CS760 Spring 2019 Homework 3

Due Mar 26 at 11:59pm

General instructions:

- Homeworks are to be done individually.
- For any written problems:
  - We encourage you to typeset your homework in LaTeX. Scanned handwritten submissions will be accepted, but will lose points if they're illegible.
  - Your name and email must be written somewhere near the top of your submission (e.g., in the space provided below).
  - **Show all your work**, including derivations.
- For any programming problems:
  - All programming for CS760, Spring 2019 will be done in Python 3.6. See the `housekeeping/python-setup.pdf` document on Canvas for more information.
  - Follow all directions precisely as given in the problem statement.
- You will typically submit a zipped directory to Canvas, containing all of your work. The assignment may give additional instructions regarding submission format.

Goals of this assignment. The goals of this assignment are for you to learn about stochastic gradient descent and backpropagation by

- implementing logistic regression and training and testing it on several data sets;
- implementing a neural network with one hidden layer and training and testing it on several data sets;
- evaluating the effect of the amount of training on the accuracy of the learned models.

Programming Problems

Your program can assume that all datasets will be provided in JSON files, structured like this example:

```
{
    'metadata': {
                    'features': [ ['feature1', 'numeric'],
                                  ['feature2', ['cat', 'dog', 'fish'],
                                  ...
                                  ['class', ['+', '-']
                                ]

                },

    'data':    [[ 3.14, 'dog', ... , '+' ],
                [ <instance 2> ],
                ...
                [ <instance N> ]]
}
```

That is, the file contains *metadata* and *data*. The metadata tells you the names of the features, and their types.

Real and integer-valued features are identified by the 'numeric' token. Categorical features are identified by a list of the values they may take.

The data is an array of feature vectors. The order of features in the metadata matches their order in the feature vectors.

JSON files are easy to work with in Python. You will find the `json` package (and specifically the `json.load` function) useful.

Part 1

(40 pts) For this part of the homework, you will implement logistic regression using a neural network and stochastic gradient descent. Logistic regression is a powerful generalized linear model that is used quite often in practice.

- For this part, you should assume the following:
  - In the JSON files that we provide, the class attribute is named `'class'` and it is the last attribute listed in the feature section.
  - Your network is intended for binary classification problems only, and therefore it has one output unit with a sigmoid function. The sigmoid should be trained to predict 0 for the first class in the `'class'` attribute, and 1 for the second class.
  - You should use **stochasic gradient descent** to minimize the **cross-entropy error**, that is you should update the weights after each training instance.

  - For each numeric feature, you should use one input unit. For each categorical feature, you should use a *one-hot* encoding.
  - You should standardize numeric features as described in this document.
  - Each epoch refers to one complete pass through the training instances. The datasets that are provided to you are randomly shuffled already, hence for our autograding, please **do not** randomize the order of the training and testing instances anywhere.

- Your program should be called `logistic`, and must be callable from a bash terminal, as follows:

  ```
  $ ./logistic <learning-rate> <#epochs> <train-set-file> <test-set-file>
  ```

  That is,

  - you ought to have an executable script called `logistic`;
  - the 2nd argument specifies the learning rate;
  - the 3rd argument specifies the number of training epochs;
  - the 4th argument is the path to a training set file;
  - and the 5th argument is the path to a test set file.

  You *must* have this call signature—otherwise, the autograder will not be able to analyze your implementation correctly.

- After training for `#epochs` epochs on the training set, your program should use the learned neural net to predict a classification for every instance in the test set. Your program should output the following in order:

  - After each training epoch, print the epoch number (starting from 1), the cross-entropy error summed across the training instances, the number of training instances that are correctly classified, and the number of instances that are misclassified. Print these four values on one line with a space between each of them. To determine a classification, use a threshold of 0.5 on the activation of the output unit (i.e. the value computed by the sigmoid). The output for cross-entropy error should have **12 digits of precision**.

    You should print in the following format (given n epochs):

    ```
    1 <cross-entropy error 1> <#correctly_classified> <#misclassified>
    2 <cross-entropy error 2> <#correctly_classified> <#misclassified>
    ...
    n <cross-entropy error n> <#correctly_classified> <#misclassified>
    ```

  - After training, for each test instance, print the activation of the output unit, the predicted class, and the correct class on one line with a space between each of them. The output for activation should have **12 digits of precision**.

