東南大學
SOUTHEAST UNIVERSITY

SOUTHEAST UNIVERSITY

Course report

模式识别原理课程报告

Author:
王欢 (181499)

Supervisor:
杨万扣

2019 年 4 月 5 日

目录

第 1 章　Mnist 数据集

1.1　关于 Mnist 数据集

MNIST 数据集作为机器学习界的“Hello World”, 是学习模式识别和各大深度学习框架的基础数据库。2017 年，大神 Hinton 的最新研究方向 CapsNet 就是基于该数据集做的实验。该数据集是手写字符，如下图所示。数据集源自 NIST，后经 CNN 鼻祖 Lecun 等人修改后创建，共包含 60000 个训练样本和 10000 个测试样本，验证样本每张图为 28*28 的灰度图像。

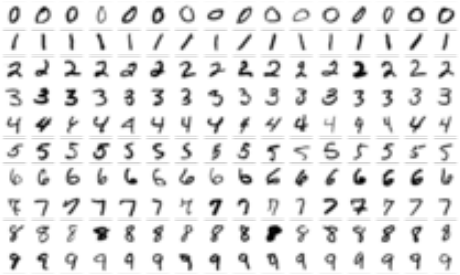

图 1.1: MnistExamples

1.2　Mnist 数据集组成

训练数据集 train 和测试数据集 test 都分为 label 和 image 两个文件。image 文件格式为 idex3-ubyte, label 文件格式为 idx1-ubyte。具体文件格式可以参见下图。像素灰度值为 0-255，0 代表白色，255 代表黑色。从图中可以看出训练集大小为 60000，测试集大小为 10000。每一幅图片大小为 28*28。

TRAINING SET LABEL FILE (train-labels-idx1-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
....			
xxxx	unsigned byte	??	label

The labels values are 0 to 9.

TRAINING SET IMAGE FILE (train-images-idx3-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
....			
xxxx	unsigned byte	??	pixel

图 1.2: 训练集 image 与 label 文件格式

TEST SET LABEL FILE (t10k-labels-idx1-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	10000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
....			
xxxx	unsigned byte	??	label

The labels values are 0 to 9.

TEST SET IMAGE FILE (t10k-images-idx3-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	10000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
....			
xxxx	unsigned byte	??	pixel

图 1.3: 测试集 image 与 label 文件格式

1.3 Python 中读取 Mnist 数据集

读取 mnist 数据集其实就是读取二进制文件。

1.3.1 idx3-ubyte 文件解码

```
def decode_idx3_ubyte(idx3_ubyte_file):
    # 读取二进制数据
    bin_data = open(idx3_ubyte_file, 'rb').read()

    # 解析文件头信息，依次为魔数、图片数量、每张图片高、每张图片宽
    offset = 0
    fmt_header = '>iiii' #因为数据结构中前4行的数据类型都是32位整型，所以采用i格式，但我们需要读取前4行数据，所以需要4个i。我们后面会看到标签集中，只使用2个ii。
    magic_number, num_images, num_rows, num_cols = struct.unpack_from(fmt_header, bin_data, offset)
    print('魔数:%d, 图片数量: %d张, 图片大小: %d*%d' % (magic_number, num_images, num_rows, num_cols))

    # 解析数据集
    image_size = num_rows * num_cols
    offset += struct.calcsize(fmt_header)  #获得数据在缓存中的指针位置，从前面介绍的数据结构可以看出，读取了前4行之后，指针位置（即偏移位置offset）指向0016。
    print(offset)
    fmt_image = '>' + str(image_size) + 'B'  #图像数据像素值的类型为unsigned char型，对应的format格式为B。这里还有加上图像大小784，是为了读取784个B格式数据，如果没有则只会读取一个值（即一副图像中的一个像素值）
    print(fmt_image,offset,struct.calcsize(fmt_image))
    images = np.empty((num_images, num_rows, num_cols))
    #plt.figure()
    for i in range(num_images):
        if (i + 1) % 10000 == 0:
            print('已解析 %d' % (i + 1) + '张')
            print(offset)
        images[i] = np.array(struct.unpack_from(fmt_image, bin_data, offset)).reshape((num_rows, num_cols))
        #print(images[i])
        offset += struct.calcsize(fmt_image)
#         plt.imshow(images[i],'gray')
#         plt.pause(0.00001)
#         plt.show()
    #plt.show()
```

```
30
31      return images
```

