

# KNN和SVM实现图像分类

---

## KNN和SVM实现图像分类

### (一) 实验原理

1. CIFAR10数据集
2. KNN
3. SVM
4. HOG直方图

### (二) 实现说明

1. 工具函数实现
2. KNN
3. SVM
4. HOG

### (三) 运行说明

### (四) 结果展示及分析

1. KNN
2. SVM
3. HOG

### (五) 实验总结

### (六) 未来展望

## (一) 实验原理

### 1. CIFAR10数据集

CIFAR-10 是由 Hinton 的学生 Alex Krizhevsky 和 Ilya Sutskever 整理的一个用于识别普适物体的小型数据集。一共包含 10 个类别的 RGB 彩色图片：飞机（ airplane ）、汽车（ automobile ）、鸟类（ bird ）、猫（ cat ）、鹿（ deer ）、狗（ dog ）、蛙类（ frog ）、马（ horse ）、船（ ship ）和卡车（ truck ）。图片的尺寸为  $32 \times 32$ ，数据集中一共有 50000 张训练图片和 10000 张测试图片。

与 MNIST 数据集中目比，CIFAR-10 具有以下不同点：

- CIFAR-10 是 3 通道的彩色 RGB 图像，而 MNIST 是灰度图像
- CIFAR-10 的图片尺寸为  $32 \times 32$ ，而 MNIST 的图片尺寸为  $28 \times 28$ ，比 MNIST 稍大

而鸢尾花数据集数据量较小，只有约150条数据，每条样本4个属性，共3个类别。

因此，本实验采用CIFAR10数据集进行后续实验。

### 2. KNN

#### ■ 介绍

##### 1. 基本介绍

KNN算法是一个有监督学习中的分类算法，全称K Nearest Neighbors，意思是K个最近的邻居。K个最近邻居，类似无监督学习中的KMeans算法，K的取值肯定是至关重要的。KNN的原理即当预测一个新的值x的时候，根据它距离最近的K个点是什么类别来判断x所属类别。

##### 2. 距离计算

在机器学习领域有很多刻画两个向量距离的指标，例如欧几里得距离、余弦距离、曼哈顿距离、切比雪夫距离等，例如：

欧几里得距离：

$$L_p(x_i, x_j) = \left( \sum_{l=1}^n |x_i^{(l)} - x_j^{(l)}|^p \right)^{\frac{1}{p}}$$

曼哈顿距离：

$$L_1 = \sum_{k=1}^n |x_{1k} - x_{2k}|$$

选择不同的距离评价指标，将在一定程度上影响实验结果。

##### 3. K值选择

如果选择较小的K值，就相当于用较小的邻域中的训练实例进行预测，学习的近似误差会减小，只有与输入实例较近的训练实例才会对预测结果起作用，但缺点是学习的估计误差会增大，预测结果会对近邻的实例点分成敏感。如果邻近的实例点恰巧是噪声，预测就会出错。

如果选择较大K值，就相当于用较大邻域中的训练实例进行预测，其优点是减少学习的估计误差，但近似误差会增大，也就是对输入实例预测不准确，K值得增大就意味着整体模型变的简单。

因此，采用交叉验证的方式，将数据划分为多个部分，尝试每个折叠进行验证并取平均结果。

部分一	部分二	部分三	部分四(验证)	测试集
-----	-----	-----	---------	-----

部分一	部分二	部分三(验证)	部分四	测试集
-----	-----	---------	-----	-----

部分一	部分二(验证)	部分三	部分四	测试集
-----	---------	-----	-----	-----

部分一(验证)	部分二	部分三	部分四	测试集
---------	-----	-----	-----	-----

#### 4. 具体实现步骤

- 输入一个待分类数据，确定K值（K值用来确定需要几个“邻居”）
- 一次计算输入数据和训练数据距离
- 将距离从小到大排序，获取前k个距离段的数据
- 统计这k个数据所属的分类值，输出得分最高的分类作为该待分类数据所属的类别
- 优缺点
  1. 优点：
 

