Assignment 5

B551

Due: Friday December 8, 11:59:59 PM Eastern time

The last assignment of the course will give you a chance to implement and develop some machine learning classification algorithms from scratch and test out your implementations on various standard data-sets.

Guidelines for this Assignment

Please read the instructions below carefully as there are a few changes in the guidelines compared to previous assignments.

Coding requirements. For fairness and efficiency, we use a semi-automatic program to grade your submissions. This means you must write your code carefully so that our program can run your code and understand its output properly. As usual, we require the following.

1. Your code for the assignment must be written in Python 3, not Python 2. The skeleton code is already written in Python 3, but inserting Python 2 code that is not compatible with Python 3 will cause the program and autograding to fail.

2. Use the skeleton code that is provided for you and follow the instructions in the skeleton code and the specifications laid out in the assignment below. This means that you should not change file names for your python scripts nor change the parameters for functions that are already in the base skeleton code. Our autograding scripts may call these functions directly in order to test your implementations, and changing the file name or changing the parameters of these functions will cause the scripts to fail and you will likely lose points. Of course, feel free to create new functions as needed, but keep in mind that the functions in the base skeleton code must work as intended. Finally, please avoid using global (public) variables in your programs as they may cause your program to behave unexpectedly when run by our grading program.

3. You may import Python modules from the Python Standard Library for routines not related to A.I. This includes items such as basic sorting algorithms and data structures like queues, as long as they are already installed on the SICE Linux servers. For this assignment, no other packages are allowed to be imported for use with your machine learning implementations from scratch other than numpy. There are more details regarding this in the assignment description below.

4. Your code must work on the `silo.sice.indiana.edu` server. We will test your code on this system, so this is the only way to guarantee that your program will work correctly when we run it. Minor differences in Python versions, available modules and packages, etc. between the Python on your system and that of the SICE Linux servers may cause your code to work differently and may seriously affect your grade.

Groups. You'll work in a team of 2 people for **Part 1** and **Part 2** in this assignment; we've already assigned you to a team. You can find your team on canvas by looking into people –> Groups. All the groups having prefix **A5** are the intended groups for this assignment. The group having the button ***Visit*** is your assigned group. You should only submit **one** copy of the assignment for your team, through GitHub. All the people on the team will receive the same grade, except in unusual circumstances, so we will collect feedback about how well your team functioned.

Coding style and documentation. We will not explicitly grade based on coding style, but it's important that you write your code in a way that we can easily understand it. Please use descriptive variable and function names, and use comments when needed to help us understand code that is not obvious.

Report. For this assignment, we will require a written report that summarizes your programming solutions and that answers any specific questions that we pose in the problem descriptions below. Please put the report in the `README.md` file in your GitHub repository. Reports that are located somewhere else will not be graded. For each programming problem, your report should include: (1) the answers to the questions that are asked in the assignment description, if any; (2) a brief description of how you formulated each problem; (3) a brief description of how your program works; (4) and a discussion of any problems you faced and any assumptions, simplifications, and/or design decisions you made. These comments will help us better understand your code and your implementations. They are especially important if your code does not work perfectly, since it is a chance to document the energy and thought you put into your solution that may not otherwise be reflected by the code itself. Additionally, in your report please describe (1) how you divided the work among team members, (2) contribution of each team member.

Assignment Description

Please read the guidelines above before starting on the assignment's problems as described below.

In this assignment, your main task will be to implement two machine learning classification algorithms from scratch: ***k***-nearest neighbors and multilayer perceptron (which is a class of a feedforward artificial neural networks). Your implementations from scratch for these classifiers will be tested on various datasets, and the performance of them will be compared to the respective implementations from the popular machine learning package scikit-learn. Additionally, you will have to implement some utility functions that these machine learning algorithms rely on, including activation functions for the neurons in the multilayer perceptron network and distance formulas used during k-nearest neighbors.

You are required to implement these classifiers and utility functions from scratch. As a result, you are not allowed in any capacity to use any pre-implemented packages such as scikit-learn or scipy (using numpy is fine and is in fact highly encouraged). We know that these pre-implemented packages will likely perform much better, but the point of the assignment is to dig deep into the machine learning algorithms and really learn how they work under the hood. The skeleton code we provide helps take care of details such as class and function definitions and leaves it to you to write the necessary code to implement the functions related to machine learning (that is, the functions you must implement will be clearly indicated).