1.3.2 idx1-ubyte 文件解码

```
1   def decode_idx1_ubyte(idx1_ubyte_file):
2       # 读取二进制数据
3       bin_data = open(idx1_ubyte_file, 'rb').read()
4
5       # 解析文件头信息，依次为魔数和标签数
6       offset = 0
7       fmt_header = '>ii'
8       magic_number, num_images = struct.unpack_from(fmt_header, bin_data, offset)
9       print('魔数:%d, 图片数量: %d张' % (magic_number, num_images))
10
11      # 解析数据集
12      offset += struct.calcsize(fmt_header)
13      fmt_image = '>B'
14      labels = np.empty(num_images)
15      for i in range(num_images):
16          if (i + 1) % 10000 == 0:
17              print ('已解析 %d' % (i + 1) + '张')
18          labels[i] = struct.unpack_from(fmt_image, bin_data, offset)[0]
19          offset += struct.calcsize(fmt_image)
20      return labels
```

1.3.3 加载训练图片

```
1   def load_train_images(idx_ubyte_file=train_images_idx3_ubyte_file):
2       """
3       TRAINING SET IMAGE FILE (train-images-idx3-ubyte):
4       [offset] [type]          [value]          [description]
5       0000     32 bit integer  0x00000803(2051) magic number
6       0004     32 bit integer  60000            number of images
7       0008     32 bit integer  28               number of rows
8       0012     32 bit integer  28               number of columns
9       0016     unsigned byte   ??               pixel
10      0017     unsigned byte   ??               pixel
11      ........
12      xxxx     unsigned byte   ??               pixel
13      Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (
                                                 white), 255 means foreground (black).
14
15      :param idx_ubyte_file: idx文件路径
16      :return: n*row*col维np.array对象，n为图片数量
17      """
```

```
18      return decode_idx3_ubyte(idx_ubyte_file)
```

1.3.4 加载训练标签

```
1   def load_train_labels(idx_ubyte_file=train_labels_idx1_ubyte_file):
2       """
3       TRAINING SET LABEL FILE (train-labels-idx1-ubyte):
4       [offset] [type]          [value]          [description]
5       0000     32 bit integer  0x00000801(2049) magic number (MSB first)
6       0004     32 bit integer  60000            number of items
7       0008     unsigned byte   ??               label
8       0009     unsigned byte   ??               label
9       ........
10      xxxx     unsigned byte   ??               label
11      The labels values are 0 to 9.
12
13      :param idx_ubyte_file: idx文件路径
14      :return: n*1维np.array对象，n为图片数量
15      """
16      return decode_idx1_ubyte(idx_ubyte_file)
```

1.3.5 加载测试图片

```
1   def load_test_images(idx_ubyte_file=test_images_idx3_ubyte_file):
2       """
3       TEST SET IMAGE FILE (t10k-images-idx3-ubyte):
4       [offset] [type]          [value]          [description]
5       0000     32 bit integer  0x00000803(2051) magic number
6       0004     32 bit integer  10000            number of images
7       0008     32 bit integer  28               number of rows
8       0012     32 bit integer  28               number of columns
9       0016     unsigned byte   ??               pixel
10      0017     unsigned byte   ??               pixel
11      ........
12      xxxx     unsigned byte   ??               pixel
13      Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (
                                                 white), 255 means foreground (black).
14
15      :param idx_ubyte_file: idx文件路径
16      :return: n*row*col维np.array对象，n为图片数量
17      """
18      return decode_idx3_ubyte(idx_ubyte_file)
```

1.3.6 加载测试标签

```
def load_test_labels(idx_ubyte_file=test_labels_idx1_ubyte_file):
    """
    TEST SET LABEL FILE (t10k-labels-idx1-ubyte):
    [offset] [type]          [value]          [description]
    0000     32 bit integer  0x00000801(2049) magic number (MSB first)
    0004     32 bit integer  10000            number of items
    0008     unsigned byte   ??               label
    0009     unsigned byte   ??               label
    ........
    xxxx     unsigned byte   ??               label
    The labels values are 0 to 9.

    :param idx_ubyte_file: idx文件路径
    :return: n*1维np.array对象，n为图片数量
    """
    return decode_idx1_ubyte(idx_ubyte_file)
```