简单有效、重新训练代价低、算法复杂度低、适合类域交叉样本、适用大样本自动分类。
  2. 缺点：
 

惰性学习、类别分类不标准化、输出可解释性不强、不均衡性、计算量较大。

### 3. SVM

- 介绍：
  1. 基本介绍：
 

SVM 是一种线性分类器。其构造出一个矩阵，计算图像中3个颜色通道中所有像素的值与权重的矩阵乘，从而得到分类分值。根据我们对权重设置的值，对于图像中的某些位置的某些颜色，函数表现出喜好或者厌恶（根据每个权重的符号而定）。其中线性公式为：

$$f(x_i, W, b) = Wx_i + b$$

鉴于本题使用CIFAR10数据集进行测试，其中每个数据为32\*32的3通道RGB值，共3072个特征。同时本数据集中有10个类。因此得出的 $W$  矩阵的规模为10 \* 3072。其中每一行代表了对某个类别的特征模板，每一列构成了对检测图像特定位置和特定颜色通道(某个特征)的权值。

在计算过程中，为了便于后续推导，常举证加入平移的因素，将 $b$ 融入至矩阵 $W$ 中，构成调整矩阵。在原有的 $W$ 基础上，增加一列1，从而使得新矩阵和图像向量进行点积运算时相当于在原矩阵和图像向量进行点积运算之后又将所得分数进行了偏置。

$$f(x_i, W) = Wx_i$$

通过矩阵与图像向量的点积，可以计算出图像在每一类上的得分。最后将测试数据通过该调整矩阵，最后取得分最高的类别作为预测类别。
  2. 损失函数：
 

在训练过程中企图构建一个 $W$ ，使得正确类别在得分上比其他类别的得分均高出一个特定的阈值，这样能使得最后的正确类别得分较明显的高于其他值。因此定义损失函数，若不能高出该阈值，则可视为了产生了损失，并将损失应用于对 $W$ 的进一步改进上。

损失矩阵为：

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

其中 $s_{y_i}$  为在正确类上的得分， $s_j$  为在其他类上的得分， $\Delta$ 为阈值。训练矩阵的最终目的是将所有图像的该损失函数地平均值最小化，即尽可能满足每张图像在正确类别上的得分均比其他类别上的得分高出一个特定阈值的目标。

### 3. 正则化惩罚：

为了防止构造出的矩阵中权重过于复杂（即矩阵的特殊性非常高）而使得其在测试集上出现过拟合的现象，需要将矩阵的复杂程度纳入到损失函数中，对过于复杂的矩阵进行惩罚，以求得到尽可能简单、通用的分类矩阵。一般采用L2正则化、L1正则化、弹性网络正则化、最大规范正则化。经过查阅资料，实际评估后，本次实验采用L2正则化：

$$R(W) = \sum_k \sum_l W_{k,j}^2$$

因此，经过以上分析得到最终的损失函数为：

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i) + \lambda R(W)$$

其中 $\lambda$ 为超参数，后续将对此展开讨论。

#### ■ 优缺点：

##### 1. 优点：

- 1) 解决了小样本情况下的机器学习。
- 2) 与KNN分类器不同，SVM优势在于一旦通过训练学习到了参数，就可以将训练数据丢弃。同时该方法对于新的测试数据的预测非常快，只需要与权重 $W$ 进行一个矩阵乘法即可。
- 3) 由于使用核函数方法克服了维数灾难和非线性可分的问题，所以向高维空间映射时没有增加计算的复杂度。（由于支持向量机算法的最终决策函数只由少数的支持向量所确定，所以计算的复杂性取决于支持向量的数目，而不是整个样本空间的维数）。

##### 2. 缺点

- 1) 支持向量机算法对大规模训练样本难以实施，这是因为支持向量算法借助二次规划求解支持向量，这其中会设计 $m$ 阶矩阵的计算，所以矩阵阶数很大时将耗费大量的机器内存和运算时间。
- 2) 经典的SVM只给出二分类的算法，而在数据挖掘中，一般要解决多分类的分类问题，而支持向量机对于多分类问题解决效果并不理想。
- 3) 现在常用的SVM理论都是使用固定惩罚系数 $C$ ，但是正负样本的两种错误造成的损失是不一样的。

## 4. HOG直方图

在基础KNN和SVM的实现过程中，均采用图像三维RGB像素作为特征值，以此来区分不同的类别。这种做法显然是比较直接朴素的。每个图像对应3072个特征，且有50000训练数据。因此，在大规模数据的作用下，通过现有的分类器也能提取出一些有效特征。

但是仅仅通过颜色特征进行判别存在较大弊端，当图像背景色调近似，但主体类别不同，常常会因为背景占据较大部分而导致最终判别结果出错。我们需要寻找一种特征提取方式，进一步提取图像特征，使用更能代表图像类别的特征作为数据集，以实现更好的结果。于是，受到计算机视觉课程传授知识的启发，了解到HOG方向梯度直方图。图像的内容应该通过图像的形状来得到更好的反应，而图像局部目标的表象和形状能够被梯度或边缘的方向密度分布很好地描述。因此，将尝试使用HOG，提取更好的图片特征。

