WangZhiyi, 922182514
ZhangTianyao, 922182332

Project: Phoneme

1 Importation

Firstly, we need to import the data sets.

```
import scipy.io as scio

def import_data_set():
    data_file = './data/data.mat'
    database = scio.loadmat(data_file)
#   print(database)
    train_set_x = database['trainSet'][0][0][0]
    train_set_y = database['trainSet'][0][0][1]
    test_set_x = database['testSet'][0][0][0]
    test_set_y = database['testSet'][0][0][1]
    train_set_x = train_set_x.T
    train_set_y = train_set_y.T
    test_set_x = test_set_x.T
    test_set_y = test_set_y.T
#   print(train_set_x, train_set_y, test_set_x, test_set_y)
    return train_set_x, train_set_y, test_set_x, test_set_y
```

We choose *scipy.io* to import the file – *data.mat*
After some try, we found that several nesting arrays in the file, so we use $database['dataSet'][0][0][x]$ to directly use the matrix. And also, we need to transpose the matrixs
Take *TrainSet* as an example

```
# previous matrix

[[(array([[-0.93546556, -1.21860935, -0.94174184, ..., -0.81037782,
        -0.54345126, -0.08398244],
       [ 0.36033807,  0.81163857,  0.44064021, ...,  0.84241991,
        -0.04577354,  1.02363125],
       [-1.93954278, -0.409726  ,  1.64841315, ..., -0.38010014,
         0.77302696, -1.32637319],
       ...,
       [-0.79382152, -0.3033749 , -1.65567945, ..., -0.69134904,
         0.65083445, -0.93428549],
       [-0.15949132, -0.47462633,  1.2547672 , ..., -2.09062876,
        -0.42782701,  0.35771258],
       [-1.1610882 , -0.81546287, -0.92937287, ..., -0.38855035,
        -0.90878288,  2.06137403]]), array([[1, 0, 0, ..., 1, 0, 1]], dtype=uint8))]]


# matrix after transposion

[[-0.93546556  0.36033807 -1.93954278 ... -0.79382152 -0.15949132 -1.1610882 ]
 [-1.21860935  0.81163857 -0.409726   ... -0.3033749  -0.47462633 -0.81546287]
 [-0.94174184  0.44064021  1.64841315 ... -1.65567945  1.2547672  -0.92937287]
 ... # dataset_x
 [-0.81037782  0.84241991 -0.38010014 ... -0.69134904 -2.09062876 -0.38855035]
```

```
 [-0.54345126 -0.04577354  0.77302696 ...  0.65083445 -0.42782701 -0.90878288]
 [-0.08398244  1.02363125 -1.32637319 ... -0.93428549  0.35771258  2.06137403]]

[[1]
 [0]
 [0]
 ... # dataset_y
 [1]
 [0]
 [1]]
```

