Project 1: KNN

Wentao Li, G43589062
Resources: numpy, pandas, sklearn, matplotlib, Kaggle PIMA and MNIST datasets
Programming laguage: python

Part1—Pima Indians Diabetes Database

—to see part 2 MNIST please jump to page 10

Dataset details:

The original dataset "diabetes.csv" is from https://www.kaggle.com/uciml/pima-indians-diabetes-database

I spilt the dataset into 3 parts train, validation and test in proportion of 8:1:1.(614:77:77) Furthermore, I use both 10 times cross validation and 9 times random spilt ways to avoid overfitting.

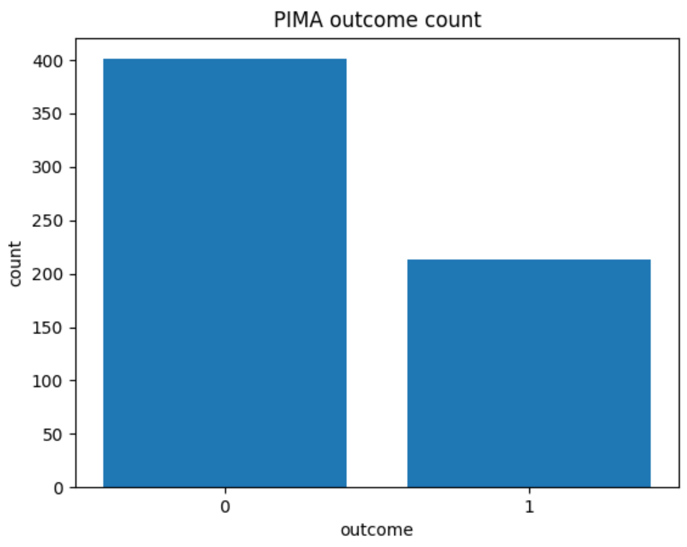
There is the analysis of train dataset first. It is believed that validation and test should not be taken into consideration or the design model may become overfitted.

1. Y_label count

    In the 614 train datas, 401 datas' Y_label or "outcome" are "0" and 213 datas' Y_label are "1" which is showing in the figure.

The 0_count / total_count = 65.3%
The 1_count / total_count = 34.7%
The 1_count / 0_count = 53%

In the following analysis, I take the proportion of 1_count / total_count which is 34.7% as an important indicator.



2. Features distribution and percentage of “outcome == 1”

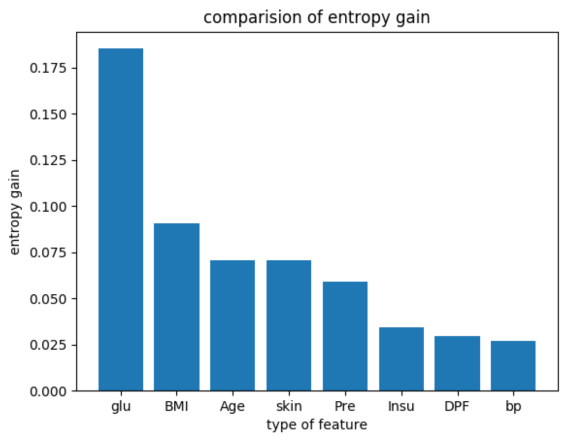
Since we know the percentage of 1_count / total_count in the train dataset is 34.7 %, We can easily calculate the entropy of the datasets by following code:

```
Entropy = -(get / total) * math.log2(get / total) - (notGet / 
total) * math.log2(notGet / total)
```

Also, we can calculate the entropy gain by the following code:

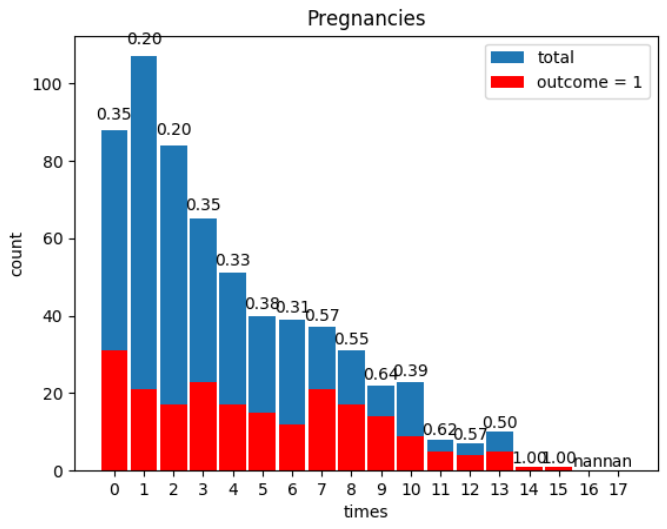
```
for subI in range(featureSize):
    cur -= globalCount[i][subI] * (probabilitySet[i][subI][0] * 
math.log2(probabilitySet[i][subI][0]) + probabilitySet[i][subI]
[1] * math.log2(probabilitySet[i][subI][1]))
res.append(totalEntropy - cur)
```

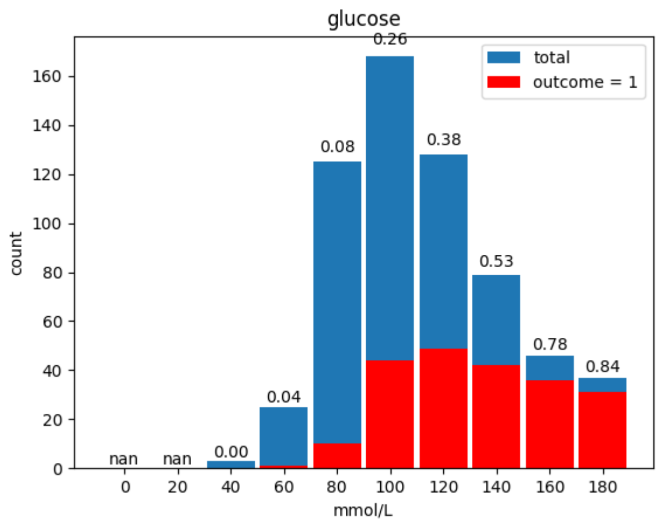
Where the subI is the index of each gap in a feature, cur is the entropy after choosing a feature, totalEntropy is the Entropy we get previously, and the difference of them is the entropy gain. Although I will not use apply decision tree model in this task, using entropy gain as feature’s weight (normalization) is still a efficient way to improve KNN model.

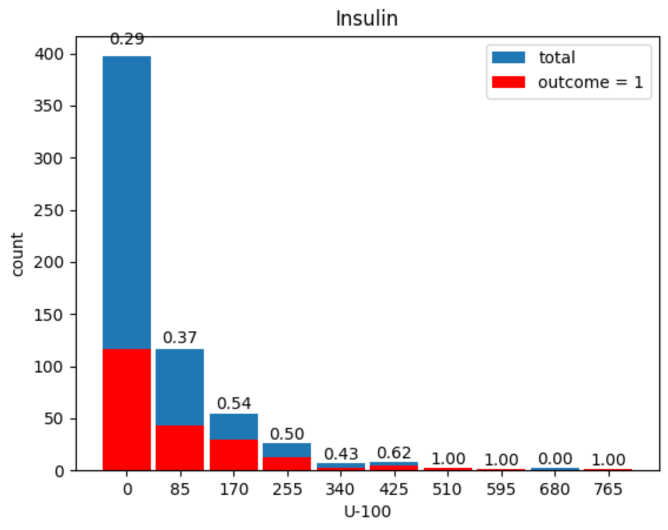


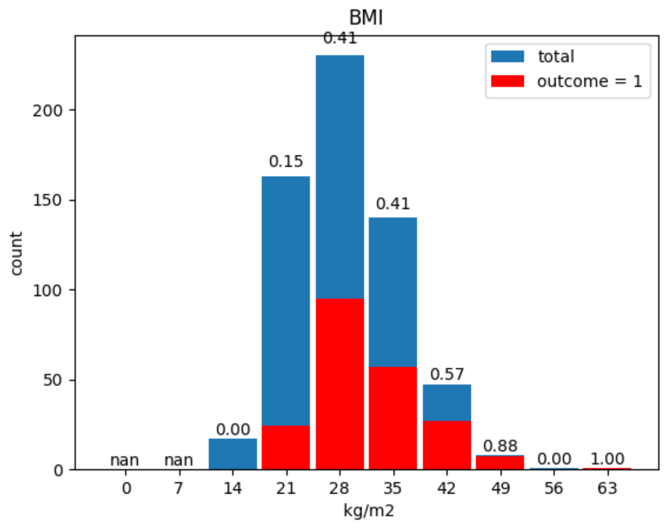
“Feature influential rank”

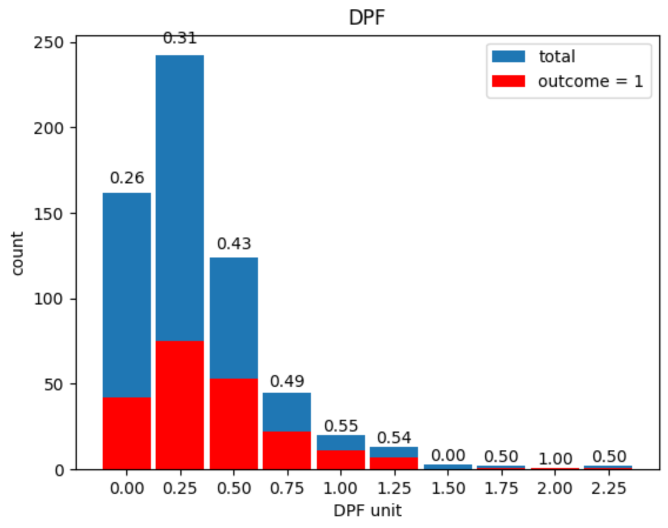
The following figures will mainly show the distribution of gap in each feature and their 1_outcome/ total proportion. By the Impurity Function and Entropy knowledge we can know that the distance from each bar’s proportion to total proportion (34.7%) is a measurement of Impurity rate. A lower impurity and high density feature which is far away to the 34.7% in other word should be given more attention.

