Handwritten Digits Recognition by Support Vector Machine

Yangxintong Lyu
Vrije Universiteit Brussel
yangxintong.lyu@vub.be

Abstract

Handwritten digits recognition is a very interesting multi-classification problem. And Support Vector Machine is a good binary classifier. I implement a Support Vector Machine using hard margin and soft margin in this report. The recalls are in range of $[0.5, 1.0]$ *and the precisions are in range of* $[0.33, 1.0]$ *by different digits*

I. Introduction

Support Vector Machine is a binary classifier based on solving a convex quadratic programming problem. My task is to recognize the handwritten digits using THE MNIST DATABASE of handwritten digits. All the data are parsed to gray images in size of 28×28 pixels. There are totally 70000 data, including 60000 training data and 10000 test data. These data are obtained from 500 different writers. Due to the time-consuming factor, I only use a part of data to test my implementation.

II. Methods

[3] The main strategy of Support Vector Machine is to find a hyperplane which can maximum the margin between support vectors and the plane. In order to recognize the handwritten digits correctly, I mainly implement two methods – Hard Margin Maximization linear SVM and Soft Margin Maximization linear SVM.

- Hard Margin Maximization
  To maximum the distance between feature points and hyperplane, we can transfer this problem to the following:

$$max_{\omega,b} \quad \frac{2}{||\omega||}$$

$$s.t. \quad y_i(\omega^T x_i + b) \geq 1, \quad i = 1, 2, ..., N.$$

  where ω is the coefficient matrix of the hyperplane, b is the intercept, x_i is the ith feature vector and y_i is the ith class label which are $\{1, -1\}$. N is the number of training data. Because $max\frac{2}{||\omega||}$ is equivalent to $min\frac{1}{2}||\omega||^2$. So the final programming problem is:

$$min_{\omega,b} \quad \frac{1}{2}||\omega||^2$$

$$s.t. \quad y_i(\omega^T x_i + b) \geq 1, \quad i = 1, 2, ..., N.$$

- Soft Margin Maximization
  The Soft Margin Maximization method is on condition that the data are linearly separable, while it is a fact that we can not guarantee all the data are linearly separable. Thus, we need to handle these outliers without which the data are linearly separable. Here, we introduce a relax-variable ξ for outlier. For the target function, we

add a regularization term. So, the final programming problem is:

$$min_{\omega,b,\xi} \quad \frac{1}{2}||\omega||^2 + C\sum_{i=1}^{N}\xi_i$$

$$s.t. \quad y_i(\omega^T x_i + b) \geq 1 - \xi_i, \quad i = 1, 2, ..., N.$$
$$\xi_i \geq 0, \quad i = 1, 2, ..., N.$$

III. Implementations

[1,2,3] My implementation includes these three main steps. And in each part, I describe more details.

i. Data Parser

The format of data obtained from the MNIST database is not the normal image format, such as '.png', '.jpg', etc. All the data are stored in a kind of binary file. So, firstly, I process all the data from binary to integer. Here, I use a python package 'struct' to help me parse the binary files. Every image is represented by a 28×28-dimension numpy array and each pixel is represented by one gray values ranging from 0 to 255. Every image has a label which is for digit representation from 0 to 9.

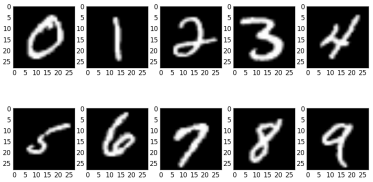

Figure 1: *example of digits 0 - 9*

ii. Support Vector Machine Imlementation

I have implemented Linear Hard Margin Maximization SVM and Linear Soft Margin Maximization SVM. I use a SVM class to encapsulate these two method, and for solving the convex quadratic programming, I use a python package 'cvxopt'. Additionally, the data structures of 'cvxopt', such as matrix, diagonal matrix and sparse matrix, are used for data storage. According to the documentation of 'cvxopt', the convex quadratic programming problem is represented by the following form:

$$min_{\omega,b,\xi} \frac{1}{2}\times$$

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_m \\ b \\ \xi_1 \\ \xi_2 \\ \vdots \\ \xi_N \end{bmatrix}^T \times \begin{bmatrix} I & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \times \begin{bmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_m \\ b \\ \xi_1 \\ \xi_2 \\ \vdots \\ \xi_N \end{bmatrix}$$

$$+ \quad C \times \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 1 & 1 & \cdots & 1 \end{bmatrix} \times \begin{bmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_m \\ b \\ \xi_1 \\ \xi_2 \\ \vdots \\ \xi_N \end{bmatrix}$$

$$s.t. \quad -y_i \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_m \\ b \end{bmatrix}^T \cdot \begin{bmatrix} x_1^{(i)} \\ \vdots \\ x_m^{(i)} \\ 1 \end{bmatrix} - \xi_i \leq -1, \quad i = 1, 2, ..., N.$$

$$\text{Let } P = \begin{bmatrix} I & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix},$$

$$q = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 1 & 1 & \cdots & 1 \end{bmatrix}.$$

Here, N is the number of training data, m is the number of features. I is a m-rank identity matrix. P is a $(m+n+1) \times (m+n+1)$ matrix, and there are $(m+1)$ 0s and (n) 1s in q.
In order to show the correctness of my implementation, I use some simple data to visualize my svm and 'scikit-learn' linear svm. The data are from the first 100 iris dataset, which includes 2 different classes. And the penalty term $C = 1$.

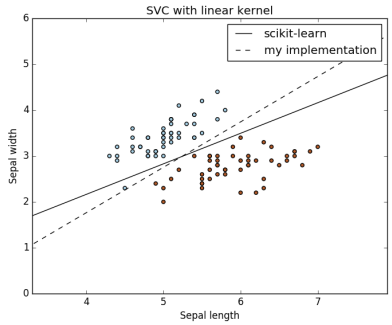

Figure 2: *The hyperplane by my svm and 'scikit-learn' svm*

SVM is a binary classifier, but recognization of digits needs multi-classifier. Thus, I handle this problem by one-vs-the-rest scheme.

iii. Cross Validation

I use K-Fold Cross Validation to evaluate the algorithm. And let $K = 10$ in my experiment.

IV. Results

[4] The environment of experiment is MacOS(10.12.4), 2.7 GHz Intel Core i5 and 8 GB 1867 MHz DDR3. In the experiment, I use two strategies for experiment. The first one is use different values of C. The second one is 10-fold cross validation. Additionally, I use Recall, Precision and F-measure to evaluate the results.

i. The first strategy

As for the first strategy, I choose the first 1500 data from the training set as training data, and the first 100 data from the test set as test data. The reason that I do not choose all the data is training a model by all the data need so long period, because the time complexity be proportional to the number of training data. And the information of training data and test data are shown below.

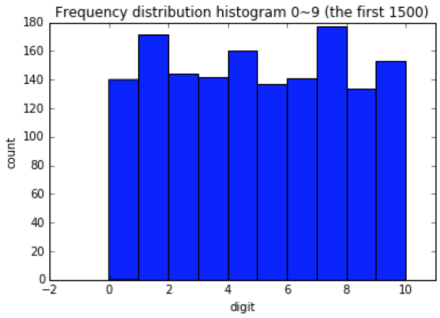

Figure 3: *Frequency Distribution Histogram of training data*

As Figure 3. shows, there is little difference between the number of different digits.

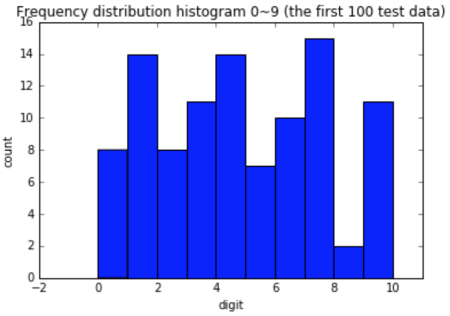

Figure 4: *Frequency Distribution Histogram of test data*

In Figure 4., the number of different digits are in-balanced in test dataset.