2 Procession

2.1 KNN Classifier

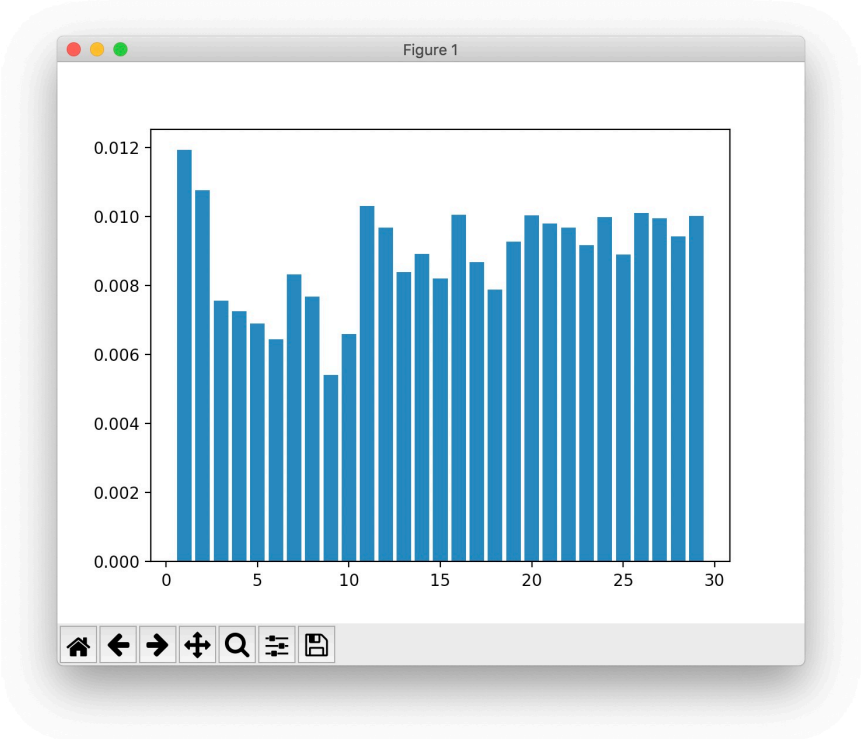
2.1.1 Model Parameters Selection

We need to select the K – how many neighbors to use.
And also, there are several methods to calculate the distances between the neighbors.
We choose three of them – *euclidean*, *manhattan* and *chebyshev*.
euclidean: the **Euclidean distance** between points **p** and **q** is the length of the line segment connecting them ($\overline{\mathbf{pq}}$). – $\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$
(https://en.wikipedia.org/wiki/Euclidean_distance)
manhattan: the **Taxicab Distance**, $d_1 d_1$, between two vectors $\mathbf{p}, \mathbf{q}$ in an n-dimensional real vector space with fixed Cartesian coordinate system, is the sum of the lengths of the projections of the line segment between the points onto the coordinate axes. – $\sum_{i=1}^{n} |x_i - y_i|$
(https://en.wikipedia.org/wiki/Taxicab_geometry)
chebyshev: the **Chebyshev Distance** between two vectors or points x and y, with standard coordinates x_i and y_i, respectively, is – $\max_i |x_i - y_i|$
(https://en.wikipedia.org/wiki/Chebyshev_distance)

Then we use **10-Fold Cross Validation** to select the best K and *distance_method*.

```
def k_selection(x, y):
    result_mean = []
    result_std = []
    method = ['euclidean', 'manhattan', 'chebyshev']
    count = 0
    mini = 0
    for k in range(1, K_MAX):
        for m in range(len(method)):
            k_model = KNeighborsClassifier(n_neighbors = k, metric = method[m])
            result = cross_val_score(k_model, x, y, cv = 10)
```

```
            result_mean.append([k, m, np.mean(result)])
            result_std.append([k, m, np.std(result)])
            if result_mean[mini][2] < result_mean[count][2]:
                mini = count
            count += 1
    res_k = result_mean[mini][0]
    res_m = method[result_mean[mini][1]]
    print(result_mean, result_std)
    result_mean = np.array(result_mean)
    result_std = np.array(result_std)
    result_mean = result_mean.T
    result_std = result_std.T
    plt.bar(result_mean[0], result_mean[1])
    plt.show()
    plt.bar(result_std[0], result_std[1])
    plt.show()
    print(result_mean, result_std)
    return res_k, res_m
```



Result: $K = 9$, *distance_method* = 'euclidean'

2.1.2 Prediction

```
def knn():
    x_train, y_train, x_test, y_test = import_data_set()
    k_neighbors, method = k_selection(x_train, y_train)
    print(k_neighbors)
    model = KNeighborsClassifier(n_neighbors = k_neighbors, metric = method)
    model.fit(x_train, y_train)
    score = model.score(x_test, y_test)
    print(score)
```