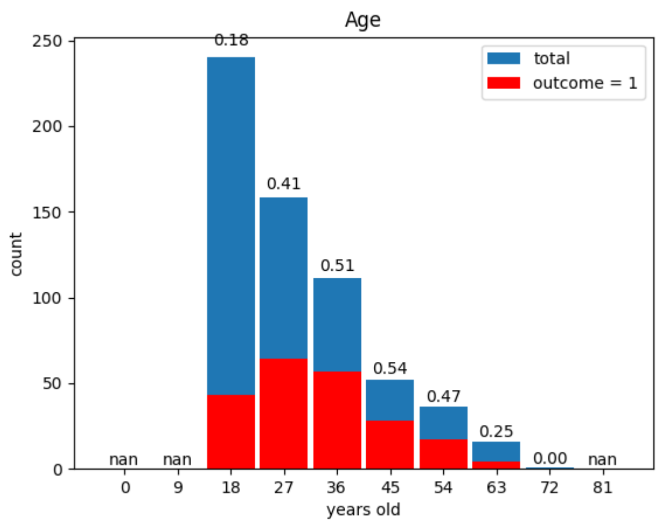

Pregnancies
total
outcome = 1
count
times
0.35 0.20 0.20 0.35 0.33 0.38 0.31 0.57 0.55 0.64 0.39 0.62 0.57 0.50 1.00 1.00 nan nan
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

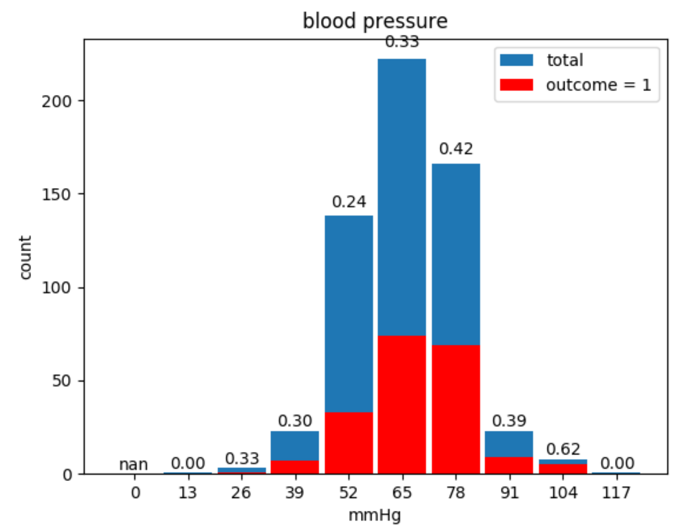

glucose
total
outcome = 1
count
mmol/L
nan nan 0.00 0.04 0.08 0.26 0.38 0.53 0.78 0.84
0 20 40 60 80 100 120 140 160 180

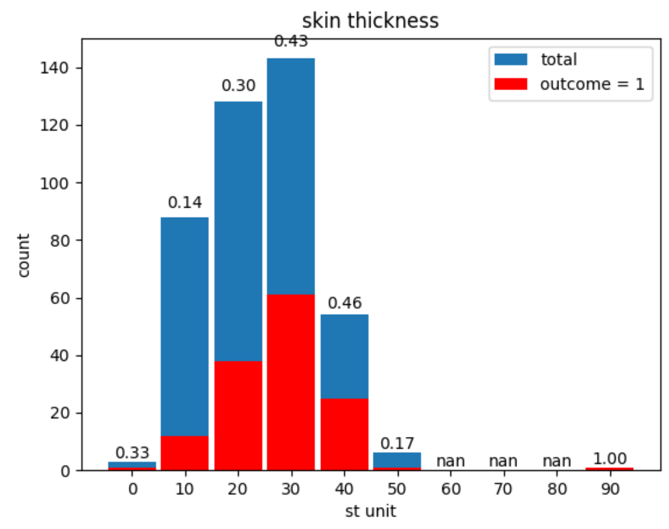

Insulin
total
outcome = 1
count
U-100
0.29 0.37 0.54 0.50 0.43 0.62 1.00 1.00 0.00 1.00
0 85 170 255 340 425 510 595 680 765


BMI
total
outcome = 1
count
kg/m2
nan nan 0.00 0.15 0.41 0.41 0.57 0.88 0.00 1.00
0 7 14 21 28 35 42 49 56 63

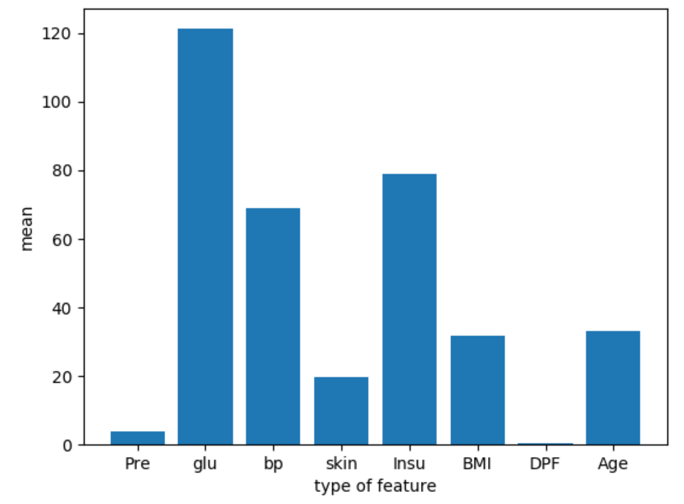

DPF
total
outcome = 1
count
DPF unit
0.26 0.31 0.43 0.49 0.55 0.54 0.00 0.50 1.00 0.50
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25

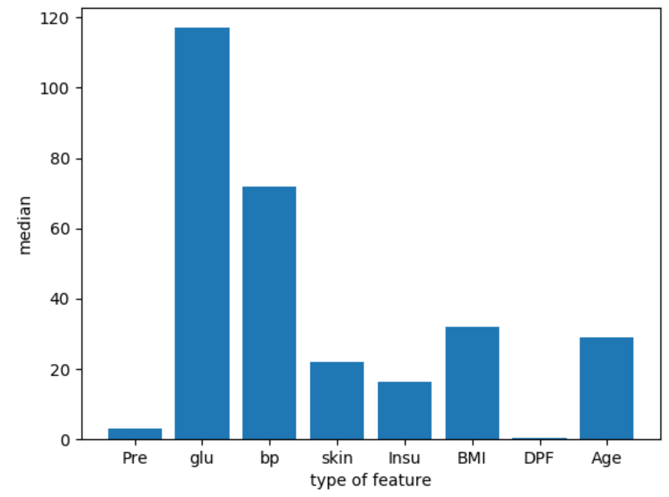

Age
total
outcome = 1
count
years old
nan nan 0.18 0.41 0.51 0.54 0.47 0.25 0.00 nan
0 9 18 27 36 45 54 63 72 81

