United-States  country_Vietnam  country_Yugoslavia
   0.438356                 1.0               0.0  ...                    1.0              0.0                 0.0

   Neighbor 2 at index 4787:
        age  sector_Federal-gov  sector_Local-gov  ...  country_United-States  country_Vietnam  country_Yugoslavia
   0.821918                 1.0               0.0  ...                    0.0              0.0                 0.0

   Neighbor 3 at index 2591:
        age  sector_Federal-gov  sector_Local-gov  ...  country_United-States  country_Vietnam  country_Yugoslavia
   0.849315                 1.0               0.0  ...                    1.0              0.0                 0.0
   ```

   The original rows in the `train` dataset:

```
$ sed -n 4873p income.train.txt.5k
33, Federal-gov, Bachelors, Married-civ-spouse, Adm-clerical, White, Male, 42, United-States, >50K
$ sed -n 4788p income.train.txt.5k
47, Federal-gov, Bachelors, Married-civ-spouse, Adm-clerical, White, Male, 45, Germany, >50K
$ sed -n 2592p income.train.txt.5k
48, Federal-gov, Bachelors, Married-civ-spouse, Prof-specialty, White, Male, 44, United-States, >50K
```

Also notice that in this example, the 3-NN prediction is wrong, as the top-3 closest examples are all `>50K`.

Finally, remember that you don't really need to sort the distances in order to get the top-k closest examples.

Note: If you couldn't get the same top-3 people listed above, it is possible that you included the target label in your feature map.

Question: Before you implement your k-NN classifier, try to verify the top-k neighbors and their distances you found with the `sklearn` implementation. What the (Euclidean/Manhattan) distances of the top-3 people? Report both the results from `sklearn` and your own implementation. (0.5 pts)

2. **Implement your own k-NN classifier** (with the default Euclidean distance).

   (a) Q: Is there any work in training after finishing the feature map? (0.5 pts)

   (b) Q: What's the time complexity of k-NN to test one example (dimensionality d, size of training set $|D|$)? (1 pt)

   (c) Q: Do you really need to sort the distances first and then choose the top k? Hint: there is a faster way to choose top k without sorting. (0.5 pts)

   (d) Q: What numpy tricks did you use to speed up your program so that it can be fast enough to print the training error? Hint: (i) broadcasting (such as matrix - vector); (ii) `np.linalg.norm(..., axis=1)`; (iii) `np.argsort()` or `np.argpartition()`; (iv) slicing. The main idea is to do as much computation in the vector-matrix format as possible (i.e., the Matlab philosophy), and as little in Python as possible. (1 pt)

   (e) Q: How many seconds does it take to print the training and dev errors for $k = 99$ on ENGR servers? Hint: use `time python ...` and report the user time instead of the real time. (Mine was about 14 seconds). (0.5 pts)

3. **Redo the evaluation using Manhattan distance.** Better or worse? (1 pt)

5 Deployment (4 pts)

Now try more k's and take your best model and run it on the semi-blind test data, and produce `income.test.predicted`, which has the same format as the training and dev files.

Q: At which k and with which distance did you achieve the best dev results? (0.25 pts)

Q: What's your best dev error rates and the corresponding positive ratios? (0.25 pts)

Q: What's the positive ratio on test? (0.25 pts)

Part of your grade will depend on the accuracy of `income.test.predicted` (3.25 pts).

IMPORTANT: You should use our `validate.py` to verify your `income.test.predicted`; it will catch many common problems such as formatting issues and overly positive or overly negative results:

```
cat income.test.predicted | python3 validate.py
```

We also provided a `random_output.py` which generates random predictions (∼50% positive) and it will pass the formatting check, but fail on the positive ratio:

```
$ cat income.test.blind | python3 random_output.py | python3 validate.py
```

which might output:

```
Your file passed the formatting test! :)
Your positive rate is 49.6%.
ERROR: Your positive rate seems too high (should be similar to train and dev).
PLEASE DOUBLE CHECK YOUR BINARIZATION AND kNN CODE.
```

If you test the dev set, it will certainly pass this test (try `cat income.dev.txt | python3 validate.py`).

Our automatic grading system will assume your `incoming.test.predicted` passes this test; if it doesn't, you will receive 0 points for the blind test part.

6 Observations (2 pts)

1. Q: Summarize the major drawbacks of k-NN that you observed by doing this HW. There are a lot! (1 pt)
2. Q: Do you observe in this HW that best-performing models tend to exaggerate the existing bias in the training data? Is it due to overfitting or underfitting? Is this a potentially social issue? (1 pt)

7 Extra Credit Question (extra 2 pts)

For an additional challenge, visualize the development error rate of all three versions against the values of k. Plot these three error rate curves on a single graph to allow for easy comparison. Use different colors or styles for each curve and be sure to include a legend to distinguish between them.

Debriefing (required in your report)

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone, or mostly with other people?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?