    You should print in the following format (given N test instances):

    ```
    <instance 1's activation> <instance 1's predicted class> <instance 1's true class>
    <instance 2's activation> <instance 2's predicted class> <instance 2's true class>
    ...
    <instance N's activation> <instance N's predicted class> <instance N's true class>
    ```

– Print the number of correctly classified and the number of incorrectly classified test instances when a threshold of 0.5 is used on the activation of the output unit. The two values are separated by a space.

  You should print in the following format:

  ```
  <#correctly_classfied> <#incorrectly_classfied>
  ```

– Finally, output the F1 score for the model with **12 digits of precision**. F1 score is defined here. The equation is given by

$$\frac{2 \times precision \times recall}{precision + recall}$$

  You should print in the following format:

  ```
  <F1_score>
  ```

- Here is an example:

```
$ ./logistic 0.01 3 train.json test.json
1 132.123456789012 500 200
2 131.341456509013 501 199
3 130.849572638495 502 198
0.905985077339 1 1
0.672020558219 1 1
...
0.852904361771 1 0
80 20
0.873849509238
```

Details of implementation:

Here are some details on the implementation, you are required to follow these requirements, otherwise, your output may not be the same as ours.

1. When implementing *one-hot* encoding for categorical attributes, you are required to do this based on the order of that attribute's possible values shown in the metadata of each JSON file.

   For example, given attribute `A`, which has possible values `['a', 'b', 'c']` shown in metadata, for any instance that has `'a'` for attribute `A`, we encode it as `[1, 0, 0]`.

2. In order to make sure your program is deterministic, you need to call `numpy.random.seed` function. For more details of this function, you can look at this.

   Specifically, for this assignments, at the entrance of your program, you should include this line:

   ```
   numpy.random.seed(0)
   ```

   Here is an example of adding this line in your code:

   ```
   import numpy as np
   import sys

   if __name__ == "__main__":      # the entrance of your program
       np.random.seed(0)
       main(sys.argv)              # your code starts here
   ```

3. You need to add one bias unit to the input units. Specifically, for each instance, you should implement the bias parameter by adding a feature, **at index 0 of the feature vector**, that is always set to 1. This should be done after you have standardized the numeric features and one-hot encoded the categorical features.

4. You are required to generate the weights using the following code to insure your output to be consistent with our reference output:

   ```
   import numpy as np
   w = np.random.uniform(low=-0.01, high=0.01, size=(1, #input_units)))
   ```

   That is,

   - `#input_units` specifies the number of input units, including the bias unit.
   - `w` is the weight vector from input units to the output unit. It has size $1 \times$ `#input_units`, where `w(i)` denotes the weight from the ith input unit to the output unit.
   - All weights and bias parameters are initialized to random values in [-0.01, 0.01]. Hence, `low=-0.01` and `high=0.01`.

5. For each instance, the order of the features should not be changed. That means you need to do standardization and one-hot encoding in place.

Part 2

(50 pts) For this part, you will expand upon the framework you developed for logistic regression, and implement a neural network with a single, fully connected hidden layer, that uses cross-entropy as the loss function.

- You can make the exactly same assumptions as described in Part 1.
- Your program should be called `nnet`, and must be callable from a bash terminal, as follows:

  ```
  $ ./nnet <learning-rate> <#hidden-units> <#epochs> <train-set-file> <test-set-file>
  ```

  That is,

  - you ought to have an executable script called `nnet`;
  - the 2nd argument specifies the learning rate;
  - the 3rd argument specifies the number of hidden units;
  - the 4rd argument specifies the number of training epochs;
  - the 5th argument is the path to a training set file;
  - and the 6th argument is the path to a test set file.

  You *must* have this call signature—otherwise, the autograder will not be able to analyze your implementation correctly.

- You can reuse any of the functionality you developed for Part 1. However, we will need the two separate scripts `logistic` and `nnet` that we can call separately for each part.
- The output for this part has exactly the same format as Part 1. Please refer to the output formats and examples in Part 1.