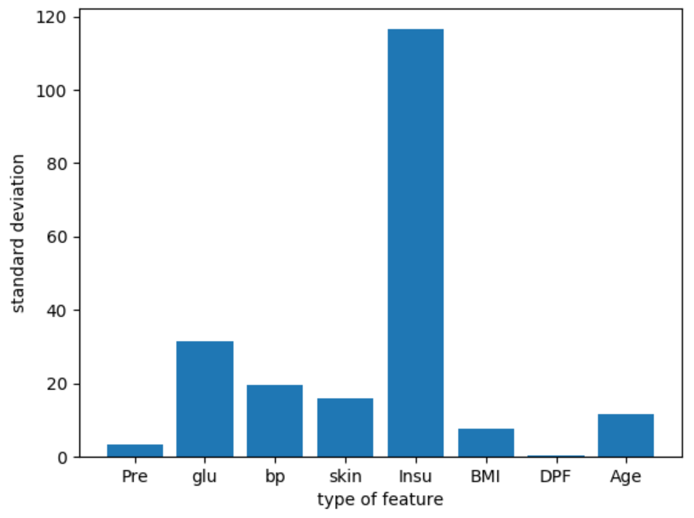




3. Mean. Median and Standard deviation





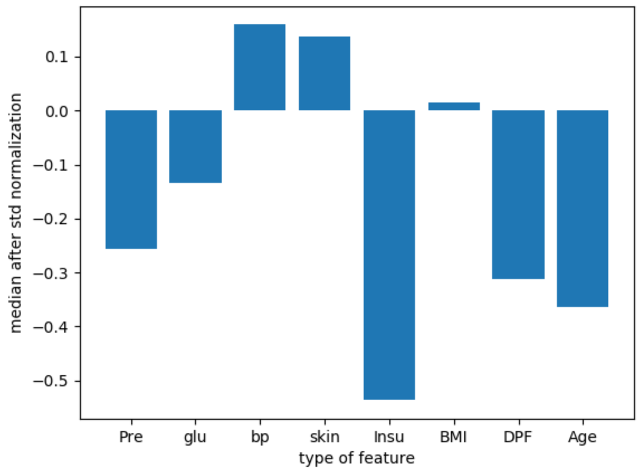


4. Min-Max normalization, standard score

Standard score:

$$\frac{X-\mu}{\sigma}$$

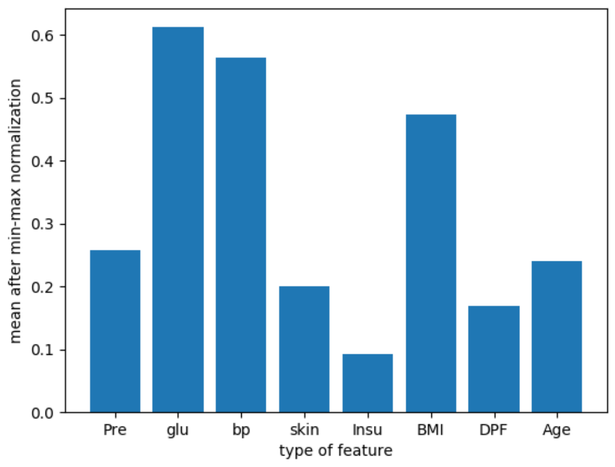
Normalizing errors when population parameters are known. Works well for populations that are normally distributed [1]

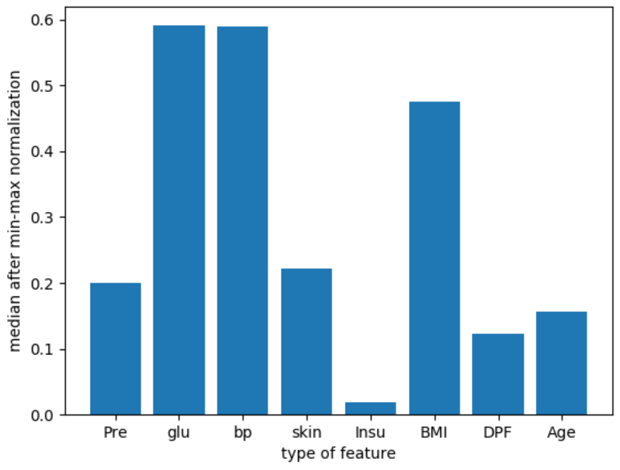


Min-max normalization (Feature scaling):

$$X' = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

Feature scaling is used to bring all values into the range [0,1]. This is also called unity-based normalization. This can be generalized to restrict the range of values in the dataset between any arbitrary points and using.





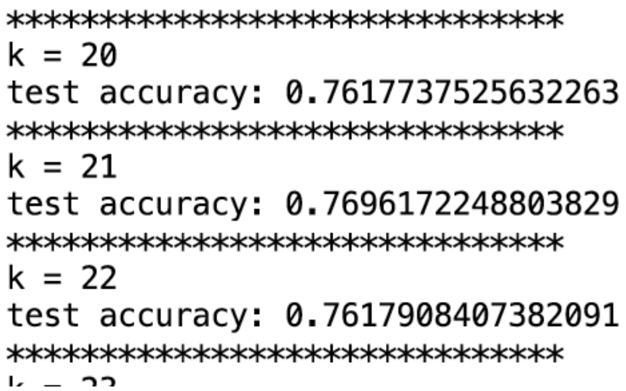
Generally speaking, in Pima test, standard score = unit length > min-max normalization.
One of the reasons is that these feature's models are highly correspond to normal distribution as we can see in the previous distribution figures.

5. Cross validation V.S. random validation
In this part, I use the package function:
1. "Sklearn.model_selection.train_test_split" for random validation
2. "Sklearn.model_selection.KFold" for cross validation

With k from 1 to 30 and each for 10 loops test. The effect of these two validation method is almost same.

The main idea of cross validation is like that you have a list and a window and you move the window for n times to go through the list. Each time window stay in a place, the inside datas are treated as validation part while the other are treated as the training part.

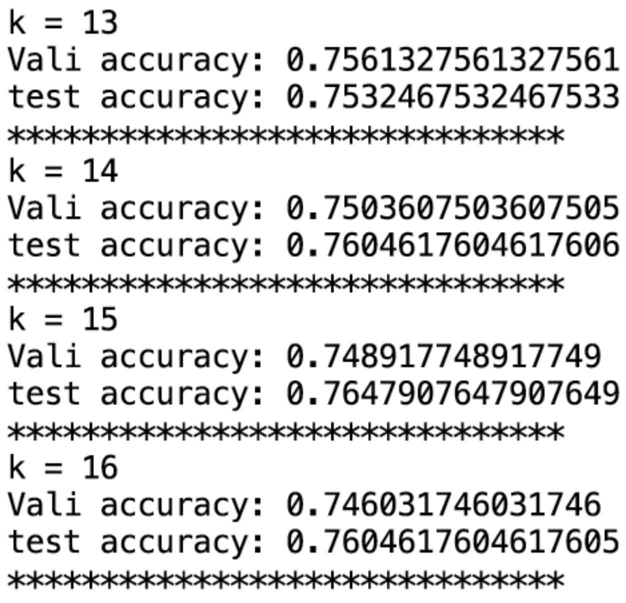
The main idea of random validation is just use a random.randint to pop(randint) from a data stack or something like this and use these datas as validation and use the remain part as training part.



```
******************************
k = 20
test accuracy: 0.7617737525632263
******************************
k = 21
test accuracy: 0.7696172248803829
******************************
k = 22
test accuracy: 0.7617908407382091
******************************
```

👆 Part of cross validation result

Part pf random validation result 👉



```
k = 13
Vali accuracy: 0.7561327561327561
test accuracy: 0.7532467532467533
******************************
k = 14
Vali accuracy: 0.7503607503607505
test accuracy: 0.7604617604617606
******************************
k = 15
Vali accuracy: 0.748917748917749
test accuracy: 0.7647907647907649
******************************
k = 16
Vali accuracy: 0.746031746031746
test accuracy: 0.7604617604617605
******************************
```

6. About "nan" value

Obviously, blood pressure, skin thickness and Insulin should not be 0 therefore these "nan" datas should be abandoned at first. However, KNN is a comparing process and if we do not take some "nan" data into consideration, it will let the distance at that feature be 0 in anyway which is not what we want at all. So I decide to set the nan data as the median value of the dataset at a specific feature which will make the model more generate.
After changing model with "nan" detecting and refitting in following way:

1. Check “nan” , remove it and memo the position
2. Replace the “nan” val with median val of the feature

The prediction accuracy boosts about 1% on average. Not too much, through, it makes the model more steady in the 10 times cross validation which means that the recall and precision is more close in some degree。

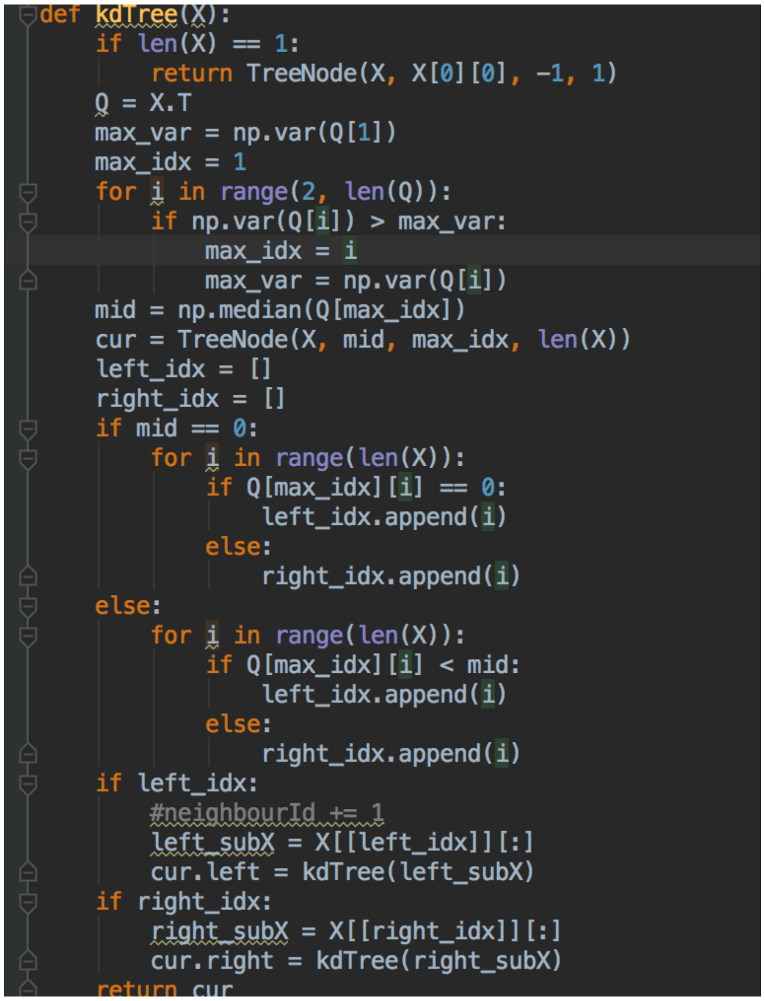
Algorithm Description:

Pre-processing:

Kd-tree. Pima only has 700 datas and each data with 8 features and a binary label which is very small. Thus we may not need to consider the problem of time complexity. However, we can add a Kd-tree algorithm as a pre-processing database. In this preprocessing, we first construct a binary tree root node. Then loop and spilt the our dataset and set the subsets in the subtrees. These loops will end until we can not spilt any subsets.

After the construction of kd-Tree. We can use search algorithm to find the most nearest >k subset of the tree for the validation data. After that, with a hash queue and backtracking algorithm, we can easily find the k-most nearest datapoint and use the major label directly as our predict result.

Parts of preprocessing codes:



```
def kdTree(X):
    if len(X) == 1:
        return TreeNode(X, X[0][0], -1, 1)
    Q = X.T
    max_var = np.var(Q[1])
    max_idx = 1
    for i in range(2, len(Q)):
        if np.var(Q[i]) > max_var:
            max_idx = i
            max_var = np.var(Q[i])
    mid = np.median(Q[max_idx])
    cur = TreeNode(X, mid, max_idx, len(X))
    left_idx = []
    right_idx = []
    if mid == 0:
        for i in range(len(X)):
            if Q[max_idx][i] == 0:
                left_idx.append(i)
            else:
                right_idx.append(i)
    else:
        for i in range(len(X)):
            if Q[max_idx][i] < mid:
                left_idx.append(i)
            else:
                right_idx.append(i)
    if left_idx:
        #neighbourId += 1
        left_subX = X[[left_idx]][:]
        cur.left = kdTree(left_subX)
    if right_idx:
        right_subX = X[[right_idx]][:]
        cur.right = kdTree(right_subX)
    return cur
```

However, we only can get about 100 validation or test data in this PIMA prediction task. This kind of algorithm needs a long time to construct the tree which is not useful.

Feature-scaling:

1. As I said before, in this pima task. The traditional normalization ways should be rank in, **unit length = std score** > min-max normalization(accuracy on average, **77%:77%:58%**). At the

same time, replacing the **“nan”** datas with its’ **feature median** will also improve about 1% accuracy rate.

2. In Pima design, I also used the first time **entropy gains** as scaling weights directly and it works very well which improve the accuracy from **71% to 77%.** The method to use this entropy gain as weight have been introduced in detail in page 2.

Finally, the scaling value should be update at both train set and validation set or we can not get a real answer.

```
curDistance -= ((X[t_idx][f_idx] - X_val[v_idx][f_idx]) *
weight[f_idx] / Scaler[f_idx]) ** 2
```

Algorithm results:

Since the Kaggle do not provide test data for this dataset I can only spilt the test file by myself.

10 times cross validation accuracy: 77%.
Corresponding k: 21

Random validation accuracy:96.5%
Randome test accuracy: 95%
Corresponding k: 15

Here we only take the cross validation result to analyze the TP, TN, FP, FN (validation dataset size: 77)

Randomly choose 1 validation example:

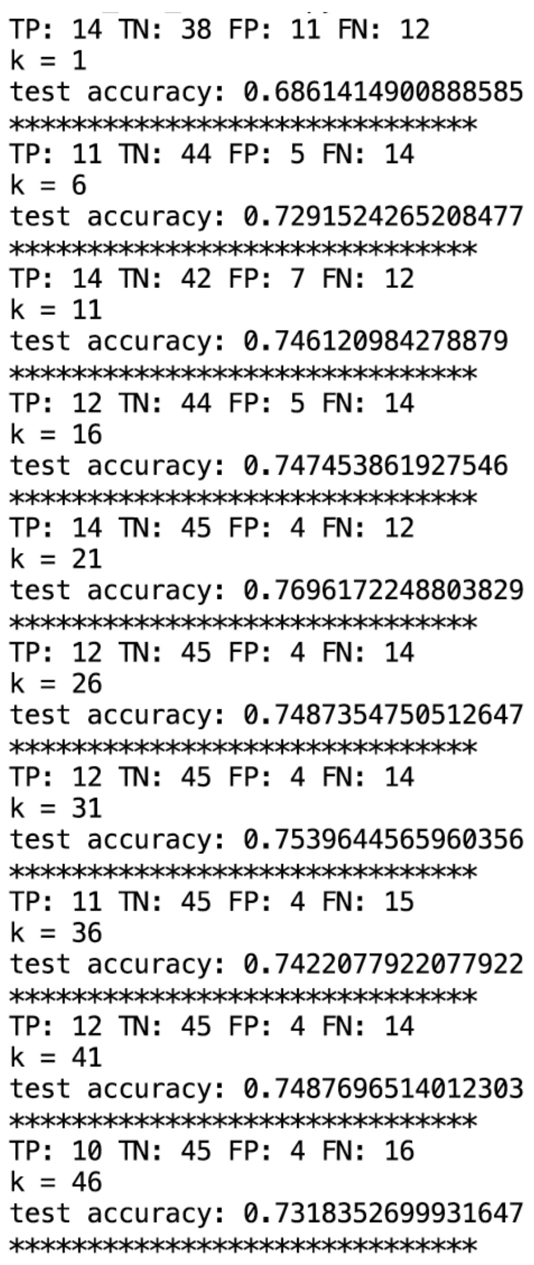
	predict 1	predict 0
Actual 1	20 (True Positive)	12 (False Negative)
Actual 0	7 (False Positive)	38 (True Negative)

Mean TP, TN, FP, FN count table (10 times cross validation average):

	predict 1	predict 0
Actual 1	14 (True Positive)	13 (False Negative)
Actual 0	5 (False Positive)	45 (True Negative)

From the mean count table we know:
Accuracy = (14 + 45) / 77 = 76.7%

Precision = 20 / (20 + 7) = 74%
Recall = 14 / (14 + 13) = 52%



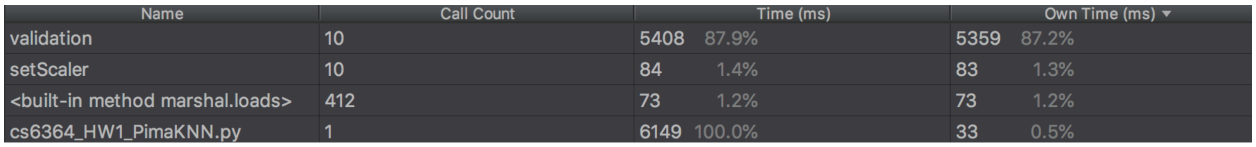
```
TP: 14 TN: 38 FP: 11 FN: 12
k = 1
test accuracy: 0.6861414900888585
******************************
TP: 11 TN: 44 FP: 5 FN: 14
k = 6
test accuracy: 0.7291524265208477
******************************
TP: 14 TN: 42 FP: 7 FN: 12
k = 11
test accuracy: 0.746120984278879
******************************
TP: 12 TN: 44 FP: 5 FN: 14
k = 16
test accuracy: 0.747453861927546
******************************
TP: 14 TN: 45 FP: 4 FN: 12
k = 21
test accuracy: 0.7696172248803829
******************************
TP: 12 TN: 45 FP: 4 FN: 14
k = 26
test accuracy: 0.7487354750512647
******************************
TP: 12 TN: 45 FP: 4 FN: 14
k = 31
test accuracy: 0.7539644565960356
******************************
TP: 11 TN: 45 FP: 4 FN: 15
k = 36
test accuracy: 0.7422077922077922
******************************
TP: 12 TN: 45 FP: 4 FN: 14
k = 41
test accuracy: 0.7487696514012303
******************************
TP: 10 TN: 45 FP: 4 FN: 16
k = 46
test accuracy: 0.7318352699931647
******************************
```

K-ACCURACY relation model: y = -0.00017 * (x-21)^2 + 0.77

Runtime:

As we know, KNN do not need to train model if we don’t use kd-tree and KMKNN and the only preprocessing time cost is scaler the data.