1.4 测试图片读取

```
train_images = load_train_images()
train_labels = load_train_labels()
test_images = load_test_images()
test_labels = load_test_labels()
plt.figure(figsize=(10,10))
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow(train_images[i], cmap='gray')
    plt.title("label: "+str(int(train_labels[i])))
plt.show()
```

输出如图1.4所示：

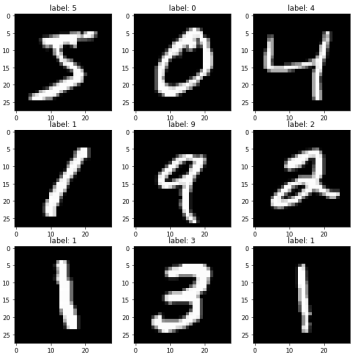

图 1.4: 测试数据读取

第 2 章　kNN 分类器实现

2.1　kNN 原理介绍

2.1.1　什么是 kNN

k 最近邻 (kNN，k-NearestNeighbor) 分类算法是数据挖掘分类技术中最简单的方法之一。所谓 K 最近邻，就是 k 个最近的邻居的意思，说的是每个样本都可以用它最接近的 k 个邻居来代表。

kNN 算法的核心思想是如果一个样本在特征空间中的 k 个最相邻的样本中的大多数属于某一个类别，则该样本也属于这个类别，并具有这个类别上样本的特性。该方法在确定分类决策上只依据最邻近的一个或者几个样本的类别来决定待分样本所属的类别。kNN 方法在类别决策时，只与极少量的相邻样本有关。由于 kNN 方法主要靠周围有限的邻近的样本，而不是靠判别类域的方法来确定所属类别的，因此对于类域的交叉或重叠较多的待分样本集来说，kNN 方法较其他方法更为适合。

下面通过一个简单的例子说明一下：如下图，绿色圆要被决定赋予哪个类，是红色三角形还是蓝色四方形？如果 K=3，由于红色三角形所占比例为 2/3，绿色圆将被赋予红色三角形那个类，如果 K=5，由于蓝色四方形比例为 3/5，因此绿色圆被赋予蓝色四方形类。由此也说明了 kNN 算法的结果很大程度取决于 K 的选择。

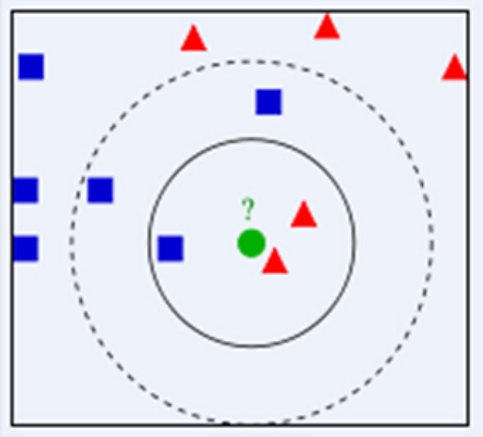

图 2.1: kNN 原理介绍图

在 kNN 中，通过计算对象间距离来作为各个对象之间的非相似性指标，避免了对象之间的匹配问题，在这里距离一般使用欧氏距离 $d(x,y)=\sqrt{\sum_{k=1}^{N}(x_k-y_k)^2}$ 或曼哈顿距离 $d(x,y)=\sum_{k=1}^{N}|x_k-y_k|$。同时，KNN 通过依据 k 个对象中占优的类别进行决策，而不是单一的对象类别决策。这两点就是 KNN 算法的优势。

2.1.2 kNN 算法流程

接下来对 KNN 算法的思想总结一下：就是在训练集中数据和标签已知的情况下，输入测试数据，将测试数据的特征与训练集中对应的特征进行相互比较，找到训练集中与之最为相似的前 K 个数据，则该测试数据对应的类别就是 K 个数据中出现次数最多的那个分类，其算法的描述为：

Algorithm 1 kNN 算法

1. 计算测试数据与各个训练数据之间的距离
2. 按照距离的递增关系进行排序
3. 选取距离最小的 k 个点
4. 确定前 k 个点所在类别的出现频率
5. 返回前 k 个点中出现频率最高的类别作为测试数据的预测分类

2.2 数据预处理

从1我们已经从 Mnist 数据集读取了我们训练分类器所需要的图片和标签数据，然而得到的图片是 28*28 的数组，为了便于后续计算，我们需要将二维数组转为一维数组。

```
# 训练集图片
train_input=train_images.reshape((train_images.shape[0],train_images.shape[1]*
                                  train_images.shape[2]),order='F')
#测试集图片
test_input=test_images.reshape((test_images.shape[0],test_images.shape[1]*
                                test_images.shape[2]),order='F')
```