#### ■ 介绍：

##### 1. 基本介绍：

HOG即histogram of oriented gradient, 是用于目标检测的特征描述子，该技术将图像局部出现的方向梯度次数进行计数，该方法和边缘方向直方图、scale-invariant feature transform类似，不同的是HOG的计算基于一致空间的密度矩阵来提高准确率。Navneet Dalal and Bill Triggs首先在05年的CVPR中提出HOG，用于静态图像或视频的行人检测。

HOG的核心思想是所检测的局部物体外形能够被光强梯度或边缘方向的分布所描述。通过将整幅图像分割成小的连接区域（称为cells），每个cell生成一个方向梯度直方图，这些直方图的组合可表示出所检测目标的目标。此后block滑窗的方式求得整体图像的方向梯度直方图，并将其作为图像特征。

## 2. 实现步骤:

### ■ 图像灰度化

将CIFAR10数据集中的 $32 * 32 * 3$ 的RGB图像转化为 $32 * 32 * 1$ 的灰度图像。

### ■ 计算图像每个pixel上的gradient

对于每个像素绘制，采用如下方法计算水平和垂直方向上的灰度值梯度：

$$G_x(x, y) = H(x + 1, y) - H(x - 1, y)$$

$$G_y(x, y) = H(x, y + 1) - H(x, y - 1)$$

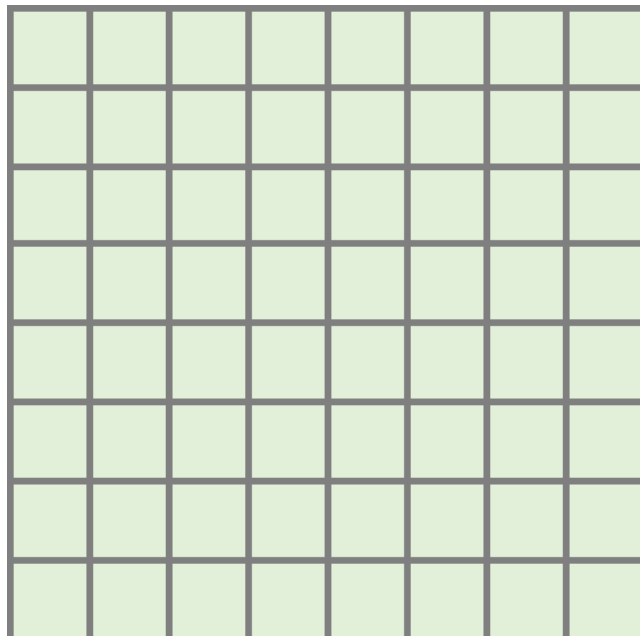
其中 $G_x(x, y)$ 表示水平方向上的梯度， $G_y(x, y)$ 表示垂直方向上的梯度， $H(x, y)$ 表示对应像素的灰度值。此后对水平和垂直方向上的梯度进行处理，求出梯度幅值和梯度方向：