In this place, I take the 10 times cross validation test as an example



Name	Call Count	Time (ms)		Own Time (ms) ▾	
validation	10	5408	87.9%	5359	87.2%
setScaler	10	84	1.4%	83	1.3%
<built-in method marshal.loads>	412	73	1.2%	73	1.2%
cs6364_HW1_PimaKNN.py	1	6149	100.0%	33	0.5%

My setScalar():
Average 83 / 10 = 8.3 ms.
Including calculate the mean, median and std of the dataset. It is O(#data*#featureType)
The scaling part is also O(#data*#featureType)
In PIMA, it should be O = c * 768 * 8

My validation():
Average 5359/10 = 536 ms.
Including calculate the distance from each validation to each train data point. It is O(#data * #val * #featureType)
Heap takes O(#k * #changeMax)

Check:

O(setScalar) : O(validation)
= O(#data * #featureType) : O(#data * #val * #featureType + O(setScalar))
= c * 768 * 8 : (c * 768 * 77 * 8 + c * 768 * 8)
= c * (6144 : 479232)
= c * (8.3 : 647) is about equal the actual test proportion 8.3ms : 536ms

Part 1 Reference:
[1] Dodge, Y (2003) *The Oxford Dictionary of Statistical Terms*, OUP. ISBN 0-19-920613-9 (entry for normalization of scores)

Part2—Digit Recognizer MNIST

Dataset details:

The original datasets "train.csv" and "test.csv" are from https://www.kaggle.com/c/digit-recognizer/data

1. X datas analyze

MNIST is a dataset with 42000 and 28000 28*28 digit pictures in train.csv and test.csv respectively. the Y label distribution of this dataset is balance and I do not want to apply entropy in this largeScale feature dataset. Thus, I will only consider the correlation in X data set in this task.

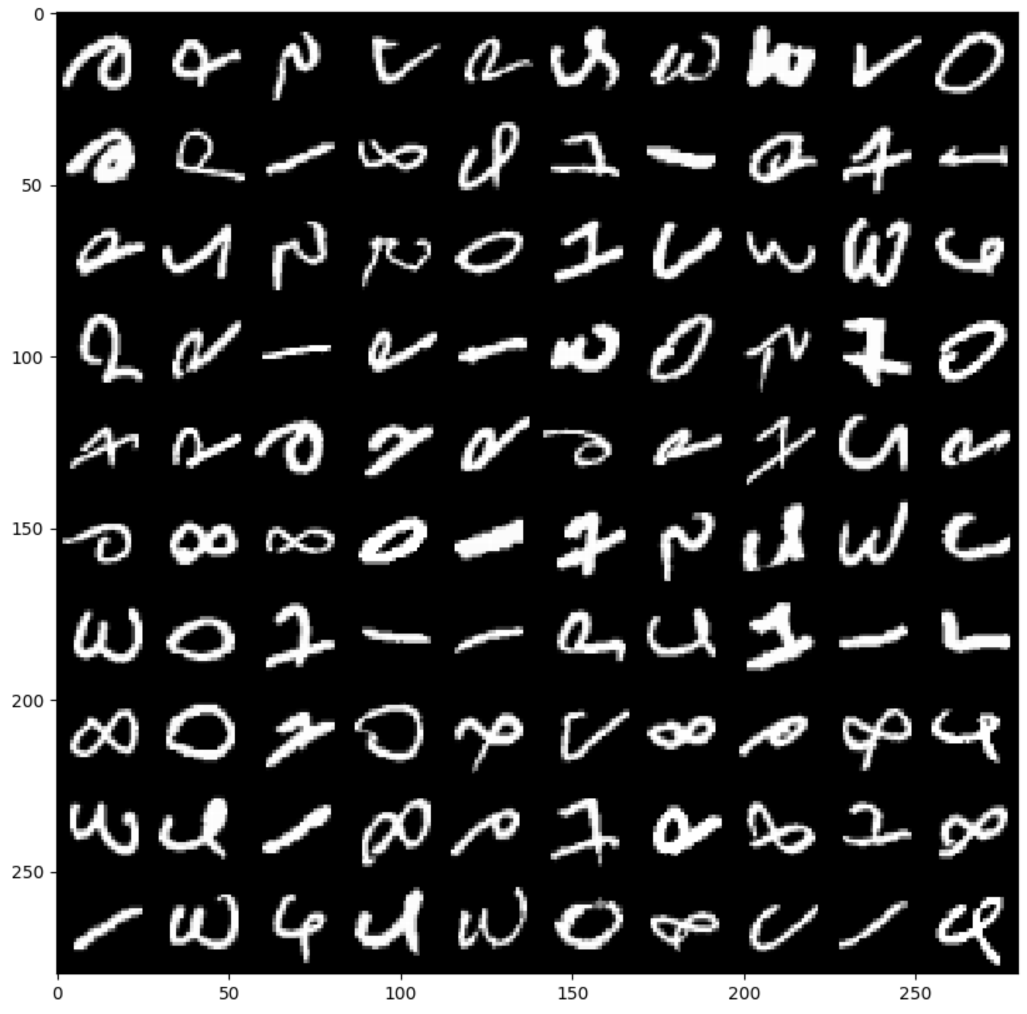
In this task, the first thing I want to find all the "nan" points which means that some pixel positions in all train.csv examples are 0. In other word, it is worthless and need to be abandoned. However, considering that the dot product is tolerable to get the square distance.

The spilt of both train and validation dataset to abandon the “nan” points will be more time consuming. Nevertheless, when we apply the kd-tree model, this “nan” abandon treatment can still improve the speed.

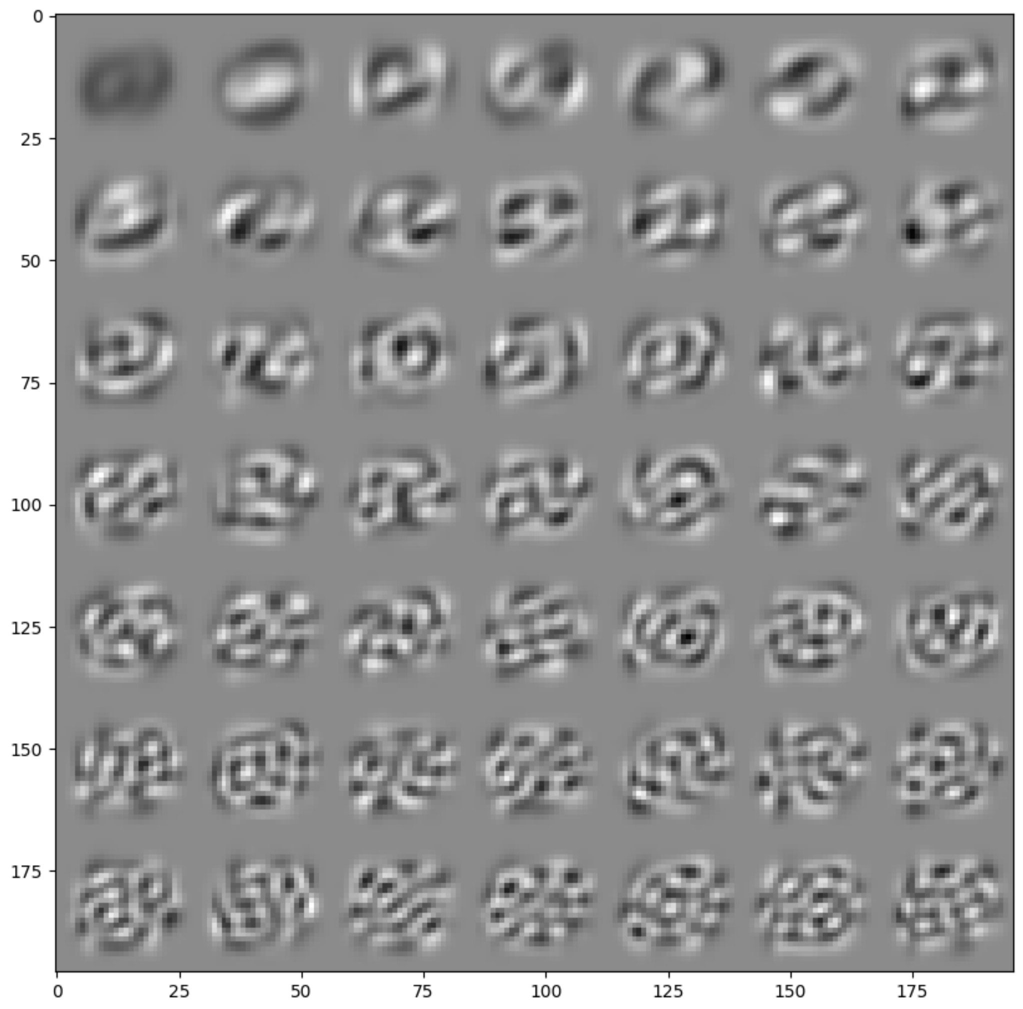
Secondly, it is important to know the feature space, feature scale and feature size. The feature size is 786 (28*28), however all features are in a same feature space (unit: gray degree) and therefore we do not need to do any feature scaling for different feature.

