CS760 Spring 2019 Homework 1

Due Feb 21 at 11:59pm

General instructions:

- Homeworks are to be done individually.
- For any written problems:
  - We encourage you to typeset your homework in LaTeX using this file as template (i.e., use pdflatex). 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.

Name: YOUR NAME
Email: YOUR EMAIL

Goals of this assignment. Your goals are to implement your first learning method and explore several methodological concepts we have covered in class, including

1. using a validation set for model selection,
2. characterizing predictive accuracy as a function of training set size using a learning curve, and
3. characterizing tradeoffs between false positive and true positive rates using an ROC curve.

Problem Set

1. (60 pts) Implement a kNN classifier satisfying these requirements:

   - Your implementation must be callable from a bash terminal, as follows:

     ```
     $ ./knn_classifier <INT k> <TRAINING SET> <TEST SET>
     ```

     That is, you ought to have an executable script called `knn_classifier`;
     the 2nd argument is an integer (the k parameter);
     the 3rd argument is the path to a training set file;
     and the 4th 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.

   - 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'],
                                       ...
                                       ['label', ['+', '-']
                                     ]

                     },

         '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. Your program can assume that the last feature is named 'label', and corresponds to our prediction target.

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.

- Your classifier should compute distances as described on page 4 of the `instance-based-learning.pdf` slides.

  I.e., standardize the numeric features, and then compute distance as the sum of (i) Manhattan distances for numeric features and (ii) Hamming distance for categorical features.

  When standardizing, be sure to compute μ and σ from the *training set*, and then store them for use during prediction.

- The `knn_classifier` executable script should predict labels for the test set and then print them to standard output. For example, we should be able to catch its output in a text file by running

```
$ ./knn_classifier <INT k> <TRAINING SET> <TEST SET> > outfile.txt
```

  Output should have the following format:

```
<label 1 votes>,<label 2 votes>,...,<label m votes>,<predicted label>
<label 1 votes>,<label 2 votes>,...,<label m votes>,<predicted label>
...
```

  That is, for an m-class classification task each row should contain m integers (which sum to k), followed by the predicted class label. These should be separated by commas (no spaces).

  For illustration: pretend that we have three classes: cat, dog, and fish. Then our output may look like this:

```
$ ./knn_classifier 5 train.json test.json
0,2,3,fish
4,1,0,cat
1,2,2,dog
(etc.)
```

- Whenever there is a tie between classes, choose the class listed first in the metadata.
- Don't worry about implementing a KD-tree. Try using numpy and pandas for fast array-based computations.

2. (15 pts) Tune the hyperparameter k, using a validation set. Then use a test set to estimate the kNN's accuracy on unseen instances, using the optimal value of k. Requirements:

   - You must provide an executable script called `hyperparam_tune`, with the following call signature:

     ```
     $ ./hyperparam_tune <max k> <TRAIN SET> <VAL. SET> <TEST SET>
     ```

     The second argument is a maximum value for the k hyperparameter. We will search over every value $1 \leq k \leq$ `max k`.

     The third, fourth and fifth arguments are paths to training, validation, and test datasets respectively. They will be in the same JSON format as in part 1.

     This script will perform the following computations.

   - For each value of k, train your kNN classifier on the training set and compute its accuracy on the *validation set.* Select the value of k which maximizes validation accuracy. In the case of a tie, choose the smallest value of k maximizing validation accuracy.

   - Train your kNN classifier on the **combined** training and validation sets. Then—using the optimal k selected just previously—compute your classifier's accuracy on the *test set.*

     This gives us an estimate of the classifier's accuracy on unseen instances.

   - Your `hyperparam_tune` script should print results to standard output, in the following format:

     ```
     1,<val. accuracy>
     2,<val. accuracy>
     ...
     <k max>,<val. accuracy>
     <optimal k>
     <test accuracy>
     ```

     I.e. there should be a row for each k in our hyperparameter search, containing k and the validation accuracy for that k. They are separated by a comma, with no spaces.

     These will be followed immediately by one row containing a single integer: the optimal value of k.

     Lastly, there will be a row containing a single number: the test set accuracy.

   - Produce a plot of the hyperparameter search: validation accuracy as a function of k. Use the `digits_*.json` datasets. Save your plot as a PDF: `tune_k.pdf`. Be sure to label your axes!

3. (10 pts) Generate a learning curve for the kNN classifier. Requirements:

   - You must provide an executable script called `learning_curve`. It should have the following signature:

     ```
     $ ./learning_curve <INT k> <TRAINING SET> <TEST SET>
     ```

     where each argument is the same as in previous problems.