$$G(x, y) = \sqrt{G_x(x, y)^2 + G_y(x, y)^2}$$

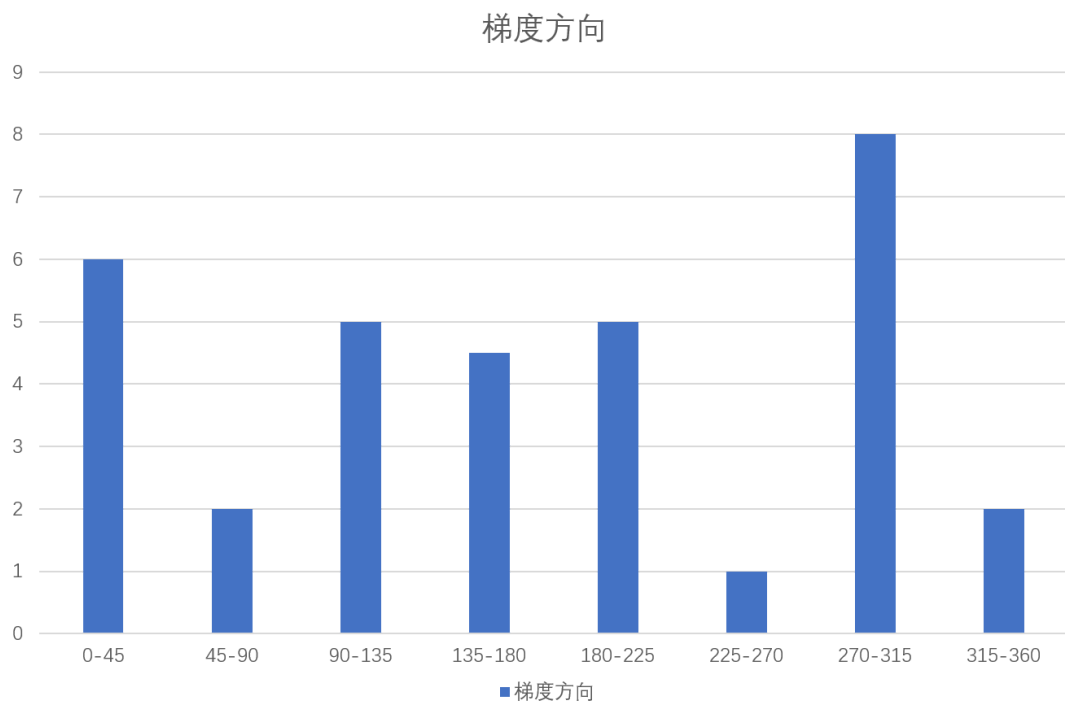
$$\alpha(x, y) = \tan^{-1}\left(\frac{G_y(x, y)}{G_x(x, y)}\right)$$

### ■ 进行cell划分，求得方向梯度直方图

将图像进行划分，每 $8 * 8$ 的像素划分为一个cell，每个cell提取一个方向梯度直方图信息。

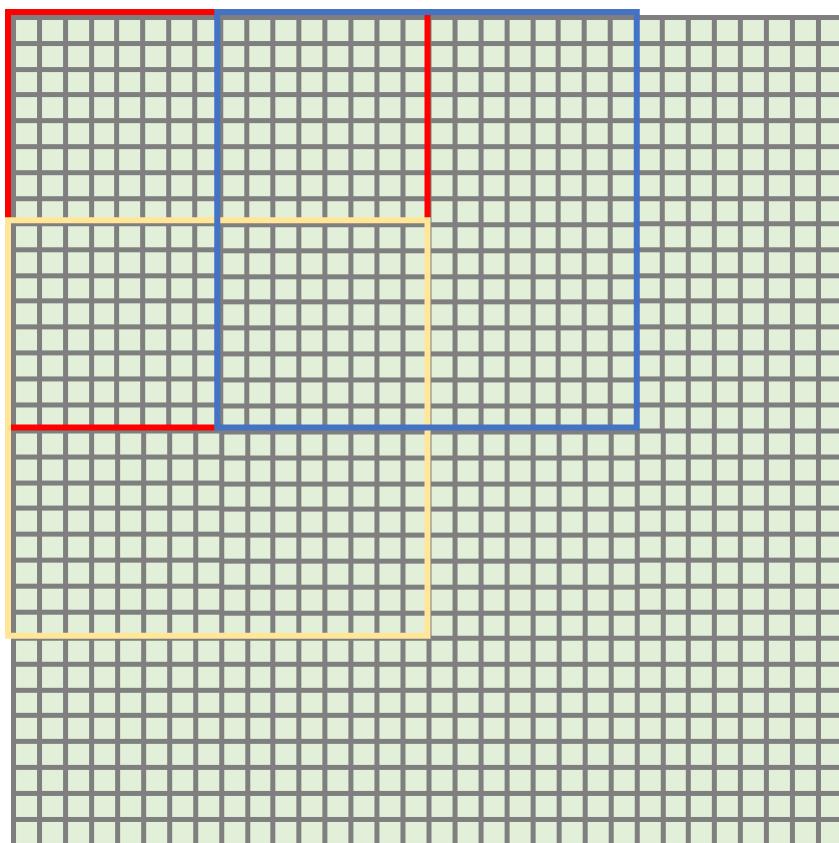


方向梯度直方图根据cell方向信息进行绘制，将0-360度划分为8类，每一类方向间隔为45度。当其梯度方向位于第 $i$ 个区间且其梯度幅值为 $a$ 时，就在其所属 cell 的方向梯度直方图中，将幅值加上 $a$ ，得到的方向梯度直方图示意如下所示：



#### ■ 滑动block

将每四个cell组合成一个block，并将cells的方向梯度直方图串联，得到特征向量 $(\alpha_1, \alpha_2, \dots, \alpha_{32})$ 。block采用  $2 * 2$  的方式组合cells，并且采用滑动窗的方式进行在一个Image中提取9个block。每个cell可以被多个block使用，作为其特征向量中的一部分：



