Tutorial: Digit Recognition

The Kaggle competition *Digit Recognizer* provides an interesting data set for k-nearest-neighbor prediction. The data were donated in a competition to correctly label optically-scanned handwritten digits. For details, navigate to the website https://www.kaggle.com/c/digit-recognizer. The data consist of 42,000 digitized images of digits, one record per image. Accordingly, the group label of a particular optical image must be one of the set $\{0, 1, \ldots, 9\}$. The predictor vector extracted from an optical image consists of darkness intensity[5] measured on each of the $28 \times 28 = 748$ pixels comprising the optical image. The prediction problem is to use the vector of 748 darkness values obtained from an unlabeled image and correctly predict the handwritten digit.

[5] Darkness is recorded on a scale of 0–255.

The tutorial guides the reader through the programming of a `Python` script that will estimate the accuracy of the conventional and exponentially weighted k-nearest neighbor prediction functions for the task posed above. Our approach to accuracy estimation is to draw a subset R from the data set D and use it to build a conventional k-nearest-neighbor prediction function and an exponentially weighted k-nearest-neighbor prediction function. A second subset E, disjoint from R, is drawn to evaluate the accuracy of the prediction functions. The subsets R and E are referred to as the training and test sets, respectively. We estimate accuracy by obtaining a prediction $\widehat{y}_i = f(\mathbf{x}_i|R)$ from each observation $\mathbf{z}_i \in E$. A comparison of the predicted and actual labels yields the proportion of correct predictions. Insuring that the training and test sets are disjoint is important because using training observations to evaluate the accuracy of a prediction function poses a significant risk of overestimating accuracy.

But more can be done with the results than computing the proportion of correctly labeled test observations. The result of a prediction, say, $\widehat{y}_i = f(\mathbf{x}_i|R)$ may be one of $g = 10$ values, as may be the actual value. Thus, there are 10×10 possible combinations of outcomes. It may be that some digits are more frequently confused than others. For instance, 3 and 8 might be confused more often than 1 and 8. To extract some information on the types of errors that are most likely, we'll cross-classify the predictions against the actual target labels by tabulating the number of times that each combination is observed. The table containing the cross-classification of actual and predicted target values is called the *confusion matrix*. Let's suppose that a prediction is computed for $\mathbf{z}_0$ and that the actual label of y_0 is j and the predicted label is h. Then, the count in row j and column h will be incremented by one. After processing a reasonably large number of test observations, we'll have an understanding of the type of errors incurred by the prediction function. A convenient way of determining the number of correctly classified test observations is to compute the sum of the diagonal elements of the confusion matrix. Since every test observation is classified once, the accuracy is estimated by the sum of the diagonal divided by the sum over the entire table.

In the tutorial, we'll build a three-dimensional array consisting of two back-to-back $g \times g$ confusion matrices, one for the conventional k-nearest-neighbor prediction function and the second for the exponentially weighted k-nearest-neighbor prediction function. The label pairs $\{(y_i, \widehat{y}_i)|\mathbf{z}_i \in E\}$ provide the data that fills the confusion matrix.

The data set is large by conventional standards with 42,000 observations and consequently, execution time is slow for the k-nearest neighbor prediction functions. To speed development and testing of the `Python` code, the tutorial does not use all of the observations in D.

The principal four steps of the tutorial are as follows.

1. Create training and test sets. The prediction functions will be constructed from, or trained on, the training set R and tested on the test set E. The formation of R and E is accomplished by systematically sampling

the Kaggle training set, `train.csv`. Though `train.csv` is small enough (66 MB) that in-memory storage is feasible, the tutorial instructs the reader to process the file one record at a time. In any case, the first task is to draw training and test sets of $n_R = 4200$ and $n_E = 420$ observations respectively from the data set.