Result: *test score* = 0.8366

2.1.3 Optimization

1. The samples are **not balanced**.
An important shortcoming of this algorithm in classification is that when the sample is unbalanced, that is, the sample size of one class is large, and the number of samples of other classes is very small, it is likely that when an unknown sample is input, the sample is The majority of the K neighbors have a large sample size. However, such samples are not close to the target sample, and a small number of such samples are very close to the target sample. At this time, we have reason to think that the sample of the location belongs to a class with a small number of samples. However, KNN does not care about this problem. It only cares about which type of sample is the most, and does not take into account the distance. , we can use the weight method to improve. The neighbor weight with the small sample distance is large, and the neighbor weight with the sample distance is relatively small. Therefore, the factors of the distance are also taken into consideration, and the situation of misjudgment caused by one sample being too large is avoided.
2. The amount of **calculation** is **too large**. (Algorithm complexity is too high)
The KD tree (K-Dimension Tree) is a tree-shaped data structure that stores instance points in k-dimensional space for quick retrieval. The KD tree is a binary tree representing a division of the k-dimensional space. Constructing a KD tree is equivalent to continuously dividing the K-dimensional space with a hyperplane perpendicular to the coordinate axis to form a series of K-dimensional super-rectangular regions. Each node of the KD tree corresponds to a k-dimensional super-rectangular region. Using KD trees can save the search for most data points, thus reducing the amount of computation.
The method of constructing the KD tree is as follows: construct a root node, so that the root node corresponds to a super-rectangular region containing all instance points in the K-dimensional space; by recursively, the k-dimensional space is continuously segmented to generate a child node. Selecting an axis on the super-rectangular area and a segmentation point on the axis to determine a hyperplane that passes the selected segmentation point and perpendicular to the selected coordinate axis, and the current super-rectangular region It is divided into two sub-regions (child nodes); at this time, the instance is divided into two sub-regions, and the process is terminated until there is no instance in the sub-region (the node at the termination is a leaf node). In the process, save the instance on the corresponding

node. Usually, the selected axis of the loop is segmented into space, and the median of the training instance point on the coordinate axis is selected as the segmentation point, so that the obtained KD tree is balanced (balanced binary tree: it is an empty tree, or The absolute difference between the depths of the left sub-tree and the right sub-tree does not exceed 1, and its left sub-tree and right sub-tree are balanced binary trees).

2.2 Naive Bayes Classifier

2.2.1 Histogram

When we draw out the picture of the dataset, we can assume that it is normal **Gaussian distribution**.

2.2.2 Prediction

```
def nbc():
    x_train, y_train, x_test, y_test = import_data_set()
    nbc_g = GaussianNB()
    nbc_g.fit(x_train, y_train)
    predict = nbc_g.predict(x_test)
    count = 0
    for left, right in zip(predict, y_test):
        if left == right:
            count += 1
    print(count / len(y_test))

#   nbc_m = MultinomialNB()
#   nbc_m.fit(x_train, y_train)
#   predict = nbc_m.predict(x_test)
#   count = 0
#   for left, right in zip(predict, y_test):
#       if left == right:
#           count += 1
#   print(count / len(y_test))
#
#   nbc_b = BernoulliNB()
#   nbc_b.fit(x_train, y_train)
#   predict = nbc_b.predict(x_test)
#   count = 0
#   for left, right in zip(predict, y_test):
#       if left == right:
#           count += 1
#   print(count / len(y_test))
```

Result: $test\ score = 0.8366$

2.2.3 Optimization

1. Data analysis (pretreatment)
2. Feature selection
3. Specific classifier optimization

2.3 Perceptron Algorithm

2.3.1 Prediction

```
def pa():
    x_train, y_train, x_test, y_test = import_data_set()
    sc = StandardScaler()
    sc.fit(x_train, y_train)
    x_train_std = sc.transform(x_train)
    x_test_std = sc.transform(x_test)

    ppn = Perceptron(eta0 = 0.1, random_state = 0)
    ppn.fit(x_train_std, y_train)

    predict = ppn.predict(x_test_std)
    count = 0
    for left, right in zip(predict, y_test):
        if left == right:
            count += 1
    print(count / len(y_test))
```

Result: *test score* = 0.8102