由于实际图像中梯度幅值的变化和差异可能很大（受局部光照等因素影响），因此需要对每个block的特征向量进行归一化处理。

$$\alpha_i = \frac{\alpha_i}{\sqrt{\sum_{i=1}^{32} \alpha_i^2}}$$

每个block得到归一化后的32维特征向量 $(\alpha_1, \alpha_2, \dots, \alpha_{32})$ 。

#### ■ 连接block，形成图像特征

将所有block进行串联，得到该Image最终的特征，共 $32 * 9$ 维的 $(\alpha_1, \alpha_2, \dots, \alpha_{288})$ 的特征向量。对所有训练/测试集进行特征提取后，统一喂入分类器便可得到识别结果。

■ 优缺点：

1. 优点：

- 由于HOG是在图像的局部方格单元上操作，所以它对图像几何的和光学的形变都能保持很好的不变性，这两种形变只会出现在更大的空间领域上。
- HOG表示的是边缘（梯度）的结构特征，因此可以描述局部的形状信息。
- 采取在局部区域归一化直方图，可以部分抵消光照变化带来的影响。
- 位置和方向空间的量化一定程度上可以抑制平移和旋转带来的影响。

2. 缺点：

- 描述子生成过程冗长，导致速度慢，实时性差。
- 很难处理图像类别主体存在遮挡问题。
- 由于梯度的性质，HOG对噪点相当敏感。

## (二) 实现说明

### 1. 工具函数实现

本实验在tools.py文件中定义了在各种文件中均可能使用的工具函数：

#### ■ 获取数据集函数

从cifar10数据集中获取用于训练的数据集，并可根据获取比例，从原始数据集中提取部分数据作为运行数据，以降低程序的运行时长。其中参数ratioTrain、ratioTest、ratioVerify分别代表从数据集中提取训练集、测试集以及验证集的提取比，verifyNum表示选择那个batch作为验证集。

```
def datasetGet(ratioTrain = 0, ratioTest = 0, ratioVerify = 0, verifyNum = 5):

    # 生成训练集
    trainFileNum = 5                                # 数据集总
    个数
    trainData = []                                  # 训练集数
    据
    trainLabels = []                                # 训练集标
    记
    for i in range(trainFileNum):                    # 文件遍历
        if i+1 != verifyNum:                          # 验证集选
            择
            trainFile = open('../cifar10/data_batch_' + str(i+1), 'rb') # 打开训练
            集文件
            trainFileData = pickle.load(trainFile, encoding='bytes') # 存储为字
            典格式
            count = 0                                # 计数参数
            for each in trainFileData[b'data']:        # 数据遍历
                count += 1                             # 计算参数
                if ratioTrain != 0:                     # 需要进行
                    训练集分割
                    if count % ratioTrain == 0:        # 训练集
                        ratio
                        trainData.append(each)          # 训练集添
                        加
                    else:                                # 所有
                        cifar10添加
                        trainData.append(each)          # 数据添加
                        count = 0                        # 计数参数
                        for each in trainFileData[b'labels']: # 标签遍历
                            count += 1                 # 计数参数
                            if ratioTrain != 0:         # 需要进行
                                训练集分割
                                if count % ratioTrain == 0: # 训练集
                                    ratio
                                    trainLabels.append(each) # 训练集标
                                    签添加
                                else:                    # 所有
                                    cifar10添加
                                    trainLabels.append(each) # 数据添加

    # 验证集
    verifyData = []
```



```

verifyLabels = []
verifyFile = open('../cifar10/data_batch_'+str(verifyNum), 'rb') # 打开测试
集文件
verifyFileData = pickle.load(verifyFile, encoding='bytes') # 存储为字
典格式
count = 0
for each in verifyFileData[b'data']:
    count += 1
    if ratioVerify != 0: # 需要进行
验证集分割
        if count % ratioVerify == 0: # 验证集
ratio
            verifyData.append(each) # 验证集数
据添加
        else:
            verifyData.append(each)
count = 0
for each in verifyFileData[b'labels']:
    count += 1 # 用于计数
    if ratioVerify != 0: # 需要进行
验证集分割
        if count % ratioVerify == 0: # 验证集
ratio
            verifyLabels.append(each) # 验证集标
签添加
        else:
            verifyLabels.append(each)

# 生成测试集
testData = []
testLabels = []
testFile = open('../cifar10/test_batch', 'rb') # 打开测试
集文件
testFileData = pickle.load(testFile, encoding='bytes') # 存储为字
典格式
count = 0
for each in testFileData[b'data']:
    count += 1
    if ratioTest != 0: # 需要进行
测试集分割
        if count % ratioTest == 0: # 测试集
ratio
            testData.append(each) # 测试集数
据添加
        else:
            testData.append(each)
count = 0
for each in testFileData[b'labels']:
    count += 1
    if ratioTest != 0:
        if count % ratioTest == 0:
            testLabels.append(each)
    else:
        testLabels.append(each)

# 数据预处理
trainData = np.array(trainData).astype('float32') # 形成矩阵
trainLabels = np.array(trainLabels) # 形成矩阵
testData = np.array(testData).astype('float32') # 形成矩阵
testLabels = np.array(testLabels) # 形成矩阵
verifyData = np.array(verifyData).astype('float32') # 形成矩阵