2. Construct a function f_{order} that will determine the neighbors of $\mathbf{z}_0 \in E$, ordered with respect to the distance between $\mathbf{x}_0$ and $\mathbf{x}_1, \ldots, \mathbf{x}_{n_R}$. The function will return the ordered labels of the neighbors. More formally, the function will compute an ordered arrangement of the training observation labels $\mathbf{y}^o = (y_{[1]}, \ldots, y_{[n_R]})$, hence, $\mathbf{y}^o = f_{\text{order}}(\mathbf{x}_0|R)$.
3. Write a function f_{pred} that computes two predictions of y_0 from $\mathbf{y}^o$ using the conventional and exponentially weighted k-nearest-neighbor prediction functions, respectively. The three arguments passed to f_{pred} are $\mathbf{y}^o$, the neighborhood size k, and the vector of weights $\mathbf{w} = [w_1 \ \cdots \ w_{n_R}]^T$ The weight vector determines the exponentially weighted k-nearest-neighbor function in the same way that k determines the conventional k-nearest-neighbor function.
4. Fill the confusion matrix with the outcome pairs $\{(y_i, \widehat{y}_i)|\mathbf{z}_i \in E\}$. To do so, every test observation $\mathbf{z}_i = (y_i, \mathbf{x}_i) \in E$ will be cross-classified according to its actual (y_i) and predicted label $(\widehat{y}_i)$. Accuracy estimates for the two prediction functions will be computed from the matrix.

Detailed instructions follow.

1. Download `train.csv` from https://www.kaggle.com/c/digit-recognizer. The file is rectangular in the sense that aside from the first record containing the variable names, each record has the same number of attribute values. There are $748 = 28^2$ attributes, each of which is a measurement of darkness for one pixel in a 28×28 field. Attributes are comma-delimited.
2. Initialize dictionaries `R` and `E` to store the training and test sets, respectively. Read the data file one record at a time. The first record contains the column names. Extract that record using the `readline` attribute of `f` before iterating over the remainder of the file. Print the variable names. Iterate over the file and print the record counter `i`.

```
import sys
import numpy as np
R = {}
E = {}
path = '../train.csv'
with open(path, encoding = "utf-8") as f:
    variables = f.readline().split(',')
    print(variables)
    for i, string in enumerate(f):
        print(i)
```

On each iteration, `i` is incremented by using the `enumerate` function.

3. As the file is processed, build the dictionaries `R` and `E`. The dictionary keys are record counters, or indexes, and the dictionary values will be pairs consisting of a target value y and a predictor vector $\underset{748\times 1}{\mathbf{x}}$. Store $\mathbf{x}$ as a list.

   Add observation pairs to the training dictionary whenever the count of processed records is a multiple of 10. Add pairs to the test dictionary whenever $i \bmod 100 = 1$. None of the training pairs will be included in the test set.

```
if i%10 == 0:
    record = string.split(',')
    y = int(record[0])
    x = [int(record[j]) for j in np.arange(1, 785)]
    R[i] = (y, x)
if i%100 == 1:
    record = string.split(',')
    y = int(record[0])
    x = [int(record[j]) for j in np.arange(1, 785)]
    E[i] = (y, x)
```

   Note that the first element in `record` identifies the digit and therefore is the target value of the ith observation. This code segment must execute every time that a record is read.

4. Initialize an array `confusionArray` to store the results of predicting the test targets using the conventional and exponentially weighted k-nearest-neighbor prediction functions. Since there are two prediction functions (conventional and exponentially weighted k-nearest-neighbor prediction functions), we need two 10×10 confusion matrices to count the occurrences of each combination. It's convenient use one three-dimensional array to store the two confusion matrices as side-by-side 10×10 arrays.

   Initialize the constants and the storage array for the confusion matrices.

```
p = 748   # Number of attributes.
nGroups = 10

confusionArray = np.zeros(shape = (nGroups, nGroups, 2))
acc = [0]*2  # Contains the proportion of correct predictions.
```

   The code segment executes upon completion of building the dictionaries `R` and `E`.

