

## Homework 02

### 1. k-Nearest Neighbors classification

Recall from class that, given training data  $(X, y)$  and a query point  $x$ , kNN finds the  $k$  closest training points to  $x$  – denoted  $\mathcal{N}_k(x)$  – and takes a majority vote over their labels to predict the label of  $x$ . Said formally, the first nearest neighbor to  $x$  is given by

$$\mathcal{N}_1(x) = \operatorname{argmin}_i \|x - X_i\|^2; \quad (1)$$

likewise, for  $k > 1$ , one returns a length- $k$  vector of the indices of the  $k$  closest points to  $x$ . The predicted label of  $\hat{y}$  is then calculated as

$$\hat{y} = \operatorname{argmax}_c \sum_{i \in \mathcal{N}_k(x)} \delta_{y_i, c}, \quad (2)$$

where  $\delta_{y_i, c}$  is the Kronecker delta (1 if  $y_i = c$ , 0 if  $y_i \neq c$ ) and  $c$  runs over class labels.

First, write your own code for kNN. You should clearly implement the following three functions:

- i. `knn_find_nearest(X, x, k)`: Given training data in the N-by-D matrix **X** and a D-dimensional query point **x**, return the indices (of **X**) of the **k** nearest neighbors of **x**.<sup>1</sup>
- ii. `knn_predict(X, y, x, k)`: Given training data in the N-by-D matrix **X** and N-dimensional vector **y**, along with a D-dimensional query point **x**, return the predicted value **yhat** of the query point as the mode of the labels of the **k** nearest neighbors of **x**.
- iii. `knn_cross_validate(X, y, kvec, split)`: Given data in the N-by-D matrix **X** and the N-dimensional vector **y**, partition the data into (randomized) training and testing sets, where **split** specifies the percentage of the data to use for training. **kvec** is a vector of the values of **k** for which to run the cross-validation. Return the training and test error for each value in **kvec**.

Use classification error (the fraction of correctly classified examples) for the training/test error.

Apply your kNN code to hand-written digit classification, as demonstrated in class. The data are available from the “Elements of Statistical Learning” website at:

---

<sup>1</sup>It’s worth a bit of thought here to implement this in a “vectorized” manner in either Python, R, or MATLAB, which will drastically decrease code runtime.

<http://www-stat.stanford.edu/~tibs/ElemStatLearn/data.html>

(See “ZIP Code” at the bottom of the page.)

Use the data in `zip.train.gz` to cross-validate over  $k=\{1,2,3,4,5\}$  with an 80/20 train/test split, selecting an optimal value of  $k$  as that which minimizes the test error. Using this value of  $k$ , evaluate your final performance on the data in `zip.test.gz` and present your results in a (10-by-10) confusion table, showing the counts for actual and predicted image labels. In addition, quote the runtime and accuracy for your results.

In addition to digit recognition, apply your kNN code to spam classification for the spambase data at the UCI repository:

<ftp://ftp.ics.uci.edu/pub/machine-learning-databases/spambase/>

Using an 80/20 train/test split, present a confusion table with results on the raw features. Then standardize the features to be mean-zero and variance-one and present the results, again as a (2-by-2) confusion table. Report the runtime and accuracy as well.

## 2. Curse of dimensionality

This problem looks at a simple simulation to illustrate the curse of dimensionality. First you’ll compare the volume of a hypersphere to a hypercube as a function of dimension. Then you’ll look at the average distance to nearest points as a function of dimension.

For  $D$  from 1 to 15 dimensions, simulate 1000 random  $D$ -dimensional points, where the value in each dimension is uniformly randomly distributed between -1 and +1. Calculate the fraction of these points that are within distance 1 of the origin, giving an approximation of the volume of the unit hypersphere to the cube inscribing it. Plot this fraction as a function of  $D$ . Also, use the value of this fraction at  $D = 2$  and  $D = 3$  to get estimates for the value of  $\pi$ , as you know the volume formulae for these cases.

Repeat this simulation, sampling 1000  $D$ -dimensional points from 1 to 15 dimensions, where the value in each dimension is uniformly randomly distributed between -1 and +1. For each value of  $D$ , generate an additional 100 query points and calculate the distance to each query point’s nearest neighbor, using your kNN code. Plot the average distance from the query points to their nearest neighbors as a function of  $D$ .

## 3. Command line exercises

Using the included `words` file<sup>2</sup>, implement one-line shell commands using `grep`, `wc`, `sort`, `uniq`, `cut`, `tr`, etc. to accomplish the following:

- Count the total number of words

---

<sup>2</sup>often available in `/usr/dict/words` or `/usr/share/dict/words` on \*nix machines

- Count the number of words that begin with **A** (case sensitive)
- Count the number of words that begin with **A** or **a** (case insensitive)
- Count the number of words that end with **ing**
- Count the number of words that begin with **A** or **a** and end with **ing**
- Count the number of words that are exactly four letters
- Count the number of words containing four or more vowels
- Count the number of words that have the same vowel consecutively repeated two or more times (e.g. **aardvark** or **book**)
- Display all words which contain a **q** that is not followed by a **u**
- Count the number of words beginning with each letter of the alphabet (case sensitive) <sup>3</sup>; display the result as **count letter**, one letter per line
- Count the number of words beginning with each letter of the alphabet (case insensitive) <sup>4</sup>; display the result as **count letter**, one letter per line

You're welcome to use alternative standard GNU tools (e.g., **sed** or **awk**) if you like, but the above can be implemented with only the commands listed above.

Submit both a shell script that executes these commands as well as the output produced by them as plain text.

---

<sup>3</sup>Although there are many ways to accomplish this, exploring the manpage for **cut** may reveal a hint for a relatively simple implementation

<sup>4</sup>Again, while many solutions exist, using **tr** with the **A-Z** and **a-z** character sets may prove useful.