CS 686-02 - Assignment 4 Spring, 2018

Assignment 4 - Implementation of k-Nearest Neighbors

The goal of this assignment is a) to implement a simple version of k-Nearest Neighbors algorithm using an object-oriented framework and b) to test this implementation against a dataset.

Background

We have been using one base class, `classifier`, for our classification algorithms. Not counting the constructor, this class has two functions: `fit` and `predict`:

- The `fit` function accepts a list of (training) features and their labels as parameters, changing the internal state of the classifier as needed.
- The `predict` function accepts a list of (test) features and outputs the classifier's hypothesis, in order, about the class membership of the instance represented by the features.

The k-Nearest Neighbors algorithm is one such classification approach which is useful when the response is non-linear with respect to one or more independent variables. The `fit` function caches the feature vector and labels, along with any hyperparameters, including "`k`" to be used when making predictions. The `predict` function hypothesizes the label of each item in the input by determining the most populous cached training label among the "`k`" nearest instances.

Requirements

You will complete this assignment by performing the following three (3) requirements:

1. Implement "`knn.py`", a k-Nearest Neighbors class in object-oriented python. Your knn.py must be a subclass of classifier, https://github.com/dbrizan/cs686-2018-01/blob/master/classifier.py. Specifically, it must have three functions at minimum: `__init__(self, k=3)`, `fit(self, X, Y)` and `predict(self, X)`... where "`self`" refers to the class instance, "`X`" refers to the list of feature vectors in train or test and "`Y`" refers to a list of labels associated with each feature vector. The "`k`" in the `__init__` function refers to the size of the neighborhood to be searched. Note that it is an optional parameter, defaulting to 3. Your functions may take any number of additional parameters as long as they are optional.
2. Test your "`knn.py`" implementation by running it against the file PhishingData.arff, located at http://archive.ics.uci.edu/ml/machine-learning-databases/00379/PhishingData.arff. This data contains a label for a website — legitimate (1), suspicious (0) or phishy (-1) — based on whether the site contains pop-up windows, etc. This file is in Attribute Relation Fire Format (ARFF). Details of the format: http://weka.wikispaces.com/ARFF. Your implementation must read PhishingData.arff, which will be contained in the same folder as your source file. Other than

`knn.py`, any modules other need not be object oriented, but they are expected to conform to reasonable guidelines of structured programming and style, including variable naming and code documenting. See "PEP 8: Style Guide for Python Code" for one guide on how to do so.

3. Determine and output the accuracy of your classifier in `knn.py` when executed with (hyperparameter) k varying between 2 and 32 and output the accuracy associated with each k. Your training data will be the first 80% of the data (all but the last 271 instances). Your test data will be the remaining data. The (actual) results for some values of K are shown in Table 1.

K	Accuracy
2	0.87457
3	0.88565
4	0.88935
5	0.88194
6	0.87089
7	0.87458
8	0.87088
9	0.88198
10	0.88564
11	0.88565
12	0.88933

Table 1: Accuracies for various values of K

Submission

Submit your source code on Canvas. (While you are free and encouraged to store your implementation on github, you may not submit your github repo.)

Grading

The requirements are serially dependent — i.e. completing the third item depends on completion of the second, which depends on completion of the first. You are required and expected to complete all items above perfectly. As such, your grade starts as 100%, and the penalties for errors are listed below. Your grade may be affected by one or more penalty.

- -10% = Implementation lacks reasonable style for code or documentation.
- -15% = Implementation works but contains minor errors (eg. output contains minor deviations from expected results).
- -25% = Implementation is missing one required item.
- -50% = Implementation is missing two required items.
- -100% = Implementation not attempted or not submitted on time.