5. Create an n_R-element list containing the labels of the training observations.

```
labels = [R[i][0] for i in R]
```

6. Set the neighborhood size k and smoothing constant α. Construct the n_R-element list of weights for the exponentially weighted k-nearest-neighbor prediction function.

```
nR = len(R)
k = 5
alpha = 1/k
wts = [alpha*(1 - alpha)**i for i in range(nR)]
```

   Check that $\sum_i w_i = 1$ by summing the elements of `wts`.

7. The program flow for the prediction task is shown in the next code segment. We iterate over the observation pairs in the test set `E` and extract a predictor vector and label on each iteration. Each predictor vector ($\mathbf{x}_0$) is passed with the training set `R` to the function `fOrder`. The function `fOrder` returns the ordered training labels $\mathbf{y}^o$ as a list named `nhbrs`. The list `nhbrs` is passed to `fPredict` to compute predictions $\widehat{y}_{\text{conv}}$ and $\widehat{y}_{\text{exp}}$ using the conventional and exponentially weighted k-nearest-neighbor prediction functions, respectively. The function `fPredict` returns a two-element list `yhats` containing $\widehat{y}_{\text{conv}}$ and $\widehat{y}_{\text{exp}}$. The `for` loop indexed by `j` updates the confusion matrices and computes the estimated accuracy rates as the proportion of correctly classified test observations.

```
yhats = [0]*2
for index in E:
    y0, x0 = E[index]

    #nhbrs = fOrder(R,x0)
    #yhats = fPredict(k,wts,nhbrs)
    for j in range(2):  # Store the results of the prediction.
        confusionArray[y0,yhats[j],j] += 1
        acc[j] = sum(np.diag(confusionArray[:,:,j]))
               /sum(sum(confusionArray[:,:,j]))
    print(round(acc[0],3),'  ',round(acc[1],3))
```

   In the segment code above, accuracy is estimated after each test observation is processed as a means of tracing the execution of the program. The functions `fOrder` and `fPredict` do not exist at this point, of course. Introduce the code segment into your script. To test the code, temporarily set `yhats = [y0, y0]`. Execute the script and verify that `acc` contains 1.0 in both positions.

8. It remains to implement the k-nearest-neighbor prediction function. It's best to implement the code not as a function, but in the main program because variables computed inside functions are local and cannot be referenced outside the function. When you're satisfied that the code is correct, then move the code to a function *outside* of the loop.

   The first function to program is `fOrder`. Code it within the `for` loop that iterates over the test set `E`. Its purpose is to create the ordered vector of labels $\mathbf{y}^o$ from a test vector $\mathbf{x}_0$ and the training set `R`. Ordering is determined by the distances of each training vector to $\mathbf{x}_0$. We'll compute the distances in a `for` loop that iterates over `R`. On each iteration of the `for` loop, compute the distance between $\mathbf{x}_0$ and $\mathbf{x}_i \in R$. Save the distances in a list named `d`:

```
d = [0]*len(R)
for i, key in enumerate(R):
    xi = R[key][1]  # The ith predictor vector.
    d[i] = sum([abs(x0j - xij) for x0j, xij in zip(x0,xi)])
```

   The distance `d[i]` is computed by zipping the vectors `x0` and `xi` together. Using list comprehension, we iterate over the zip object and build a list containing the absolute differences $|x_{0,j} - x_{i,j}|$, for $j = 1, \ldots, p$. The last operation computes the sum of the list.
9. Compute a vector $\mathbf{v}$ that will sort the distances from smallest to largest. That is, it is a vector of indexes such that

$$\mathtt{d[v[0]]} \leq \mathtt{d[v[1]]} \leq \mathtt{d[v[2]]} \leq \cdots.$$

   The vector $\mathbf{v}$ will also sort the vector of training observation labels so that the label of the nearest observation is `labels[v[0]]`, the label of the second nearest observation is `labels[v[1]]` and so on. The `Numpy` function `argsort` computes $\mathbf{v}$ from $\mathbf{d}$. Using `v`, list comprehension is used to compute the ordered neighbors $\mathbf{y}^o$, or `nhbrs`.

```
v = np.argsort(d)
nhbrs = [labels[j] for j in v] # Create a sorted list of labels.
```