2.3 kNN 实现

2.3.1 计算距离矩阵

对于后续选合适的 k 值时，不用再重新计算距离矩阵。

```
def ComputerDistanceMat(testdata,traindata,testsize):
  #将序列转换成矩阵
  testdata=np.mat(testdata)
  traindata=np.mat(traindata)
  #获取traindata的第一维大小
  trainsize=traindata.shape[0]
  distancesort_mat=np.mat(np.zeros(shape=(testsize,trainsize)))
  for i in range(testsize):
      X=testdata[i,:]
      #计算L1
```

```
        distance=np.sum(np.array(np.tile(X,(trainsize,1))-traindata)**2,1)
        distancesort = distance.argsort()
        distancesort_mat[i]=distancesort
    return distancesort_mat
```

2.3.2 kNN 分类器

```
def KNN(distancesort_mat,trainlabel,testsize,k):
    ret_pred=[]
    trainlabel=np.mat(trainlabel)
    for i in range(distancesort_mat.shape[0]):
        countdict = dict()
        for j in range(k):
            Xlabel= trainlabel[0,int(distancesort_mat[i,j])]#distancesort存储的是训
                                                            练数据集的索引
            countdict[Xlabel] = countdict.get(Xlabel,0) + 1
        countlist = sorted(countdict.items(),key=lambda x:x[1],reverse = True)
        ret_pred.append(countlist[0][0])
    return ret_pred
```

2.3.3 计算召回率

```
def compute(ret_pred,testlabel,testsize,k):
    error=0
    for i in range(0,testsize):
        if(ret_pred[i]!=testlabel[i]):
            error+=1
    return 1-error/testsize
```

2.3.4 选取合适的 k

我们知道不同的 k 影响最终的分类准确率，因此我们需要通过实验选取合适的 k。

```
accuracy=np.zeros(shape=(50,1))
print("start")
for k in range(1,50):
    ret_pred=KNN(distancesort_mat,train_labels,testsize,k)
    ac_temp=compute(ret_pred,test_labels,testsize,k)
    accuracy[k]=ac_temp
    print("k:",k,"accuracy:",ac_temp)
print("end")
```

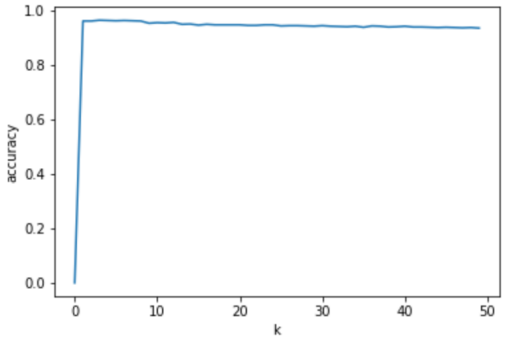

图 2.2: 不同 k 对应的准确率

选取合适的k

```
In [20]: accuracy=np.zeros(shape=(50,1))
         print("start")
         for k in range(1,50):
             ret_pred=KNN(distancesort_mat,train_labels,testsize,k)
             ac_temp=compute(ret_pred,test_labels,testsize,k)
             accuracy[k]=ac_temp
             print("k:",k,"accuracy:",ac_temp)
```

```
start
k: 1 accuracy: 0.962
k: 2 accuracy: 0.962
k: 3 accuracy: 0.965
k: 4 accuracy: 0.964
k: 5 accuracy: 0.963
k: 6 accuracy: 0.964
k: 7 accuracy: 0.963
k: 8 accuracy: 0.962
k: 9 accuracy: 0.954
k: 10 accuracy: 0.956
k: 11 accuracy: 0.955
k: 12 accuracy: 0.957
k: 13 accuracy: 0.95
k: 14 accuracy: 0.951
k: 15 accuracy: 0.947
k: 16 accuracy: 0.95
k: 17 accuracy: 0.948
k: 18 accuracy: 0.948
k: 19 accuracy: 0.948
k: 20 accuracy: 0.948
```