```

```

verifyLabels = np.array(verifyLabels) # 形成矩阵
return trainData, trainLabels, testData, testLabels, verifyData, verifyLabels
# 返回数据集

```

#### ■ 显示曲线图可视化函数

```

# @function : 显示曲线图
def graphShow(x, y, xLabel, yLabel, title):
    plt.plot(x, y)
    plt.xlabel(xLabel)
    plt.ylabel(yLabel)
    plt.show()

```

#### ■ 显示散点图可视化函数

```

# @function : 显示散点图
def scatterShow(x, y, xLabel, yLabel, title):
    plt.title(title)
    plt.scatter(x, y)
    plt.xlabel(xLabel)
    plt.ylabel(yLabel)
    plt.show()

```

#### ■ 数据预处理, Z-score函数

在图像处理领域，对图像数据的中心化和归一化是很有必要的。因此，在实验过程中，通过 datasetGet 得到相应的函数后都需将对数据进行预处理，以提升训练精度和预测准确性。

```

# @function : 数据预处理-->中心化+归一化-->Z-Score标准化处理
def ZScore(dataSet):
    length = len(dataSet) # 原始数据长度
    total = np.sum(dataSet, axis=0) # 纵向求和
    ave = total.astype('float32')/length # 均值
    dataTemp = np.zeros(dataSet.shape) # 临时计算变量
    for i in range(length): # 每个数据减去均值以求得方差
        dataTemp[i] = dataSet[i] - ave # 减去均值
    dataTemp = dataTemp * dataTemp # 数据平方
    dataT = np.sum(dataTemp, axis=0) # 纵向求和
    for i in range(len(dataT)): # 开方得标准差
        dataT[i] = pow(dataT[i], 0.5) # 开方
    res = np.copy(dataSet) # 深拷贝原始数据
    for i in range(length): # 每隔数据执行
        res[i] = (dataSet[i]-ave)/dataT # 进行Z-Score划分
    return res # 返回中心化数据

```

#### ■ 数据预处理, 中心化函数

```

# @function : 数据中心化
def preProcessing(dataSet):
    return dataSet - np.mean(dataSet, axis=0) # 数据中心化

```

## 2. KNN

#### ■ KNN调用说明:

实现一个 gonKNN 类完成 KNN 分类器任务，使用过程中，只需要实例化 gonKNN 类，直接调用其中 predict 函数进行预测即可，返回预测结果列表。

```

# 获得原始数据集
trainData, trainLabels, testData, testLabels, verifyData, verifyLabels =
tools.datasetGet(ratioTrain=50, ratioTest=100, ratioVerify=50, verifyNum=5)

# 实例化+开始预测
preLabels = gonKNN(7).predict(trainData, trainLabels, testData)