   This code segment executes *after* `d` has been filled.
10. Test the code by printing the labels of the k-nearest neighbors and the label y_0. There should be good agreement—for most test observations the majority of neighbors should have the same group label as y_0.
11. When the code appears to function correctly, move the `fOrder` function outside of the `for` loop. This function computes the distances between $\mathbf{x}_0$ and $\mathbf{x}_i \in R$ and the ordering vector `v`. Lastly, it arranges the neighbors according to the distances (instructions 8 and 9). The definition and return statements are

```
def fOrder(R,x0):
    ...
    return nhbrs
```

The function call is

```
nhbrs = fOrder(R,x0)
```

12. Move the code described in instructions 8 and 9 to the function definition. Call the function instead of executing the code in the main program. Check that the nearest k neighbors usually match the target y_0.

13. The next step is to determine which group is most common among the k-nearest neighbors. This task will be performed by the function `fPred`. As with `fOrder`, write the code in place. When it works, move it out of the main program and into a function.

    Begin with the conventional k-nearest-neighbor prediction of y_0. Initialize a list of length g to store the number of the k-nearest neighbors that belong to each of the g groups. Count the number of nearest neighbors belonging to each group:

```
counts = [0]*nGroups
for nhbr in nhbrs[:k]:
    counts[nhbr] += 1
```

    Since `nhbr` is a group label, and either $0, 1, \ldots, 8$ or 9, the statement `counts[nhbr] += 1` increments the count of the k-nearest neighbors belonging to the group membership of the neighbor.

14. The exponentially weighted k-nearest-neighbor prediction function estimates the group membership probabilities $\widehat{\Pr}(y_0 = j|\mathbf{x}_0)$ for each group $j \in \{0, \ldots, g-1\}$. The calculation uses the list of weights `wts` computed in instruction 6. Iterate over the n_R neighbors of $\mathbf{z}_0$ using `i` as an index. Accumulate the weights for each group by adding w_i to the group to which $\mathbf{z}_{[i]}$ belongs.

    Use the `for` loop coded in instruction 13 but iterate over *all* of the neighbors instead of ending with the kth neighbor. The code segment is

```
counts = [0]*nGroups
probs = [0]*nGroups
for i, nhbr in enumerate(nhbrs): # Iterate over all neighbors.
    if i < k:
        counts[nhbr] += 1
    probs[nhbr] += wts[i]        # Increment the probability of
                                 # membership in the nhbr's group.
```

Execute the code and print `probs` and the sum of `probs` as each test observation is processed. The sums (`sum(probs)`) must be equal to 1.

15. Identify the most common group among the k-nearest neighbors for the conventional k-nearest-neighbor prediction function. Also determine the index of the largest estimated probability for the exponentially weighted k-nearest-neighbor prediction function:

```
yhats = [np.argmax(counts), np.argmax(probs)]
```

The `Numpy` function `np.argmax(u)` identifies the index of the largest element in `u`. If more than one value is the maximum in `counts`, the `Numpy` function `argmax` identifies the first occurrence of the maximum. It's desirable to break ties in some other fashion.

Test the code by printing `counts` and `yhats`. The first value of `yhats` should index the largest count. The two predictions contained in `yhats` should be in agreement most of the time.

16. Build the function `fPred` using the code segment developed in instructions 13 through 15. The arguments passed to the function are `nhbrs` and `nGroups` and the function returns `yhats`.
17. Compute and print the overall accuracy estimates as the program iterates over the test set observations. Accuracy estimates that are less than .75 are improbable for these data and suggest programming errors.
18. Double n_R and n_E and execute the script.