图 2.3: k 对应准确率

2.4 kNN 分类器训练结果

右图2.3可以看出，当我们选取 k 为 3 时可以得到较好的分类准确率，并且运算量相对较少。

第 3 章　朴素贝叶斯分类器实现

3.1　朴素贝叶斯 (Naive Bayesian, NB) 分类器原理

NB 分类算法是概率学派的经典算法，也是机器学习中的一个非常经典非常基础的分类算法，NB 算法有很强的数学理论作为支撑。

假设我们有一组训练数据：

$$\{(x_1, c_1), (x_2, c_2), \cdots, (x_n, c_n)\}$$

其中 ci 是每个数据的标签类别，xi 是特征向量，我们需要根据这些训练数据训练一个朴素贝叶斯分类器来对新数据进行分类。

既然是贝叶斯学派的算法，我们首先不得不引入贝叶斯公式，这也是算法的基础：

$$P(c|x) = \frac{P(c)P(x|c)}{P(x)} \tag{3.1}$$

这里可以进行理解：

1. $P(c|x)$：对于给定特征属性 x 条件下，数据属于 c 类别的概率
2. $P(c)$：样本空间中各个类别所占的比例
3. $P(x|c)$：对于给定类别的条件下的特征属性组合是 x 的概率

然而 $P(x|c)P(x|c)$ 是不能直接求得的,因为这里涉及到 x 的联合概率,即 $P(x|c)$ $P(x_1, x_2, \cdots, x_n|c)$ 直接估计每个类别 c 下的特征属性 x 的频率是不准确的，首先 x 的排列组合空间数可能会非常大，会引起组合爆炸问题，其次 x 的所有排列组合中有很多情况是样本数据中并没有出现的，这就引起了数据稀疏的问题，所以 $P(x|c)P(x|c)$ 是不能直接进行求解的。

为了解决上述问题，于是，在 NB 算法中作出一个非常重要的假设，那就是数据的每个属性是条件独立的，因为属性是相互独立的，那么我们就不需要使用联合概率来表示上面的 $P(x|c)$ $P(x_1, x_2, \cdots, x_n|c)$ 了，我们可以得到新的公式：

$$P(c|x) = \frac{P(c)P(x|c)}{P(x)} = \frac{P(c)}{P(x)} \prod_{i=1}^{d} P(x_i|c) \tag{3.2}$$

我们可以看到，因为有了属性条件独立的假设，$P(x|c)$ $\prod_{i=1}^{d} P(x_i|c)$，d 为属性个数。这样我们就很好进行计算了，只需要分别统计每个类别下的属性取值所占比例即可。这也就是我们的似然函数，我们可以将 $\prod_{i=1}^{d} P(x_i|c)$ 取对将连乘转换为累加，避免数据下溢，来进行计算[1]。

3.2 朴素贝叶斯分类器实现

3.2.1 数据预处理

从1我们已经从 Mnist 数据集读取了我们训练分类器所需要的图片和标签数据，然而得到的图片是 28*28 的数组，为了便于后续计算，我们需要将二维数组转为一维数组。

```
# 训练集图片
train_input=train_images.reshape((train_images.shape[0],train_images.shape[1]*
                                  train_images.shape[2]),order='F')
#测试集图片
test_input=test_images.reshape((test_images.shape[0],test_images.shape[1]*
                                test_images.shape[2]),order='F')
```

将图像进行二值化处理，以减少了属性量，从而减少了运算量。

```
def normalize(data):
  m=data.shape[0]
  n=np.array(data).shape[1]
  for i in range(m):
      for j in range(n):
          if data[i,j]!=0:
              data[i,j]=1
  return data
```

3.2.2 NB 分类器实现

首先计算先验概率和条件概率。

```
def BayesModel(traindata,trainlabel,classnum):
  traindatasize=traindata.shape[0]
  samplesize=traindata.shape[1]

  #先验概率
  PriorProbability=np.zeros(classnum)
  #条件概率
  ConditionalProbability=np.zeros((classnum,samplesize,2))
  #计算先验概率和条件概率
  for i in range(traindatasize):
      sample=traindata[i]
      label=int(trainlabel[i])
      #统计label类的label数量
      PriorProbability[label]+=1
      for j in range(samplesize):
          temp=int(sample[j])
          ConditionalProbability[label][j][temp]+=1

```

```
    #将概率归到[1.10001]
    for i in range(classnum):
        for j in range(samplesize):
            #经过二值化的图像只有0，1两种取值
            pix0=ConditionalProbability[i][j][0]
            pix1=ConditionalProbability[i][j][1]

            #计算0，1像素点对应的条件概率
            probability0=(float(pix0)/float(pix0+pix1))*10000+1
            probability1 = (float(pix1)/float(pix0 + pix1)) * 10000 + 1

            ConditionalProbability[i][j][0]=probability0
            ConditionalProbability[i][j][1]=probability1

    return PriorProbability,ConditionalProbability
```