Thirdly, Although all features are in the same feature space, the importance of each pixel is still different and there are some correlation between some features. Thus, I want to use PCA to construct a new features coordinate and use it to reconstruct our datasets X part: X_train and X_test.

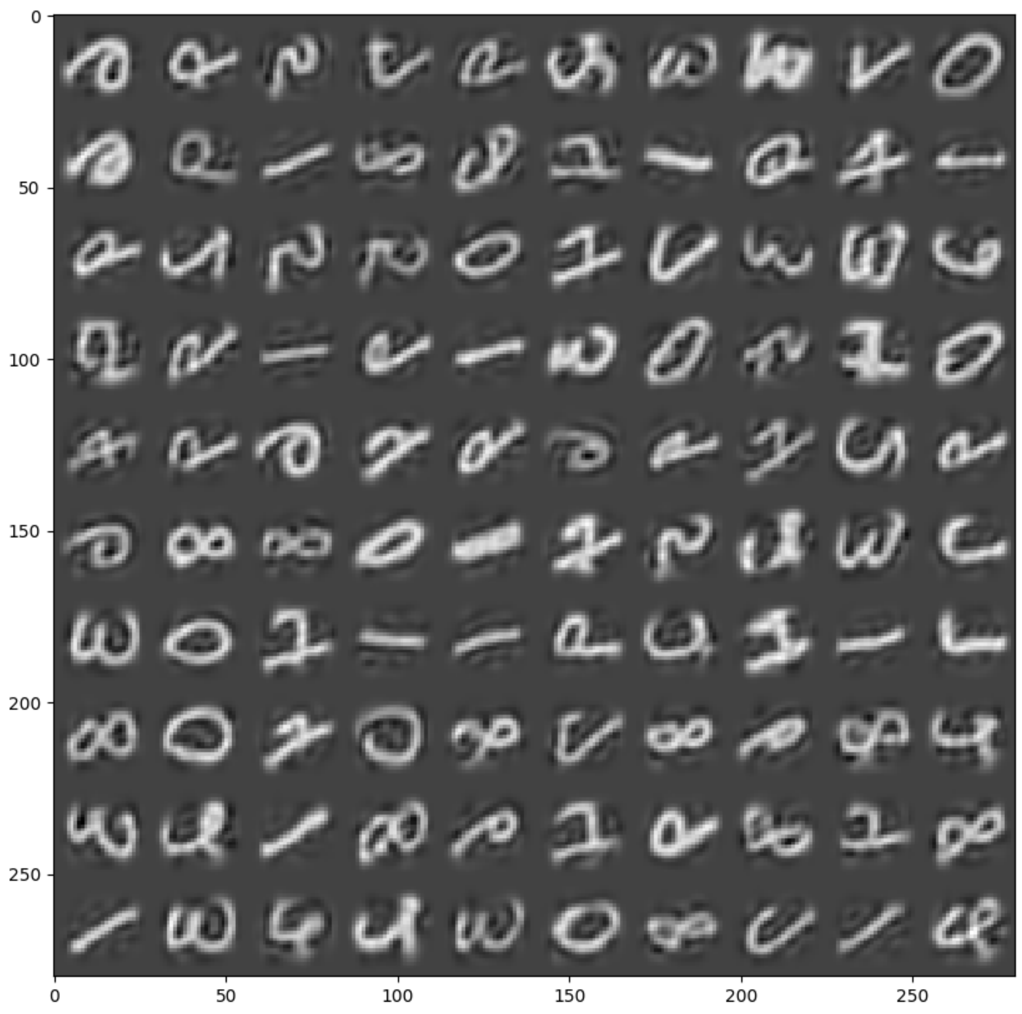
2. X datas visualization
First 100 digit in train.csv (100 * 784)



3. PCA eigen digit visualization and digit’s reconstruction
First 49 eigenvectors (eigendigit) in PCA model (784 for each)



First 100 digits recover from (100*50 compress data and top 50 784 eigenvectors)



4. Train, validation and test split

Same as Pima, please jump to page 6

Algorithm Description:

Preprocessing1:

Kd-tree is introducing the page7. Although It is easy and fast to use kd-tree to find the most close data in dataset to the validation data. When the problem comes to K-most nearest finding, it is hard to backtrack in such a broad features size.

I mainly use the heap, backtrack and range restriction to find the K-most nearest point. However, it is very time consuming to construct such a large scale dataset (which means you need to spilt the dataset by a appropriate feature and a appropriate position) or the validation step of backtrack finding will be a disaster.

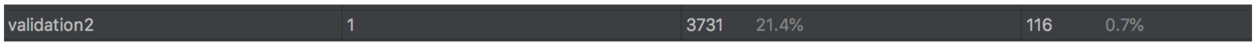
There is also a algorithm which name is KMKNN which construct a k-means dataset and use the arcs of each centroids to find the validation KNN. I have not finish this algorithm yet so there will be no detail about it.

Preprocessing2:

-PCA algorithm:
The PCA method use the will allow us to find the most n important element in the eigenvectors se. In this model, I both try n = 20, n = 50 and n = 100 and finally choose n = 50 one.
It make the validation() method run much faster than previous. The time complexity is jump from #data * #val * 784 to #data * #val * 50.

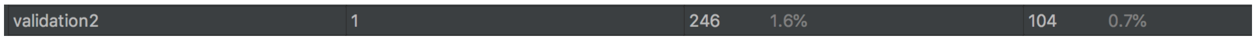
-PCA vs OriginalData:

10 digits predict without PCA: 373ms/prediction (*784 pixels)



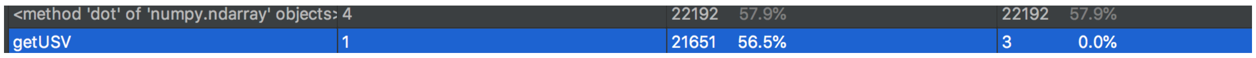
validation2	1	3731	21.4%	116	0.7%

10 digits predict with PCA: 24.6ms/prediction (*50)



validation2	1	246	1.6%	104	0.7%

USV get U and reconstruct X_train, X_test time cost: 21497ms (acceptable when validation set size is larger 65)



<method 'dot' of 'numpy.ndarray' objects>	4	22192	57.9%	22192	57.9%
getUSV	1	21651	56.5%	3	0.0%

-SVD part:
Firstly, I construct the covariance matrix by

```
cov_matrix = X.T.dot(X) / X.shape[0]
```

Then, I directly use the scipy.linalg package to get the U S V

```
U, S, V = scipy.linalg.svd(cov_matrix, full_matrices=True, 
compute_uv=True)
```

The detail of svd decomposition is below:

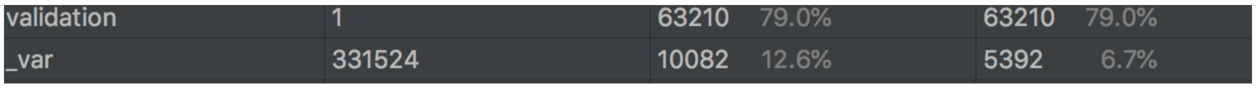
1. Compute A.transpose and np.dot(A.transpose,A).
2. Determine the eigenvalues of np.dot(A.transpose,A) in order and square root it to get singular values.
3. Replace these singular values with the diagonal line a a new empty S matrix and compute its inverse.
4. Use eighteen values to compute Eigen vector of np.dot(A.transpose,A) and use a new empty A matrix to hold it and compute its transpose.
5. U = A dot V dot S-1
6. Return U, S, V.T

-Data reset part:
Find a appropriate level of eigenvectors in U to do dot product with X_train and X_test. The new matrix we get is the project of original data to the new feature coordinate. The shape of U is 784*784 and the new X_train and X_test after dot product will become #data * #PC

Validation part:

In this part I mainly want to mention the broadcast dot product and the kd-tree backtracking.

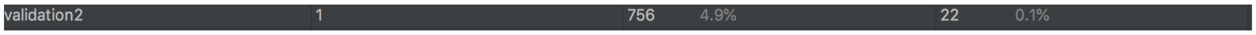
-broadcast vs for-loop (drop PCA)
For loop takes:



validation	1	63210 79.0%	63210 79.0%
_var	331524	10082 12.6%	5392 6.7%

63210ms for 2 digit validation.
Which is 31605ms for one digit prediction.