Part 0: Getting Started

To get started, clone the Github repository created for this assignment. Type below command in your command line:

`git clone https://github.iu.edu/cs-b551-fa2023/assignment5.git`
Or
`git clone git@github.iu.edu:cs-b551-fa2023/assignment5.git`

(If neither command works, you probably need to set up IU GitHub ssh keys.)

For this assignment, the usage of the package numpy is highly encouraged, and the skeleton code is already written to use numpy by default because it will make this assignment much easier and make your implementations better and faster. numpy is a fundamental package for scientific computing in Python that provides a multidimensional array object (ndarray), various derived objects, and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more. Instructions for installing numpy if you do not have it on your system can be found at https://numpy.org/install/, and a quickstart guide to get yourself familiarized on how numpy works can be found at https://numpy.org/doc/stable/user/quickstart.html.

Part 1: K-Nearest Neighbors Classification

In the machine learning world, k-nearest neighbors is a type of non-parametric supervised machine learning algorithm that is used for both classification and regression tasks. For classification, the principle behind k nearest neighbors is to find k training samples that are closest in distance to a new sample in the test dataset, and then make a prediction based on those samples.

These k closest neighbors are used to try and predict the correct discrete class for a given test sample. This prediction is typically done by a simple majority vote of the k nearest neighbors of each test sample; in other words, the test sample is assigned the data class which has the most representatives within the k nearest neighbors of the sample. An alternative method for prediction is to weigh the neighbors such that the nearer neighbors contribute more to the fit than do the neighbors that are further away. For this, a common choice is to assign weights proportional to the inverse of the distance from the test sample to the neighbor. The distance can, in general, be any metric measure, but the standard Euclidean distance and Manhattan distance metrics are the most common choices.

k-nearest neighbors is also known as a non-generalizing machine learning method since it simply "remembers" all of its training data as opposed to other methods that update specific coefficients that fit a model to the training data.

What to Do. Your goal in this part is to implement a k-nearest neighbors classifier from scratch. Your GitHub repository contains the skeleton code for two files that will be used to implement the algorithm: `utils.py` and `k_nearest_neighbors.py`.

The `utils.py` file contains helpful utility functions that will be used by the machine learning algorithms. For this part, the only functions you need to concern yourself with are the functions `euclidean_distance` and `manhattan_distance`.

The `k_nearest_neighbors.py` file defines the `KNearestNeighbors` class that we will use to implement the algorithm from scratch. As you can see, the `__init__` function has already been properly implemented for you. This function is run whenever a new `KNearestNeighbors` object is created, and it checks the arguments passed in for the parameters in addition to setting up the class attributes based on those arguments. The attributes for the class itself are described in detail in the skeleton code. When creating the `KNearestNeighbors` object, the following parameters must be specified (or their default values will be used):

- `n_neighbors:` the number of neighbors a sample is compared with when predicting target class values (analogous to the value k in k-nearest neighbors).
- `weights:` represents the weight function used when predicting target class values (can be either `'uniform'` or `'distance'`). Setting the parameter to `'distance'` assigns weights proportional to the inverse of the distance from the test sample to each neighbor.
- `metric:` represents which distance metric is used to calculate distances between samples. There are two options: `'l1'` or `'l2'`, which refer to the Manhattan distance and Euclidean distance respectively.

Between these two files, there are four functions that you are required to implement. The four functions currently raise a `NotImplementedError` in order to clearly indicate which functions from the skeleton code you must implement yourself. Comment out or remove the raise `NotImplementedError(...)` lines and implement each function so that they work as described in the documentation. You may assume that the input data features are all numerical features and that the target class values are categorical features. As a reminder from the guidelines, feel free to create other functions as you deem necessary, but the functions defined by the skeleton code itself must work as intended for your final solution, and therefore you cannot change the parameters for these functions. This is required because we may call any of these functions directly when testing your code. The four functions you must implement and their descriptions are as follows:

- `euclidean_distance (x1, x2)` in `utils.py:` computes and returns the Euclidean distance between two vectors.
- `manhattan_distance (x1, x2)` in `utils.py:` computes and returns the Manhattan distance between two vectors.
- `fit(X, y)` in `k_nearest_neighbors.py:` fits the model to the provided data matrix X and targets y.
- `predict(X)` in `k_nearest_neighbors.py:` predicts class target values for the given test data matrix X using the fitted classifier model.