```

#### ■ 代码实现细节：

KNN的实现在gonKNN类中进行实现，直接调用predict进行预测即可，KNN无需提前训练。

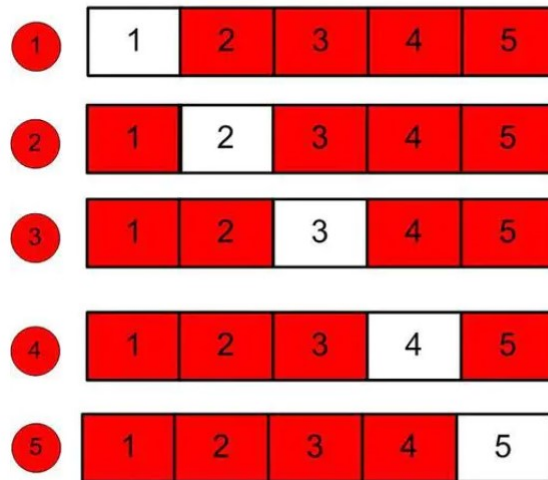
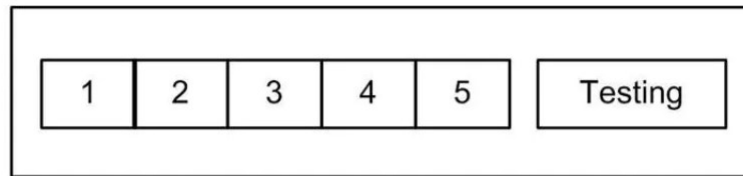
```

# @function : KNN分类器实现
class gonKNN:
    # @function : 初始化
    def __init__(self, k):
        assert k > 0
        self.k = k
        # 超参数
    # @function : 计算欧拉距离
    def eularDis(self, instance1, instance2):
        return np.sqrt(sum((instance1-instance2)**2))
        # 返回欧拉距离
    # @function : 实现KNN分类器预测
    def predict(self, trainData, trainLabels, testData):
        # 开始分类识别
        preLabels = []
        # 存储KNN分类结果
        for eachTest in tqdm(testData):
            # 进度条显示预测成果
            # 分类计算
            distancesAll = [self.eularDis(eachTrain,eachTest) for eachTrain in
trainData] # 求得每个训练数据像对测试数据的距离
            kMinIndex = np.argsort(distancesAll)[:self.k]
            # 求得距离最小的前k个训练数据的索引
            countRes = Counter(trainLabels[kMinIndex])
            # 对前k个训练数据的标签计数
            preLabels.append(countRes.most_common(1)[0][0])
            # 将出现频率最多的标签返回
        return preLabels
        # 返回预测结果标签

```

#### ■ 交叉验证获取最优超参数：

采用交叉验证求得最终超参数结果，共五个训练集，进行五次测试，每次选取训练集中的一个作为测试集。由于KNN的预测过程计算量极大，如果将原训练集10000 \* 5 全部用于验证，将花费大量时间，因此将取原训练集10000 \* 4 中的2%作为训练数据，原训练集10000 \* 1中的2%作为验证集，进行交叉验证。