   - Ordinarily, a learning curve would be generated by training on *random* subsamples of the training set. Furthermore, we would provide error bars for points on the learning curve by performing *multiple* subsamples.

     However, for purposes of grading we will generate a learning curve from *deterministic* subsets of the training data. And each point on the learning curve will be created from one such subset.

     Specifically: your script must generate a learning curve using **fractions of the training set in increments of 10%.** That is, you will compute the kNN's test set accuracy after training on the first 10%, 20%, ..., 90%, and 100% of the training set.

     For each of these fractions, the number of training examples should be **rounded down**.

     To make this mathematically explicit: if the training set is $X_1, \ldots, X_N$, then you should construct your learning curve by training the kNN on

     $$X_1, \ldots, X_{\lfloor \frac{i \cdot N}{10} \rfloor} \qquad \forall i \in \{1, 2, \ldots, 10\}$$

     and computing its test set accuracy for each i.

   - Your `learning_curve` script should print its results to standard output, in the following format:

     ```
     <number of training examples at 10%>,<test set accuracy>
     <number of training examples at 20%>,<test set accuracy>
     ...
     <number of training examples at 100%>,<test set accuracy>
     ```

     That is, it will produce 10 lines of output, with each line containing two numbers: (i) the number of training examples used and (ii) the test set accuracy obtained with that amount of the training data.

     The numbers are separated by a comma, with no spaces.

   - Produce a plot comparing learning curves for multiple values of k. Use the `digits_{train,test}.json` datasets.

     Does one value of k do best for all training set sizes? Or do different ks seem to work better for different training set sizes?

     Save your plot as a PDF: `learning_curve.pdf`. Again: label your axes.

4. (15 pts) Generate an ROC curve for the kNN classifier. Requirements:

- You must provide an executable script named `roc_curve` with the following signature:

  ```
  $ ./roc_curve <INT k> <TRAINING SET> <TEST SET>
  ```

- **Assume that the first class listed in the JSON file is the positive class.** ROC curves are only defined for binary classification. Hence, your program can assume a two-class setting.

- ROC requires that your classifier compute *confidence values*. However, a kNN classifier does not do this by default. Compute confidence values in the following way:

  $$P(y = 1|x) = \frac{\sum_{n \in neighbors(x)} w_n y_n}{\sum_{n \in neighbors(x)} w_n}$$

  where

  $$w_n = \frac{1}{d(x, x_n)^2 + \epsilon}.$$

  This is essentially kNN regression as described in the slides. In the special case where $y \in \{0, 1\}$, we can interpret this as *confidence of positive classification.*

  Set $\epsilon = 1 \times 10^{-5}$ in the denominator. Adding an ϵ to the denominator prevents issues when test examples coincide with training examples.

- Compute the ROC curve using the procedure given in the lecture slides.

- Your script should print the ROC curve to standard output in the following format:

  ```
  <FPR>,<TPR>
  <FPR>,<TPR>
  ...
  <FPR><TPR>
  ```

  Each line has two numbers: the False Positive Rate and the True Positive Rate.

  The numbers are separated by a comma, with no spaces.

- Generate a plot comparing ROC curves for two or three values of k. Use the `votes_{train,test}.json` datasets.

  Does one value of k seem to dominate the others in ROC space? Or should we choose different ks as the cost of false positives/false negatives vary?

  Save your plot as a PDF: `roc_curve.pdf`. Label your axes.

Additional Notes

Submission instructions. Organize your submission in a directory with the following structure:

```
YOUR_NETID/
    # your scripts
    knn_classifier
    hyperparam_tune
    learning_curve
    roc_curve

    # your plots
    tune_k.pdf
    learning_curve.pdf
    roc_curve.pdf

    <your various *.py files>
```

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.

Executable scripts. Writing one in bash:
`http://lmgtfy.com/?q=how+to+write+an+executable+script+bash`

Writing one in python:
`http://lmgtfy.com/?q=how+to+write+an+executable+script+python`

One additional thing, though: if you do use python for your executable script, specify the python version in the she-bang line. E.g.,
`#!/usr/bin/python3.6`
The she-bang line points to a *system* python, ignoring virtual environments.

Datasets. We've provided two datasets for you to experiment with:

- `digits_*.json`. These are pixel intensities for 8×8 images of handwritten digits. The task is to predict the digit (0-9) from the image.

  See the UCI repository for more info:
  `https://archive.ics.uci.edu/ml/datasets/optical+recognition+of+handwritten+digits`

- `votes_*.json`. These are the voting records of congresspeople in the US House of Representatives. The task is to predict the politician's political party (Democrat, Republican) from their votes.

See the UCI repository for more info:
`https://archive.ics.uci.edu/ml/datasets/congressional+voting+records`

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.