Testing your Implementation. We have provided you with a driver program called `main.py` that allows you to run and test your implementation of the `KNearestNeighbors` class and its associated functions. This driver program will run your implementation multiple times on two different datasets with different arguments set for the class' parameters. Alongside your implementation, the corresponding `scikit-learn` implementation will be run on the same datasets and with the same arguments, allowing you to directly compare the accuracy scores achieved by your program with those achieved by the standard `scikit-learn` implementation.

In order to run the program, you must have the following packages installed on your system: `numpy`, `pandas`, and `scikit-learn`. You should already have `numpy` installed from the previous instructions. To install `pandas`, please see the instructions at https://pandas.pydata.org/docs/getting_started/install.html. To install scikit-learn, please see the instructions at https://scikit-learn.org/stable/install.html.

To test your k-nearest neighbors implementation, enter the command shown below on your terminal.

```
python3 main.py knn
```

If you have not implemented one of the required functions (or have forgotten to remove the line that raises a `NotImplementedError`), then the driver program will terminate unsuccessfully. A successful program call, upon finishing, will result in the output shown below.

```
$ python3 main.py knn
Loading the Iris and Digits Datasets...

Splitting the Datasets into Train and Test Sets...

Standardizing the Train and Test Datasets...

Testing K-Nearest Neighbors Classification...
- Iris Dataset Progress:    [============================================================] 100%
- Digits Dataset Progress:  [============================================================] 100%

Exporting KNN Results to HTML Files...

Done Testing K-Nearest Neighbors Classification!

Program Finished!  Exiting the Program...
```

You should now see two new HTML files in your project directory. These HTML files contain tables depicting the accuracy scores of both your implementation and the `scikit-learn` implementation for all of the parameters tested. These files, called `knn_iris_results.html` and `knn_digits_results.html`, are accordingly named based on the dataset that was used for testing.

Open both files using an internet browser and you will see that a table was generated from the results of the driver program. In the table, there is a blank row in between each pair of implementations for readability. If you have implemented the four functions correctly, the accuracy scores computed for your implementation and the `scikit-learn` implementation should be very close to each other for all of the cases. Note that the accuracy scores may not be exactly the same due to slight differences in the implementations and the stochastic nature of machine learning algorithms.

Part 2: Multilayer Perceptron Classification

In machine learning, the field of artificial neural networks is often just called neural networks or multilayer perceptrons. As we have learned in class, a perceptron is a single neuron model that was a precursor to the larger neural networks that are utilized today.

The building blocks for neural networks are neurons, which are simple computational units that have input signals and produce an output signal using an activation function. Each input of the neuron is weighted with specific values, and while the weights are initially randomized, it is usually the goal of training to find the best set of weights that minimize the output error. The weights can be initialized randomly to small values, but more complex initialization schemes can be used that can have significant impacts on the classification accuracy of the models. A neuron also has a bias input that always has a value of 1.0 and it too must be weighted. These weighted inputs are summed and passed through an activation function, which is a simple mapping that generates an output value from the weighted inputs. Some common activation functions include the sigmoid (logistic) function, the hyperbolic tangent function, or the rectified linear unit function.

These individual neurons are then arranged into multiple layers that connect to each other to create a network called a neural network (or multilayer perceptron). The first layer is always the input layer that represents the input of a sample from the dataset. The input layer has the same number of nodes as the number of features that each sample in the dataset has. The layers after the input layer are called hidden layers because they are not directly exposed to the dataset inputs. The number of neurons in a hidden layer can be chosen based on what is necessary for the problem. The neurons in a specific hidden layer all use the same activation function, but different layers can use different ones. Multilayer perceptrons must have at least one hidden layer in their network.

The final layer is called the output layer and it is responsible for outputting values in a specific format. It is common for output layers to output a probability indicating the chance that a sample has a specific target class label, and this probability can then be used to make a final clean prediction for a sample. For example, if we are classifying images between dogs and cats, then the output layer will output a probability that indicates whether dog or cat is more likely for a specific image that was inputted to the neural network. The nature of the output layer means that its activation function is strongly constrained. Binary classification problems have one neuron in the output layer that uses a sigmoid activation function to represent the probability of predicting a specific class. Multi-class classification problems have multiple neurons in the output layer, specifically one for each class. In this case, the softmax activation function is used to output probabilities for each possible class, and then you can select the class with the highest probability during prediction.