对于给定的测试样本，可以计算先验概率与条件概率的乘积。

```
def calProbability(sample,label,PriorProbability,ConditionalProbability):
    probability=int(PriorProbability[label])#先验概率
    samplesize=sample.shape[0]
    for i in range(samplesize):#应该是特征数
        probability*=int(ConditionalProbability[label][i][sample[i].astype(int)])
    return probability
```

这样就可以确定该样本所属的分类，相当于做了 argmax 运算。

```
def predict(testdata,testlabel,PriorProbability,ConditionalProbability,classnum):
    predictLabel=[]
    testdatasize=testdata.shape[0]
    testsamplesize=testdata.shape[1]
    for i in range(testdatasize):
        sample=np.array(testdata[i])
        label=testlabel[i]
        maxlabel=0
        maxprobability=calProbability(sample,0,PriorProbability,
                                      ConditionalProbability)
        for j in range(1,classnum):
            probability=calProbability(sample,j,PriorProbability,
                                       ConditionalProbability)
            if(maxprobability<probability):
                maxprobability=probability
                maxlabel=j
        predictLabel.append(maxlabel)
    return np.array(predictLabel)
```

3.3 分类器准确性分析

3.3.1 计算准确率

```
1  def calAccuracy(testlabel,predictlabel):
2    testdatasize=testlabel.shape[0]
3    errorcount=0
4    for i in range(testdatasize):
5        if(testlabel[i]!=predictlabel[i]):
6            errorcount+=1
7    accuracy=1.0-float(errorcount)/testdatasize
8    return accuracy
```

3.3.2 分析

最终训练出的分类器的准确率约为 84.5%，可以发现远低于 kNN 分类器的 96.5% 的分类准确性，这可能由于是对数据进行二值化处理导致的，然而如果不进行二值化处理，计算量过大，本人计算机跑不出来。

第 4 章　逻辑回归分类器实现

4.1　逻辑回归实现多分类原理

通常，主要使用逻辑回归解决二分类的问题，但同样也可以使用逻辑回归解决多分类问题。

4.1.1　one vs rest

由于概率函数 $h_\theta(x)$ 所表示的是样本标记为某一类型的概率，但可以将一对一（二分类）扩展为一对多（one vs rest）：

1. 将类型 class1 看作正样本，其他类型全部看作负样本，然后我们就可以得到样本标记类型为该类型的概率 p1;
2. 然后再将另外类型 class2 看作正样本，其他类型全部看作负样本，同理得到 p2；
3. 以此循环，我们可以得到该待预测样本的标记类型分别为类型 class i 时的概率 pi，最后我们取 pi 中最大的那个概率对应的样本标记类型作为我们的待预测样本类型。

4.1.2　softmax 函数

使用 softmax 函数构造模型解决多分类问题。softmax 回归分类器需要学习的函数为：

$$h_\omega(\vec{x}) = \frac{1}{\sum_{i=1}^{k} e^{\vec{\omega_i}\cdot\vec{x}+b_i}} \begin{bmatrix} \vec{\omega_1}\cdot\vec{x}+b_1 \\ \vec{\omega_2}\cdot\vec{x}+b_2 \\ \vdots \\ \vec{\omega_k}\cdot\vec{x}+b_k \end{bmatrix} \tag{4.1}$$

其中 k 个类别的个数这里写 $\vec{\omega_i}$ 和 b_i 为第 i 个类别对应的权重向量和偏移标量。其中 $\frac{\vec{\omega_j}\cdot\vec{x}+b_j}{\sum_{i=1}^{k} e^{\vec{\omega_i}\cdot\vec{x}+b_i}} = P(y=j|\vec{x})$ 可看作样本 X 的标签为第 j 个类别的概率，且有 $\sum_{j=1}^{k} P(y=j|\vec{x}) = 1$。

与 logistic 回归不同的是，softmax 回归分类模型会有多个的输出，且输出个数与类别个数相等，输出为样本 X 为各个类别的概率，最后对样本进行预测的类型为概率最高的那个类别。

