THU AI Assignment III: Classfication

Rui Wang*
2015010445

1 Question 1: KNN

1.1 Overview

KNN(K-Nearest Neighbors) is a simple algorithm that compares the input with all training data stored, and find k closest to it. The data type that makes up the greatest portion of KNN will be regarded as of the same type as the input.

The only problem I encountered was a misunderstanding: The input in this program is not intended to be a single digit. Rather, it is a *list* of digits. And our task is to complete the classification for all of them.

1.2 Introduction to numpy, a python package

Numpy is a python module for array operation. According to my own experience, it is just like Matlab which helps users do matrix operations in a convenient, fast way.

Numpy has been a major obstacle in this assignment. The most common issue is that a one-dimensional row vector is not considered a matrix. Thus is can not be transposed. Possible ways are to use *a[:, None]* or *numpy.array([a]).T* to first change *a* into a two-dimensional array, then take its transpose.

Also, when taking inner product of two arrays, we have to make the following considerations:

- The inner product of two one-dimensional row array is just the sum of every corresponding element pair multiplied.
- The inner product of a row vector times a column vector yields an array type *[[number]]*, instead of a single int *number*.

2 Question 2: Perceptron

2.1 Implementation Notes

The algorithm for perceptron is clear. However, there are some common pitfalls.

- The program uses mini-batch for training. For common mini-batch, a batch of training data is chosen randomly at a time for one step. However, in this implementation, we shuffle all(5000) training data and rearrange them into 100 groups. We use one group to update a step at a time. This is different from common batch algorithm found online, though they are essentially the same.
- The division of two integers in python will always yields an integer! Pay attention when doing division operations.

*wangrui15@mails.tsinghua.edu.cn

- Always pay attention to the dimension of arrays. W is a 784 by 10 matrix, while b is 1 by 10. Data to interpret, on the other hand, is 100 by 784. The orders of dimension(e.g. $784 \times 100 V.S. 100 \times 784$) might be different from that in mathematical derivation.

3 Question 3: Support Vector Machines

3.1 Overview

Implementing SVM requires us to follow the instruction file closely.

The kernel trick here could be tricky. Without understanding it, we should at least know that the generation of this kernel actually involves both the data to be identified and the training data stored.

3.2 Implementation Notes

One common misunderstanding here is also about the input. Test will give 1000 cases altogether. We need to test each of them instead of one of them.

Also, the description for *optimizeConstrainedQuad* is incorrect. Both x and b parameters for this function are row vectors, so the description should be $min 1/2 x A x^T + bx$. Without recognizing this, the input dimension would be wrong. This requires us to look into the function *optimizeConstrainedQuad* and figure out what input parameters are really intended to be.

The symbol names in the given python file might be confusing. The variables that I used in the end are *alpha, bias* and all the training data for kernel generation. These data end up stored in variables whose names do not actually match their meanings.

4 Question 4 & 5: Principal Component Analysis

4.1 Basic Notations

First we clarify some notations and definitions. Data is an m by n matrix, of n column data vectors in space $\mathbf{R}^M$. We are required to find $l (l \leqslant m$, denoted as *self.dimension* in out python program) principal components for a vector.

For a common case like the digit recognition here, typically $n = 5000$ is the size of the training set, $m = 784$ is the total dimension of original feature, while $l = 32$ is the reduced feature dimension.

4.2 Relationship with Singular Value Decomposition

On lecture slides, SVD is introduced but the relationship between SVD and PCA is not illustrated. Here I find an easy-to-understand way to explain why SVD can do the same task as PCA while saving the time for eigenvalue calculation.

Note that the covariance matrix $\Sigma = \frac{1}{N} X_c {X_c}^T$. In PCA theory, we need to calculate the l$(l \leqslant m)$ eigenvectors with largest eigenvalues $u_1, u_2, ..., u_l$. Now let's suppose we have already found the SVD of X_c. We have $X_c = U \Sigma V^T$. Substitute this equation into $X_c {X_C}^T$ and we obtain:

$$X_c {X_C}^T = U \Sigma^2 U^T$$

Therefore, U is the eigenvectors of $X_c {X_C}^T$. The first l columns of U are thus the basis vectors for the principal components.

To obtain the coordinates of a vector x in u_i space, we do the inner production of x, u_i. Projecting these coordinates back to $\mathbf{R}^M$ and we obtain $\hat{x} = \sum_{i=1}^{l} (x^T u_i) ui$. This is called the **reconstruction** of x.

4.3 Implementation Notes

There are several details to be taken into careful consideration when implementing this algorithm.

- Matrix dimension. *trainingData* is n by m in our python program. However, in the algorithm, *data* has to be a matrix of column vectors, namely a m by n matrix. So we need to take the transpose of *data* before SVD.
- The SVD function *np.linalg.svd(data)* yields a result like this:

  $$[U, \Sigma', V^{T\prime}]$$

  where U is an m by m matrix, Σ is a row vector of all σs on the diagonal of the Σ in SVD's mathematical expression. When option *full_matrices* is disabled, the $V^{T\prime}$ will be the first m rows of the original V^T. Therefore, *np.linalg.svd(data, full_matrices = False)* gives us sufficient information for projecting data on its m principal directions. That's why we are disabling the *full_matrices* option.

4.4 Miscellaneous Discussion

There was a pseudo-bug that cost me more than an hour to pin down: that the result of an SVD algorithm is not UNIQUE. Instead, we can invert any of the elements of corresponding U and V, while still obtaining the valid result.

To explain in a more natural way, the direction of the basis vector can always be inverted while maintaining their orthogonality. Thus the "features" of a vector can vary in their signs. However, When the initial *autograder.py* compared the features it extracted, it did not take into consideration this possibility.

One suggested method is to simply compare the result of reconstruction with principal components extracted. This result should be unique.

The *autograder.py* was later updated to tackle this problem.