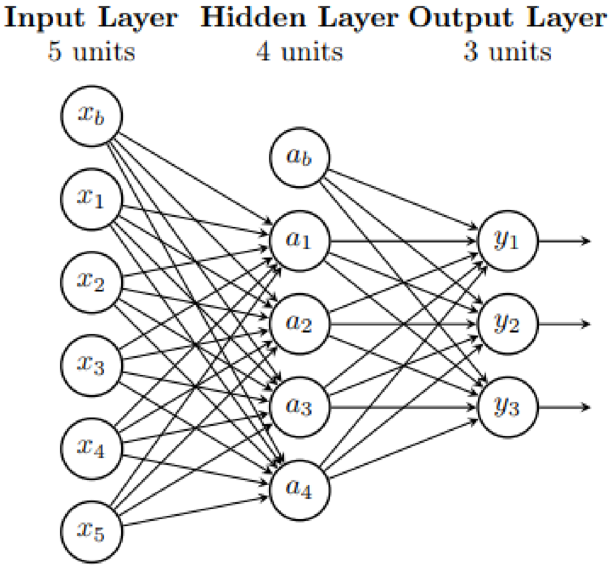
Before training a neural network, the data must be prepared properly. Frequently, the target class values are categorical in nature: for example, if we are classifying pets in an image, then the possible target class values might be either dog, cat, or goldfish. However, neural networks usually require that the data is numerical. Categorical data can be converted to a numerical representation using one-hot encoding. One-hot encoding creates an array where each column represents a possible categorical value from the original data (for the image pet classification, one-hot encoding would create three columns). Each row then has either 0s or 1 s in specific positions depending on the class value for that row. Here is an example of one-hot encoding using the dog, cat, or goldfish image classification example, where we are using five test samples and looking at their target class values.

y
dog
cat
cat
goldfish
dog

$\xrightarrow{\text{one-hot encoding}}$

y_{dog}	y_{cat}	y_{goldfish}
1	0	0
0	1	0
0	1	0
0	0	1
1	0	0

In this assignment, we will specifically be focusing on multilayer perceptron neural networks that are feedforward, fully-connected, and have exactly three layers: an input layer, a hidden layer, and an output layer. A feedforward fully-connected network is one where each node in one layer connects with a certain weight to every node in the following layer. A diagram of such a neural network is shown below, where the input layer has five nodes corresponding to five input features, the hidden layer has four neurons, and the output layer has three neurons corresponding to three possible target class values. The bias terms are also added on as nodes named with subscript of b.



Once the data is prepared properly, training occurs using batch gradient descent. During each iteration, **forward propagation** is performed where training data inputs go through the layers of the network until an output is produced by the output layer. Frequently, the cross-entropy loss is calculated using this output and stored in a history list that allows us to see how quickly the error reduces every few iterations. The output from the output layers is then compared to the expected output (the target class values) and an error is calculated. The output error is then propagated back through the network one layer at a time, and the weights are updated according to the amount that they contributed to the error. This is called **backward propagation**. A parameter called the learning rate is typically used to control how much to change the model in response to the estimated error each time the model weights are updated. Once the maximum number of iterations is reached, the neural network is finished training and it can be used to make new predictions. A prediction is made by using new test data and computing an output using forward propagation. When there are multiple output neurons, the output with the highest softmax value is chosen as the predicted target class value.

What to Do. Your goal in this part is to implement a feedforward fully-connected multilayer perceptron classifier with one hidden layer (as shown in the description above) from scratch. As before, your GitHub repository contains the skeleton code for two files that will be used to implement the algorithm: `utils.py` and `multilayer_perceptron.py`.

This time, the functions you need to concern yourself with in the `utils.py` file are the unimplemented functions defined after the distance functions. Specifically, these functions are: **`identity, sigmoid, tanh, relu, cross_entropy,`** and **`one_hot_encoding`**.