Details of implementation:

The details on the implementation for this part are the **same** as Part 1 (you need first to read the details of implementation in Part 1), **except** the following:

1. Apart from adding one bias unit for the input layer, you also need to add one bias unit at the top of hidden units, that is, for each instance, add a 1 to the index 0 of the vector of hidden values.
2. You are required to generate the weights using the following code to insure your output to be consistent with our reference output:

   ```
   import numpy as np
   w_i_h = np.random.uniform(low=-0.01, high=0.01, size=(#hidden_units, #input_units)))
   w_h_o = np.random.uniform(low=-0.01, high=0.01, size=(1, #hidden_units + 1))
   ```

   That is,

   - `#input_units` specifies the number of input units, including the bias unit.
   - `#hidden_units` specifies the number of hidden units, not including the bias unit. `#hidden_units` is also the same as the argument `#hidden-units` from the command line.
   - `w_i_h` is the weight vector from input units to the hidden units. It has size `#hidden_units` $\times$ `#input_units`, where `w_i_h(i,j)` denotes the weight from the jth input unit to the ith hidden unit.
   - `w_h_o` is the weight vector from hidden units to the output unit. It has size $1 \times$ `(#hidden_units + 1)`, where `w_h_o(i)` denotes the weight from the ith hidden unit to the output unit.
   - All weights and bias parameters are initialized to random values in [-0.01, 0.01]. Hence, `low=-0.01` and `high=0.01` for both lines.

Part 3

(10 pts) For this part, you need to choose **only one** of the datasets that are provided to you, and plot the following curves:

1. For the logistic regression network built in Part 1, **plot two curves on one graph**, 1) F1 score versus number of epochs on training dataset, and 2) F1 score versus number of epochs on testing dataset.

   Specifically, you need to select a `max_epoch`, which is the maximum number of epochs, and a learning rate. For each e from 1 to `max_epoch`, train the network for e epochs, and measure the F1 score on the training and testing datasets separately. Plot the two curves of F1 score versus e for both datasets on one graph.
2. Follow the same steps and plot two curves on one graph for the fully connected neural network as described in Part 2.
3. Discuss your findings on the above curves. For example, how does the F1 score change when the number of epochs increases, does overfitting occur, does one network outperform the other, etc.

Notes:

- You can use any tools, any programming languages to plot the curves. You can also randomly shuffle the datasets if you want. You are free to choose any one of the datasets that are provided.
- You are required to choose the same datasets and the same `max_epoch` throughout both curves, however the learning rate can be different. `max_epoch` should be at least 20.
- You need to include the plots along with their parameters (`max_epoch`, learning rate, number of hidden units, the datasets you choose, etc), and your discussion in hw3.pdf.

Additional Notes

Submission instructions

Organize your submission in a directory with the following structure:

```
YOUR_NETID/
    # your scripts
    logistic
    nnet

    # plots and discussion
    hw3.pdf

    # your source files
    <your various *.py files for part 1 and part 2>
```

Zip your directory (`YOUR_NETID.zip`) and submit it to Canvas.

The autograder will unzip your directory, `chmod u+x` your scripts, and run them on several datasets. Their results will be compared against our own reference implementation.

Resources

Executable scripts

We recommend writing your scripts in bash, and having them call your python code.

For example, your script `logistic` might look like this (given your source code named `logistic.py`):

```
#! /bin/bash

python logistic.py $1 $2 $3 $4
```

If this doesn't make sense to you, try reading this tutorial:
`http://matt.might.net/articles/bash-by-example/`

Datasets

We've provided three datasets for you to experiment with:

- `banknote*.json`. This is the banknote authentication data set.

  See the UCI repository for more info:
  `https://archive.ics.uci.edu/ml/datasets/banknote+authentication`

- `heart*.json`. This is the Heart Disease Data Set.

  See the UCI repository for more info:
  `https://archive.ics.uci.edu/ml/datasets/heart+Disease`

- `magic*.json`. This is the MAGIC Gamma Telescope Data Set.

  See the UCI repository for more info:
  `https://archive.ics.uci.edu/ml/datasets/magic+gamma+telescope`

We will provide reference output for these datasets - you will be able to check your own output against them. During grading, your code will be tested on these datasets as well as others.