Table 1: *Recall by scikit-learn*

C	0	1	2	3
0	1.000	0.642	0.500	0.909
1	0.875	0.785	0.750	0.818
2	0.750	0.785	0.750	0.818

C	4	5	6	7
0	0.785	0.0	0.600	1.000
1	0.928	0.571	0.800	0.933
2	0.928	0.571	0.800	0.933

C	8	9	overall
0	0.500	0.363	0.68
1	1.0	0.818	0.83
2	1.0	0.818	0.82

The multiple classification strategy of 'scikit-learn' is one-vs-the-rest. When $C = 0$, that means Hard Margin Maximization. As the Table 1. shows, when $C = 0$, all the recalls are pretty low except digit 1, because Hard Margin Maximization is to find a hyperplane which can perfectly seperate training data by classes without any error. However, if there is no error, the hyperplane obtained by training data is too perfect, which leads overfitting. And it is not a good hyperplane for almost test data.
Then, let $C = 1, 2$. As the result shows, the recalls are similar when C is different value. But almost all recalls are improved. Because penalty term permits the existence of outliers. And the hyperplane is not perfect fitting to the training data.

Table 2: *Recall by my implementation*

C	0	1	2	3
0	1.0	0.857	0.75	0.909
1	1.0	0.928	0.625	0.818
2	1.0	0.928	0.625	0.818

C	4	5	6	7
0	1.0	0.714	0.8	0.866
1	1.0	0.714	0.7	0.8
2	1.0	0.714	0.7	0.8

C	8	9	overall
0	1.0	0.727	0.86
1	0.5	0.636	0.81
2	0.5	0.636	0.81

In my implementation, for digit $2, 3, 6, 7, 8, 9$, the recalls are higher without penalty term than with penalty term. This phenomenon is very different from 'scikit-learn' package. Because in my implementation, 'cvxopt' provides the method for quadratic programming can only get the approximate solution and the way to get the solution is different from 'scikit-learn', which lead the difference. And the recalls of prediction in digit $0, 1, 4, 5$ are higher in my implementation than 'scikit-learn.

Table 3: *Precision by scikit-learn*

C	0	1	2	3	4
0	0.4	1.0	0.8	0.714	0.916
1	1.0	1.0	0.857	0.9	0.812
2	1.0	1.0	0.857	0.9	0.812

C	5	6	7	8	9
0	0.0	1.0	0.625	0.2	0.8
1	0.5	1.0	0.823	0.4	0.818
2	0.5	1.0	0.823	0.333	0.818

Table 4: *Precision by my implementation*

C	0	1	2	3	4
0	0.888	1.0	0.666	0.769	0.823
1	0.888	1.0	0.714	0.818	0.875
2	0.888	1.0	0.714	0.818	0.875

C	5	6	7	8	9
0	0.625	0.888	0.812	0.333	0.727
1	0.625	0.875	0.857	0.5	0.636
2	0.625	0.875	0.857	0.5	0.636

According to Table 3. and Table 4., with the increasing of C, the precisions are also increasing except digit 9 in my implementation. Additionally, no other digits are predicted to digit 1 in both 'scikit-learn' and mine. And there exists a large proportion of other digits predicted to digit 8. Look back to the frequency distribution histogram of test data. The number of digit 8 is only 2, which is too small.
Here for more details, I show the confusion matrix below obtained by 'scikit-learn' linearn svm and my svm with $C = 0$.

Table 5: *confusion matrix by scikit-learn ($C = 0$),Row 1 is the predicted digit,and Column 1 is the actual digit*

digit	0	1	2	3	4	5	6	7	8	9
0	8	0	0	0	0	0	0	0	0	0
1	0	9	0	2	0	0	0	0	3	0
2	2	0	4	1	0	0	0	1	0	0
3	0	0	1	10	0	0	0	0	0	0
4	1	0	0	0	11	0	0	1	0	1
5	5	0	0	1	0	0	0	1	0	0
6	3	0	0	0	1	0	6	0	0	0
7	0	0	0	0	0	0	0	15	0	0
8	1	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	6	1	4

Table 6: *confusion matrix by my svm ($C = 0$),Row 1 is the predicted digit,and Column 1 is the actual digit*

digit	0	1	2	3	4	5	6	7	8	9
0	8	0	0	0	0	0	0	0	1	0
1	0	12	0	2	0	0	0	0	0	0
2	0	0	6	0	1	1	0	0	0	0
3	0	0	1	10	0	0	0	0	0	1
4	0	0	1	1	14	0	1	0	1	2
5	1	0	0	0	1	5	0	1	2	0
6	0	0	0	0	1	1	8	0	0	0
7	0	0	1	0	0	1	0	13	0	0
8	0	0	0	0	0	0	0	0	2	0
9	0	0	0	0	0	0	0	2	0	8

ii. The second strategy

In this part, I use 10-fold cross validation to test my implementation. And the experiment results are shown in the following tables and diagrams.