The `multilayer_perceptron.py` file defines the `MultilayerPerceptron` class that we will use to implement the algorithm from scratch. Just like the previous part, the `__init__` function has already been properly implemented for you. The attributes for the class itself are described in detail in the skeleton code. When creating the `MultilayerPerceptron` object, the following parameters must be specified (or their default values will be used):

- `n_hidden`: the number of neurons in the one hidden layer of the neural network.
- `hidden_activation`: represents the activation function of the hidden layer (can be either `‘identity’`, `‘sigmoid’`, `‘tanh’`, or `‘relu’`).
- `n_iterations`: represents the number of gradient descent iterations performed by the `fit(X, y)` method.
- `learning_rate`: represents the learning rate used when updating neural network weights during gradient descent.

Between these two files, there are nine functions that you are required to implement for this part. The nine functions currently raise a `NotImplementedError` in order to clearly indicate which functions from the skeleton code you must implement yourself. Like before, comment out or remove the raise `NotImplementedError(...)`

lines and implement each function so that they work as described in the documentation. You may assume that the input data features are all numerical features and that the target class values are categorical features. As a reminder from the guidelines, feel free to create other functions as you deem necessary, but the functions defined by the skeleton code itself must work as intended for your final solution, and therefore you cannot change the parameters for these functions. This is required because we may call any of these functions directly when testing your code. The nine functions you must implement and their descriptions are as follows:

- `identity(x, derivative = False)` in `utils.py`: computes and returns the identity activation function of the given input data `x`. If `derivative = True`, the derivative of the activation function is returned instead.
- `sigmloid(x, derivative = False)` in `utils.py`: computes and returns the sigmoid (logistic) activation function of the given input data `x`. If `derivative = True`, the derivative of the activation function is returned instead.
- `tanh(x, derivative = False)` in `utils.py`: computes and returns the hyperbolic tangent activation function of the given input data `x`. If `derivative = True`, the derivative of the activation function is returned instead.
- `relu(x, derivative = False)` in `utils.py`: computes and returns the rectified linear unit activation function of the given input data `x`. If `derivative = True`, the derivative of the activation function is returned instead.
- `cross_entropy(y, p)` in `utils.py`: computes and returns the cross-entropy loss, defined as the negative log-likelihood of a logistic model that returns `p` probabilities for its true class labels `y`.
- `one_hot_encoding(y)` in `utils.py`: converts a vector `y` of categorical target class values into a one-hot numeric array using one-hot encoding: one-hot encoding creates new binary-valued columns, each of which indicate the presence of each possible value from the original data.
- `_initialize(X, y)` in `multilayer_perceptron.py`: function called at the beginning of `fit(X, y)` that performs one-hot encoding for the target class values and initializes the neural network weights (`_h_weights`, `_h_bias`, `_o_weights`, and `_o_bias`).
- `fit(X, y)` in `multilayer_perceptron.py`: fits the model to the provided data matrix `X` and targets `y`.
- `predict(X)` in `multilayer_perceptron.py`: predicts class target values for the given test data matrix X using the fitted classifier model.

Testing your Implementation. As with the previous part, running the driver program `main.py` allows you to test your implementation of the `MultilayerPerceptron` class and its associated functions. Assuming you have already installed the three packages (`numpy, pandas,` and `scikit-learn` as discussed before), you can test your multilayer perceptron implementation by entering the command shown below on your terminal.

```
python3 main.py mlp
```

You can also test both your k-nearest neighbors implementation and your multilayer perceptron implementation one after the other by entering the following command.

```
python3 main.py all
```

You should see the following terminal output if the driver program has tested your multilayer perceptron implementation without runtime errors.

```
$ python3 main.py mlp
Loading the Iris and Digits Datasets...

Splitting the Datasets into Train and Test Sets...

Standardizing the Train and Test Datasets...

Testing Multilayer Perceptron Classification...
- Iris Dataset Progress:    [============================================================] 100%
- Digits Dataset Progress:  [============================================================] 100%

Exporting MLP Results to HTML Files...

Done Testing Multilayer Perceptron Classification!

Program Finished!  Exiting the Program...
```

Just like before, you should now see two new HTML files in your project directory, this time named `mlp_iris_results.html` and `mlp_digits_results.html`. The format of the generated tables are the same as before, but with different columns representing the parameters of the `MultilayerPerceptron` class. If you have correctly implemented the multilayer perceptron, the accuracy scores computed for your implementation and the scikit-learn implementation should be somewhat similar in most cases. However, unlike with k-nearest neighbors, **you may see some more substantial variation in the accuracy scores between the implementations** because the way that the model weights are initialized can make a big difference in the final prediction accuracy score of the model. Your implementation is likely just fine as long as you are seeing a decent number of cases where your implementation and the scikit-learn implementation accuracy scores are close to each other. If there are any concerns regarding this, feel free to make a Q&A Community post about it and the course staff will help you out.