Broadcast takes:



validation2	1	756 4.9%	22 0.1%

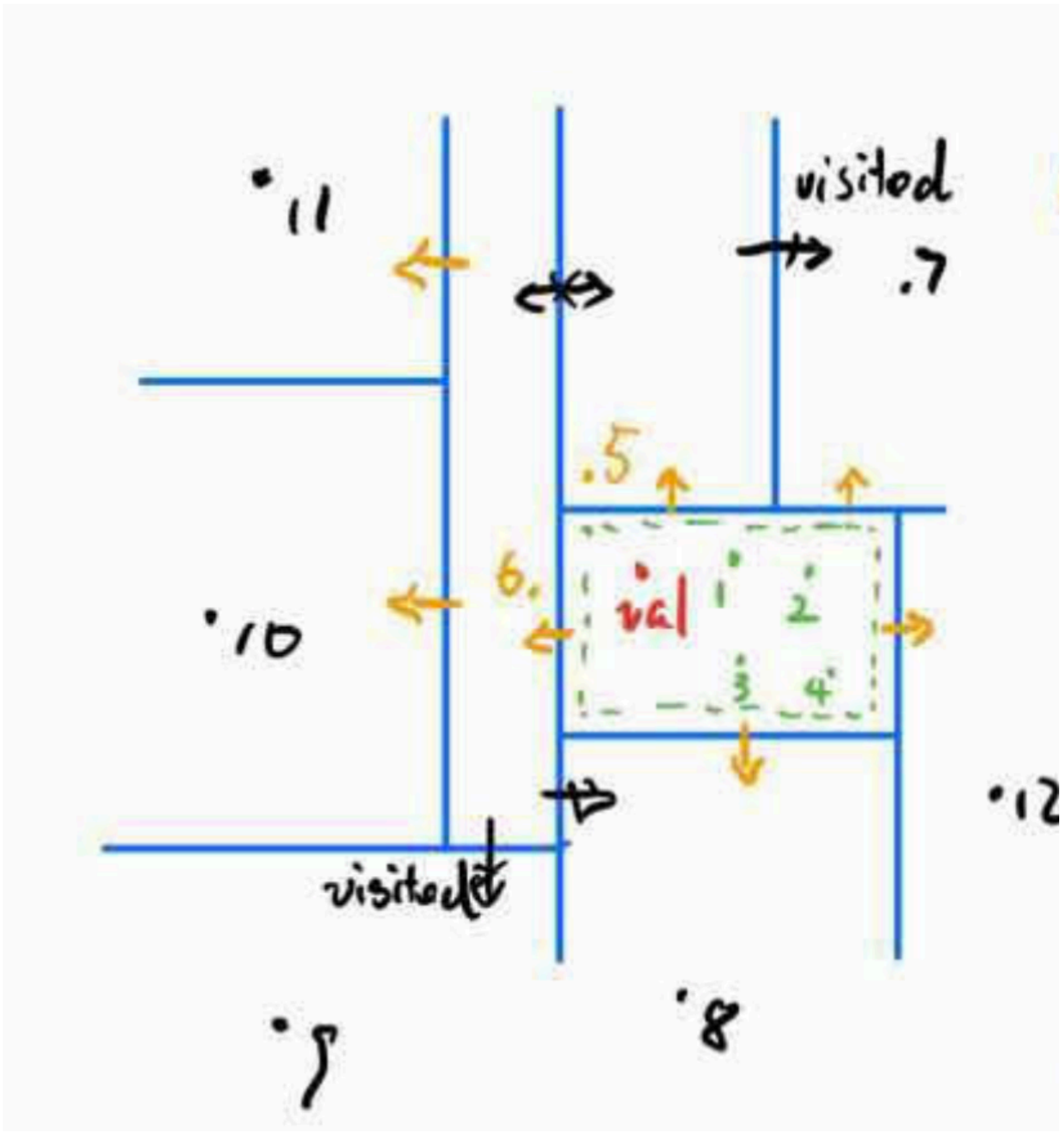
378ms for one digit prediction

Speed up 83.6 times!

Broadcast method:

```
def distance(p1, p2):
    # p2 = p2.T[useful].T
    return np.sum(np.square(p1 - p2), axis=-1)
```

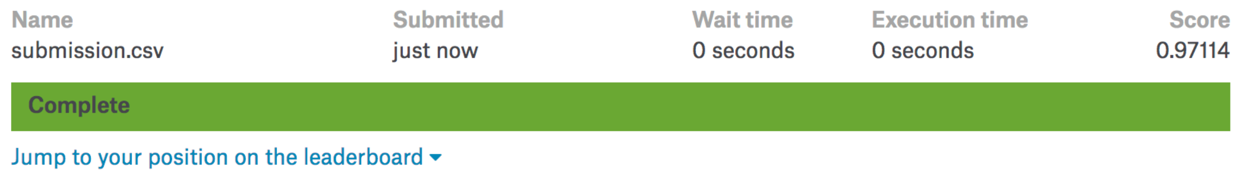
KD-tree backtracking part:



Step 1: Reduce the search area by kd-tree search
Step 2: At the process of reduction, when subTree element < k, find the point current area (green box in this figure). Add the k-most nearest point into a heap In this area.
Step 3: find the neighbor of this box and add the point smaller than heap.max into the heap. Do it recursively and stop when all points in neighbor is larger than heap max
Step 4: return heap which is our k-nn result and vote.

However, we should know that the kd-tree algorithm will only reduce the time complexity and has no effect on the prediction result.

Algorithm results:



Name	Submitted	Wait time	Execution time	Score
submission.csv	just now	0 seconds	0 seconds	0.97114

Complete

Jump to your position on the leaderboard

97.114%, this is my final Kaggle submission accuracy with 28000 test set. However I only take 20 minutes to adjust the #principle vectors in PCA and the KNN's K value and I believe that the result will be better if I choose a more general K and #principle vectors.

Confusion matrix
p.s. Since I do not have the 28000 test Y dataset, I will only use my split validation set to construct the confusion matrix.

The vertical axis is the actual result and the horizontal axis is the predict result

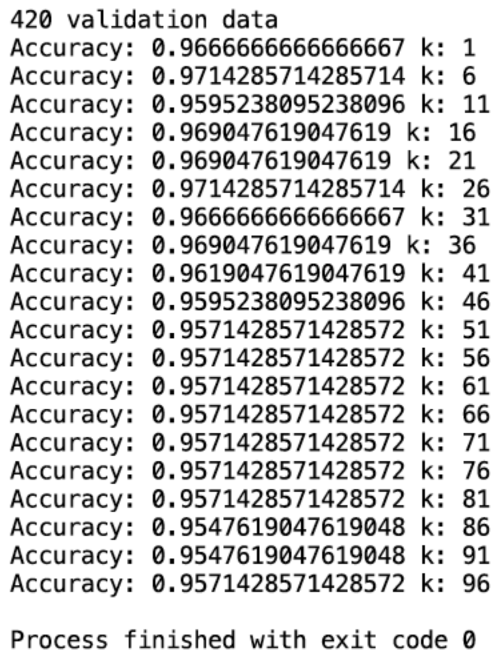
confusionMatrix

	0	1	2	3	4	5	6	7	8	9
0	409	0	2	0	0	0	0	0	0	0
1	0	482	0	1	0	0	0	2	0	0
2	2	2	391	2	0	0	1	3	1	1
3	1	3	2	401	0	3	0	0	5	3
4	0	3	0	0	443	0	4	1	0	10
5	0	0	0	4	1	360	3	0	2	2
6	3	1	0	0	0	2	407	0	0	0
7	0	6	2	0	1	0	0	434	0	3
8	0	2	3	3	1	5	2	0	359	7
9	2	1	0	3	3	1	0	9	0	390

There are two value will influent final predict accuracy: #principle vectors, k

So, I said #principle list [20,50,100], k value list [1,6,11,…,96] to generate the following result

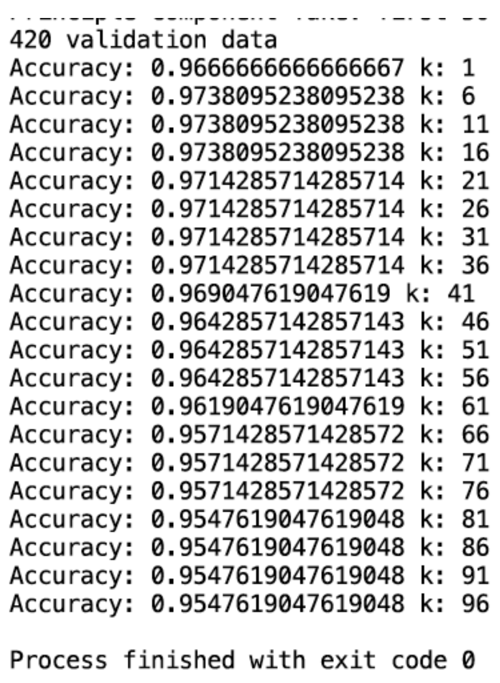
#principle vectors = 20:



```
420 validation data
Accuracy: 0.9666666666666667 k: 1
Accuracy: 0.9714285714285714 k: 6
Accuracy: 0.9595238095238096 k: 11
Accuracy: 0.969047619047619 k: 16
Accuracy: 0.969047619047619 k: 21
Accuracy: 0.9714285714285714 k: 26
Accuracy: 0.9666666666666667 k: 31
Accuracy: 0.969047619047619 k: 36
Accuracy: 0.9619047619047619 k: 41
Accuracy: 0.9595238095238096 k: 46
Accuracy: 0.9571428571428572 k: 51
Accuracy: 0.9571428571428572 k: 56
Accuracy: 0.9571428571428572 k: 61
Accuracy: 0.9571428571428572 k: 66
Accuracy: 0.9571428571428572 k: 71
Accuracy: 0.9571428571428572 k: 76
Accuracy: 0.9571428571428572 k: 81
Accuracy: 0.9547619047619048 k: 86
Accuracy: 0.9547619047619048 k: 91
Accuracy: 0.9571428571428572 k: 96

Process finished with exit code 0
```