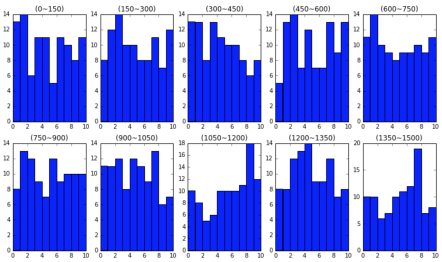

Figure 5: *frequency distribution of digits in K-fold*

Table 7: *Recall by my implementation*

C	0	1	2	3	4
1	0.901	0.929	0.751	0.741	0.821

C	5	6	7	8	9
1	0.659	0.899	0.889	0.645	0.703

Table 8: *Precision by my implementation*

C	0	1	2	3	4
1	0.930	0.927	0.787	0.747	0.819

C	5	6	7	8	9
1	0.666	0.914	0.858	0.616	0.678

Table 9: *F-measure by my implementation*

C	0	1	2	3	4
1	0.915	0.928	0.768	0.744	0.820

C	5	6	7	8	9
1	0.663	0.907	0.874	0.630	0.690

According to Table 5., Table 6. and Table 7., it is clear that the predictions of digit 0,1,4,6,7 are pretty good, while the predictions of digit 5,8,9 are bad. Comparing with the results in *i*, the relative trend of the precisions and recalls are pretty different, that because the number of test data and the features of test data are very different.

V. Discussion

i. Time Complexity

The time complexity of SVM depends heavily on the number of training data when the number of training data are much larger than the number of features. I have tried to use 60000 training data, the duration of prediction is beyond 20 hours. Additionally, I have tried to reduce the number of features by some image processing algorithms provided by 'OpenCV'. However, if the number of training data is much larger than 784(the number of features), this operation can not reduce running time. What's more, the method of solving quadratic programming also affect the running time, because the different maximum number of iterations.

ii. Conclusion

No matter which implementations('scikit-learn' or mine), the results very different by different test data. And the durations of different SVMs(trained by different size of training data) are pretty different. But there does not exist a good way to reduce the time complexity. My implementation can only work efficiently on small data set, while if the data set is too large, my SVM needs very long time to train model. In order to shorten the running time, I think the way to solve the quadratic programming is a very good point. The last and the very important point, all the training data I use are assumed linearly separable. However, this case is not always true. For further optimization of my implementation, kernel function is a good choice, which can map a low-dimension data set to a high-dimension data set. And increasement of dimension exactly can help the data to be linearly separable in high-dimensional space.

References

[1]Documentation of cvxopt.,http://cvxopt.org

[2]Documentation of scikit-learn.,http://scikit-learn.org/stable/modules/svm.html

[3]Support Vector Machine Tutorial.,http://www.cs.columbia.edu/~kathy/cs4701/documents/jason_svm_tutorial.pdf