Part 3: In Other Words...

Imagine that you've been tasked with passing an encoded message to an agent. However, this is complicated by the fact that the message must be passed discretely in a crowded area, so you want to make the message sound as ordinary as possible to those nearby.

Every fourth word in the sentence is going to contain your message, and cannot be altered. For example, if your intended message is "I got a dog", then "I waited until she GOT up to do A check on her DOG" is a fine sentence.

For this problem, you'll get practice using Hugging Face models and applying different techniques that we've covered in this course. Our intent is that you get practice developing quick solutions and improving on them with careful questions. For this reason, we won't be evaluating output accuracy, and we encourage you to keep your report succinct. Here's an outline of the steps that we recommend you follow:

1. Start by going to Hugging Face. If this is a new site for you, have a look around.
2. Select a model that looks promising, and import it. For starters, try importing ALBERT:

   ```
   from transformers import AlbertTokenizer, AlbertModel
   tokenizer = AlbertTokenizer.from_pretrained('albert-base-v1')
   model = AlbertModel.from_pretrained("albert-base-v1")
   ```

3. Now let's select a method of generating sentences. For starters, we'll try Gibbs sampling: for each word in a sample sentence, temporarily replace it with a [MASK], and see what the model suggests. Randomly select one of the words recommended, and replace the starting word with it. After repeating this process until convergence, record the frequency with which each word appears, and make a sentence using the most frequent.
4. Test a few sentences. Record your results.

By now, we have a draft of a message-encoder. For your report, include the following:

- What results did you get initially? Are they convincing?
- If you are not satisfied, is it a problem of model or sampling algorithm? Justify this! In particular, how can you tell where the problem lies?

The next step is to improve on your prior results by changing up the Hugging Face model. Import any model of your choice, and run your tests again. In your report, include the following:

- How have the results changed?
- If you saw an improvement in results, are there any drawbacks to the model you chose? When might you use each model?

Now let's refine the algorithm. You may either replace Gibbs sampling entirely, or try to improve the algorithm through any means you think reasonable. If you're not sure how to improve the algorithm, consider breaking sentences into smaller pieces, terminating before convergence, sampling from multiple sentences at once, or increasing the number of words recommended. In your report, include the following:

- What changes did you make to your algorithm? Did they help?
- What problems have both algorithms struggled to address? Do you think these problems can be dealt with?

Finally, let's consider the process as a whole. In your report, include the following:

- What changes were most impactful, relative to time spent?
- What topics or techniques would you want to spend more time practicing?

We encourage you to run through this process once more, upgrading or changing both the imported model and the algorithm, but while it can be helpful practice, this is optional.

What to turn in

For this assignment you will be creating two private repositories.

1. For **Part 1** and **Part 2**, create a private repository on GitHub under cs-b551-fa2023 using the name `userid1-userid2-a5` where the user IDs correspond to the _iu_id of team members sorted alphabetically (e.g., sanagra-sblancor-a5). Make sure that both the team members have access to the repository. You can add your team member by going to your repository –> settings –> Collaborators and teams. Submit your solution for Part 1 using this repository.
2. For **Part 3**, create a private repository using the name your_iu_id-a5 on GitHub under cs-b551-fa2023, where your_iu_id is the text before the @ sign in your IU email address (e.g., sblancor-a5). Submit your solution for Part 2 using this repository.

Make sure that both of your repositories are private (not internal or public) so others will not see your submission. Turn in the required programs on GitHub (remember to add, commit, push) - we'll grade whatever version you've put there as of 11:59:59PM on the due date. Also remember to put your report in the `README.md` file. Note that for this assignment you will be submitting two reports; one for each repository. To make sure that the latest version of your work has been accepted by GitHub, you can log into the `http://github.iu.edu` website and browse the code online. Your programs must obey the input and output formats we specify above so that we can run them, and your code must work on the SICE Linux computers.