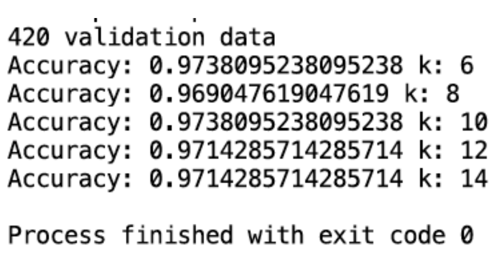
# principle vectors = 50



```
420 validation data
Accuracy: 0.9666666666666667 k: 1
Accuracy: 0.9738095238095238 k: 6
Accuracy: 0.9738095238095238 k: 11
Accuracy: 0.9738095238095238 k: 16
Accuracy: 0.9714285714285714 k: 21
Accuracy: 0.9714285714285714 k: 26
Accuracy: 0.9714285714285714 k: 31
Accuracy: 0.9714285714285714 k: 36
Accuracy: 0.969047619047619 k: 41
Accuracy: 0.9642857142857143 k: 46
Accuracy: 0.9642857142857143 k: 51
Accuracy: 0.9642857142857143 k: 56
Accuracy: 0.9619047619047619 k: 61
Accuracy: 0.9571428571428572 k: 66
Accuracy: 0.9571428571428572 k: 71
Accuracy: 0.9571428571428572 k: 76
Accuracy: 0.9547619047619048 k: 81
Accuracy: 0.9547619047619048 k: 86
Accuracy: 0.9547619047619048 k: 91
Accuracy: 0.9547619047619048 k: 96

Process finished with exit code 0
```

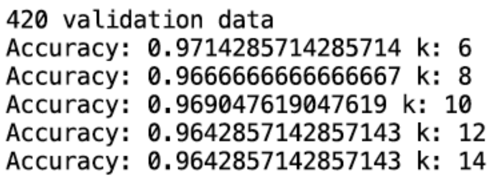
Apply more close gap in the higher accuracy range



```
420 validation data
Accuracy: 0.9738095238095238 k: 6
Accuracy: 0.969047619047619 k: 8
Accuracy: 0.9738095238095238 k: 10
Accuracy: 0.9714285714285714 k: 12
Accuracy: 0.9714285714285714 k: 14

Process finished with exit code 0
```

Apply another seed (random.seed = 2) to avoid overfitting



```
420 validation data
Accuracy: 0.9714285714285714 k: 6
Accuracy: 0.9666666666666667 k: 8
Accuracy: 0.969047619047619 k: 10
Accuracy: 0.9642857142857143 k: 12
Accuracy: 0.9642857142857143 k: 14
```

# principle vectors 100

```
420 validation data
Accuracy: 0.9666666666666667 k: 1
Accuracy: 0.9738095238095238 k: 6
Accuracy: 0.9761904761904762 k: 11
Accuracy: 0.9738095238095238 k: 16
Accuracy: 0.9738095238095238 k: 21
Accuracy: 0.9714285714285714 k: 26
Accuracy: 0.969047619047619 k: 31
Accuracy: 0.9666666666666667 k: 36
Accuracy: 0.9642857142857143 k: 41
Accuracy: 0.9642857142857143 k: 46
Accuracy: 0.9619047619047619 k: 51
Accuracy: 0.9571428571428572 k: 56
Accuracy: 0.9571428571428572 k: 61
Accuracy: 0.9547619047619048 k: 66
Accuracy: 0.9523809523809523 k: 71
Accuracy: 0.9547619047619048 k: 76
Accuracy: 0.9547619047619048 k: 81
Accuracy: 0.95 k: 86
Accuracy: 0.9523809523809523 k: 91
Accuracy: 0.95 k: 96
```

Finally, I get the approximate most better #principle vectors and k which are 50 and 10 by these analyze.

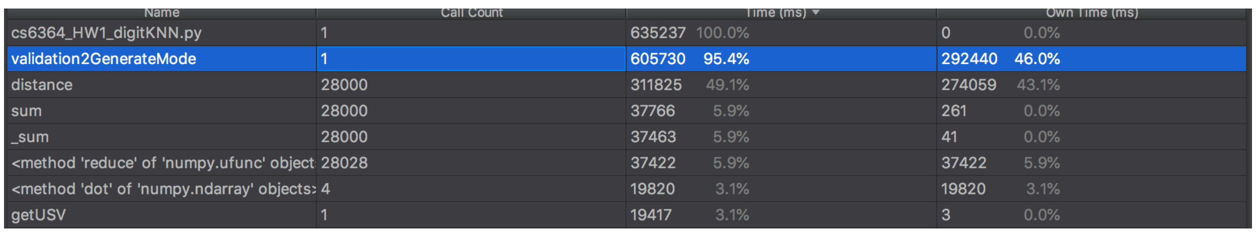
And the relationship between k and accuracy should be

Accuracy = if k < 6: 0.96 + c*k
if 6 <= k < 25: 0.97
If k >= 25: 0.97 - c*k

Runtime:

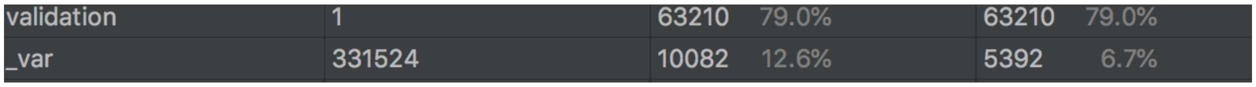
The 24.6ms/prediction is my final validation speed

For the whole 28000 test data set and preprocessing add up,
It takes 635237 ms which is about 10 minutes.
For the validation parts, it takes 605730 ms and 21.6ms for each digit validation
For the USV to PCA parts, it takes 19417ms

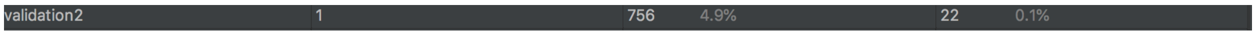


Name	Call Count	Time (ms) ▾	Own Time (ms)
cs6364_HW1_digitKNN.py	1	635237 100.0%	0 0.0%
validation2GenerateMode	1	605730 95.4%	292440 46.0%
distance	28000	311825 49.1%	274059 43.1%
sum	28000	37766 5.9%	261 0.0%
_sum	28000	37463 5.9%	41 0.0%
<method 'reduce' of 'numpy.ufunc' object	28028	37422 5.9%	37422 5.9%
<method 'dot' of 'numpy.ndarray' objects>	4	19820 3.1%	19820 3.1%
getUSV	1	19417 3.1%	3 0.0%

With **dot product**, the time / one prediction improve from 31605ms to 378ms
Time complexity for each prediction: O(#train_data * #feature * 1validation)

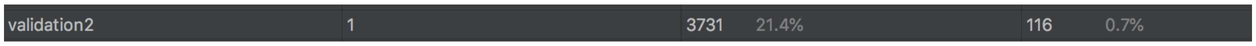


validation	1	63210	79.0%	63210	79.0%
_var	331524	10082	12.6%	5392	6.7%

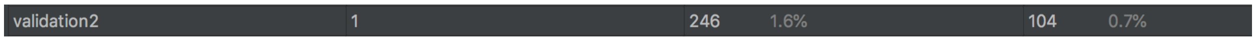


validation2	1	756	4.9%	22	0.1%

With **PCA**, the time / one prediction improve from 378ms to **24.6ms**
Time complexity for each prediction: O(#train_data * 784 feature * 1validation) -> O(#train_data * 50 feature * 1validation)



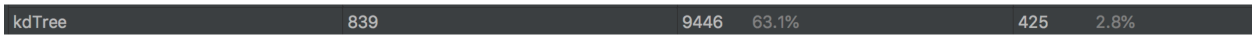
validation2	1	3731	21.4%	116	0.7%



validation2	1	246	1.6%	104	0.7%

The **construction of PCA** takes 19417ms
Time complexity of PCA construction: O(#train_data ^ 2 * #feature + # feature ^ 3 + #feature ^ 2 * #train_data)
Actually the time complexity of PCA construction and test data prediction should be in same level. However, they are 605730ms and 19417ms respectively in running time. It is believed that the build in scipy.linalg.svd is optimize and the square distance computing is time consuming while the frequently heap push and heap pop may also influent the time complexity. With PCA, the time/prediction drop from 378ms to 24.6ms which save 353ms and it means that if we need more than 55 times of validation, PCA is a good choice.

The **construction of KD-tree** takes 9446ms (not applied in final mode)



kdTree	839	9446	63.1%	425	2.8%

Expected KD-tree search time complexity: O(treeHeight), treeHeight = O(lg#data) in ideality.