[4]Precision and Recall.,https://en.wikipedia.org/wiki/Precision_and_recall

[5]Python struct documentation https://docs.python.org/3/library/struct.html

Appendices

Parse data and label information

```
def decode_idx3(idx3_file):
    bin_data = open(idx3_file, 'rb
        ').read()
    offset = 0
    fmt_header = '>iiii'
    magic_number, num_images,
        num_rows, num_cols =
        struct.unpack_from(
        fmt_header, bin_data,
        offset)
    image_size = num_rows *
        num_cols
    offset += struct.calcsize(
        fmt_header)
    fmt_image = '>' + str(
        image_size) + 'B'
```

```
    images = np.empty((num_images,
        num_rows, num_cols))
    for i in range(num_images):
        images[i] = np.array(
            struct.unpack_from(
            fmt_image, bin_data,
            offset)).reshape((
            num_rows, num_cols))
        offset += struct.calcsize(
            fmt_image)
    return images

def decode_idx1(idx1_file):
    bin_data = open(idx1_file, 'rb
        ').read()
    offset = 0
    fmt_header = '>ii'
    magic_number, num_images =
        struct.unpack_from(
        fmt_header, bin_data,
        offset)
    offset += struct.calcsize(
        fmt_header)
    fmt_image = '>B'
    labels = np.empty(num_images)
    for i in range(num_images):
        labels[i] = struct.
            unpack_from(fmt_image,
             bin_data, offset)[0]
        offset += struct.calcsize(
            fmt_image)
    return labels
```

SVM class

```
class SVM(object):

    def __init__(self, C):
        self.C = C

    def train(self, train_data,
        train_label): # default
        method---linear
        if self.C == 0:
            self.linear(train_data
                , train_label)
        else:
            self.linear_KKT(
                train_data,
                train_label)
        self.solver = solvers.qp(
            self.P, self.q, self.G
            , self.h)
        self.solution = self.
            solver['x']

    def predict(self, test_data):
        self.test_num = len(
            test_data)
        self.test_data = matrix(
            test_data).trans()
        self.prediction = self.
            test_data * self.
            solution[:self.
            feature_num]
        return self.prediction


    #  satisfy KKT condition
    #  min 1/2 * ||w||^2 + C *
        sigma(ei) #  penalty

    def linear_KKT(self,
        train_data, train_label):
        self.feature_num = len(
            train_data[0])
        self.data_num = len(
            train_label)
        temp_G = []
        temp_h = [[-1.0] * self.
            data_num + [0.0] *
            self.data_num]
        temp_diag = spdiag([1.0] *
             (self.feature_num -
            1))
        self.P = matrix([[
            temp_diag, matrix
            ([[0.0] * (self.
            data_num + 1)] * (self
            .feature_num - 1))],
                        [matrix
                            ([[0.0]
                             * (
                            self.
                            feature_num
                             +
                            self.
```

```
                                data_num
                                )] *
                                (self
                                .
                                data_num
                                 + 1)
                                )]])
        self.q = self.C * matrix
            ([0.0] * (self.
            feature_num) + [1.0] *
             self.data_num)
        self.h = matrix(temp_h)
        for index in range(len(
            train_data)):
            temp = []
            for pixel in
                train_data[index]:
                temp.append(-1.0 *
                     pixel *
                    train_label[
                    index])
            for i in range(len(
                train_data)):
                if i == index:
                    temp.append
                        (-1)
                else:
                    temp.append(0)
            temp_G.append(temp)
        A = matrix([[matrix([[0.0]
             * self.data_num] *
            self.feature_num)], [
            spdiag([-1.0] * self.
            data_num)]])

        self.G = matrix([matrix(
            temp_G).trans(), A])



#  the simplest linear model
#  min 1/2 * ||w||^2
def linear(self, train_data,
    train_label):
    self.feature_num = len(
        train_data[0])
    self.data_num = len(
        train_label)
    self.h = matrix([-1.0 for
        i in range(self.
        data_num)])
    self.G = matrix(train_data
        [:])
    # matrix(i,j) = $variable[
        i * col + j]
    for index in range(0, len(
        train_label)):
        self.G[index * self.
            feature_num: (
            index + 1) * self.
            feature_num] = -1
            * train_label[
            index] * self.G[
            index * self.
            feature_num: (
            index + 1) * self.
            feature_num]
    print(self.G)
    self.G = self.G.trans()
    temp = []
    for index in range(self.
        feature_num):
        s = []
        for i in range(self.
            feature_num):
            if i == index:
                s.append(1.0)
            else:
                s.append(0.0)
        temp.append(s[:])
    self.P = matrix(temp[:])
    self.q = matrix([0.0 for i
        in range(self.
        feature_num)])
```