我们需要通过学习得到 $\vec{\omega_i}$ 和 b_i，因此建立目标损失函数为：

$$J(\omega, b) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{l=1}^{k} 1\{y^{(j)}=l\} log(\frac{e^{\vec{\omega_l}x^{\vec{(j)}}+b_l}}{\sum_{i=1}^{k} e^{\vec{\omega_i}x^{\vec{(j)}}+b_i}}) \tag{4.2}$$

上式的代价函数也称作：对数似然代价函数。在二分类的情况下，对数似然代价函数可以转化为交叉熵代价函数。

其中 m 为训练集样本的个数，k 为类别的个数，$1\{\cdot\}$ 为示性函数，当 $y^{(j)}=l$ 为真时，函数值为 1 ，否则为 0 ，即样本类别正确时，函数值才为 1 。

通过对数性质以及梯度下降法和链式偏导，化简得：

$$\frac{\partial J(\omega,b)}{\partial \vec{\omega_r}}=-\frac{1}{m}\sum_{j=1}^{m}\left(x^{\vec{(j)}}-P(y^{(i)}=r|x^{(j)})*x^{\vec{(j)}}\right) \tag{4.3}$$

因此由梯度下降法进行迭代：

$$\vec{\omega_r}=\vec{\omega_r}-\alpha\frac{\partial J(\omega,b)}{\partial \vec{\omega_r}} \tag{4.4}$$

同理通过梯度下降法最小化损失函数也可以得到 b_i 的最优值。同逻辑回归一样，可以给损失函数加上正则化项。

当标签类别之间是互斥时，适合选择 softmax 回归分类器; 当标签类别之间不完全互斥时，适合选择建立多个独立的 logistic 回归分类器。由于 Mnist 数据集类别间互斥，故选择 softmax 回归分类器。

4.2 实现 softmax 回归分类器

由于最近在学习 tensorflow，想借用作业机会，运用 tensorflow 实现 softmax 回归分类器。

4.2.1 加载数据

tensorflow 内部集成了 Mnist 数据集，并且 label 文件格式是每个样本为一维 10 元素数组，若为哪一类则对应为 1。

```
#Mnist数据集
from tensorflow.examples.tutorials.mnist import input_data
mnist=input_data.read_data_sets('data/',one_hot=True)
trainimg=mnist.train.images #28*28*1
trainlabel=mnist.train.labels#1*10
testimg=mnist.test.images#28*28*1
testlabel=mnist.test.labels#1*10
```

4.2.2 softmax 分类器实现

分类器参数设置。

```
#逻辑回归
#参数设置
numClasses=10
inputSize=784
```

```
trainingIterations=50000
batchSize=64#设置batch大小
#指定x和y大小
X=tf.placeholder(tf.float32, shape=[None,inputSize])
y=tf.placeholder(tf.float32,shape=[None,numClasses])
#参数初始化
W1=tf.Variable(tf.random_normal([inputSize,numClasses],stddev=0))
B1=tf.Variable(tf.constant(0.1),[numClasses])
```

构建模型。

```
#构造模型
y_pred=tf.nn.softmax(tf.matmul(X,W1)+B1)

loss=tf.reduce_mean(tf.square(y-y_pred))
opt=tf.train.GradientDescentOptimizer(learning_rate=0.05).minimize(loss)

correct_prediction=tf.equal(tf.argmax(y_pred,1),tf.argmax(y,1))
accuracy=tf.reduce_mean(tf.cast(correct_prediction,"float"))

sess=tf.Session()
init=tf.global_variables_initializer()
sess.run(init)
```

迭代计算。

```
#迭代计算
for i in range(trainingIterations):
    batch=mnist.train.next_batch(batchSize)
    batchInput=batch[0]
    batchLabels=batch[1]
    _,trainingLoss=sess.run([opt,loss],feed_dict={X:batchInput,y:batchLabels})
    if i%1000==0:
        train_accuracy=accuracy.eval(session=sess,feed_dict={X:batchInput,y:
                                                             batchLabels})
        print("step %d, training accuracy %g"%(i,train_accuracy))
```

4.3 测试结果与分析

```
#测试结果
batch=mnist.test.next_batch(batchSize)
testAccuracy=sess.run(accuracy,feed_dict={X:batch[0],y:batch[1]})
print("test accuracy %g"%(testAccuracy))
```

最终得到模型准确率为 90.5%。

参考文献

[1] 周志华. 机器学习 [M]. 南京 : 清华大学出版社, 2016.