Comparison between my implmentation and scikit-learn by linear SVM

```
iris = datasets.load_iris()
X = iris.data[:100, :2]
y = iris.target[:100]
label = []
x = X
for i in range(len(y)):
    if y[i] == 0:
        label.append(-1)
    else:
```

```
        label.append(1)
last_col = np.array([1] * len(X))
train_data = np.c_[x,last_col]
C = 1.0
#my implementation
my_svm = SVM(C)
prediction = my_svm.train(
    train_data,label)
# my hyperplane
my_w0 = prediction[0]
my_w1 = prediction[1]
my_b = prediction[2]

#scikit-learn linear svm
svc = svm.LinearSVC(C=1)
svc.fit(X, y)
#scikit-learn hyperplane
w = svc.coef_
b = svc.intercept_[0]

# show two hyperplanes
p_x = np.linspace(3,8)
p_y = []
my_y = []
for i in range(len(p_x)):
    temp1 = (-w[0][0]/w[0][1]) *
        p_x[i] - b/w[0][1]
    p_y.append(temp1)
    temp2 = (-my_w0/my_w1) * p_x[i
        ] - my_b/my_w1
    my_y.append(temp2)

plt.scatter(X[:, 0], X[:, 1], c=y,
    cmap=plt.cm.Paired)
plt.plot(p_x,p_y,'k-', label='
    scikit-learn')
plt.plot(p_x,my_y,'k--',label='my␣
    implementation')
plt.xlabel('Sepal␣length')
plt.ylabel('Sepal␣width')
plt.xlim(xx.min(), xx.max())
plt.title('SVC␣with␣linear␣kernel'
    )
plt.legend()
plt.show()
```

reduce the number of dimension

```
def reduce_dimension(
    original_image, new_size=15,
    method=cv2.INTER_AREA):
    return cv2.resize(
        original_image, (new_size,
         new_size), method)
```

train model scikit-learn

```
expected = test_labels[:
    test_number]
predicted = []
predicted_images = []
for index in range(len(
    sci_classifier_diff_c)):
    sci_classifier_diff_c[index].
        fit(
        train_images_reduce_format
        [:train_number],
        train_labels[:train_number
        ])
    predicted.append(
        sci_classifier_diff_c[
        index].predict(
        test_images_reduce_format
        [:test_number]))
    predicted_images.append(list(
        zip(expected, predicted
        [-1])))
```

data processing for my SVM

```
train_images_format = []
test_images_format = []
train_labels_format = [[] for i in
    range(10)]
for i in range(10):
    for index in range(
        train_number):
        if train_labels[index] ==
            i:
            train_labels_format[i
                ].append(1)
        else:
            train_labels_format[i
                ].append(-1)

for index in range(train_number):
    train_images_format.append(
        list(np.reshape(
        train_images[index],
        length_feature)))
```

```
    train_images_format[-1].append
        (1)  # for linear in-
        equation
for index in range(len(test_images
    )):
    test_images_format.append(list
        (np.reshape(test_images[
        index], length_feature)))
    test_images_format[-1].append
        (1)  # for linear in-
        equation
```

Calculation of FP,FN,TP,TN

```
FP = [[] for i in range(10)]
FN = [[] for i in range(10)]
TP = [[] for i in range(10)]
TN = [[] for i in range(10)]
for k in range(10):
    cv_labels = [[] for s in range
        (10)]
    FP[k] = [[] for s in range(10)
        ]
    TP[k] = [[] for s in range(10)
        ]
    FN[k] = [[] for s in range(10)
        ]
    TN[k] = [[] for s in range(10)
        ]
    for s in range(10):
        cv_labels[s] =
            train_labels_format[s
            ][start_index[k]:
            end_index[k]]
    for j in range(len(
        cv_prediction[k])):
        fp = 0
        fn = 0
        tp = 0
        tn = 0
        for i in range(len(
            cv_prediction[k][j])):
            if cv_prediction[k][j
                ][i] >= 0:
                if int(cv_labels[j
                    ][i]) == 1:
                    tp += 1
                else:
                    fp += 1
            else:
                if int(cv_labels[j
                    ][i]) == 1:
                    fn += 1
                else:
                    tn += 1
        FP[k][j].append(fp)
        FN[k][j].append(fn)
        TP[k][j].append(tp)
        TN[k][j].append(tn)
```