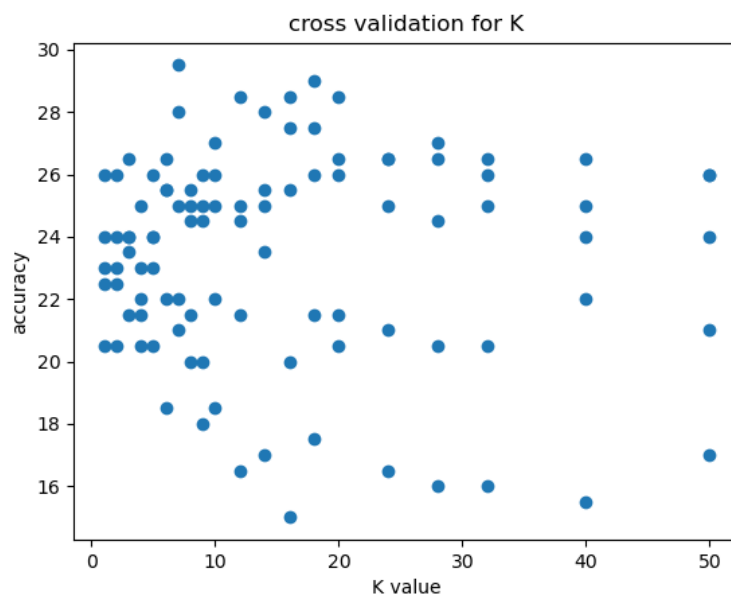


验证过程中，从1-50，选取一定间隔的k值若干，每个k执行五次交叉实验，以获得最终结果。

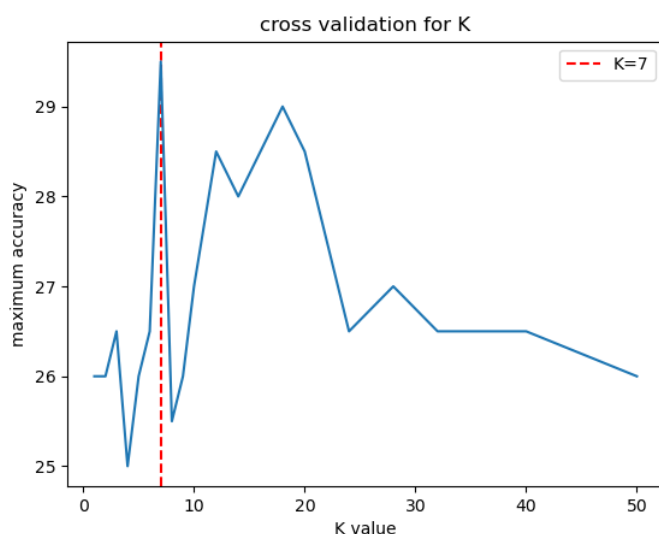
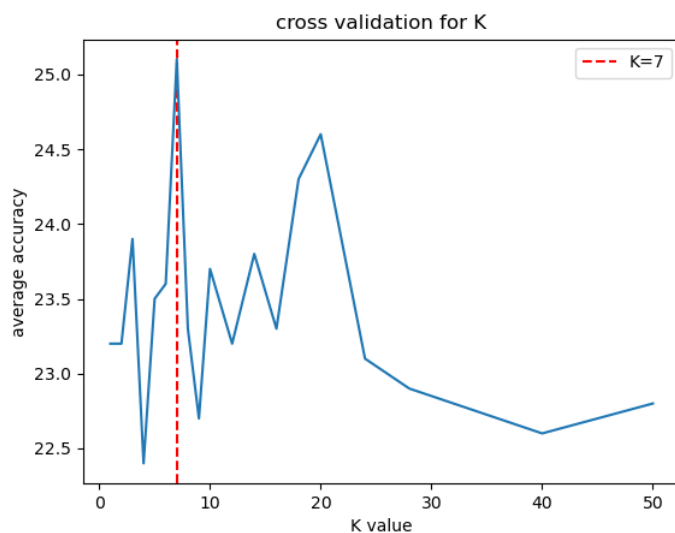
```

for i in range(len(k)):
    for j in range(5):
        trainData, trainLabels, testData, testLabels, verifyData, verifyLabels =
            tools.datasetGet(ratioTrain=50, ratioTest=100, ratioVerify=50, verifyNum=j+1)# 获取
            对应数据集
        preLabels = gonKNN(k[i]).predict(trainData, trainLabels, verifyData) # 使用
            验证集进行预测
        get = np.mean(preLabels == verifyLabels) * 100
        ans.append(get)
  
```

根据结果得到散点图：



分别根据散点图数据，对每个K下交叉验证数据取均值/最大值绘制折线图：



根据上述结果可以看出，无论是取均值还是关注最大值，毫无疑问当 $K=7$ 时，分类器呈现出最好分类效果，accuracy一度达到29.5%。因此，对于超参数 $K$ ，选择 $K=7$ 进行后续实验。

### 3. SVM

#### ■ SVM调用说明：

实现一个gonSVM类完成SVM分类器任务，使用过程中，首先实例化gonSVM对象，然后进行分类器权值的训练svmTrain，最后执行预测svmPredict。

```
# 获取原始数据
trainData, trainLabels, testData, testLabels, verifyData, verifyLabels =
tools.datasetGet()
# 实例化SVM对象
svm = gonSVM()
# 开始训练
lossHistory = svm.svmTrain(train_data, trainLabels, learningRate=1e-7,
alpha=5e3)
# 进行预测
res = svm.svmPredict(test_data)
```

#### ■ 代码实现细节：

SVM是一类需要提前训练的分类器，提前训练耗时较长，预测耗时较短。预测过程只需要调用提前训练好的模型即可。

```

# @function : SVM分类器实现
class gonSVM:
    # @function : 初始化
    def __init__(self):
        self.weight = None # 权重

    # @function : 计算SVM损失函数
    # @detail : 使用SVM的loss函数进行计算
    # @formula :  $1/N \sum_i \max\{L_i(f_i(x_i, \text{weight}), y_i)\} + \alpha \sum_k \text{Sigma}_k \text{Sigma}_l \text{weight}_{kl}^2$ 
    def svmLossGradient(self, batchData, batchLabels, alpha, delta):
        # 初始化
        batchSize = len(batchLabels)
        # 计算损失
        loss = 0.0 #
        # 存储损失函数值
        scores = batchData.dot(self.weight) #
        # 矩阵点乘运算(x_i * weight)-->产生batchSize*10的二维矩阵 每一行表示每个测试数据的判定结果
        correctScore = scores[range(batchSize), list(batchLabels)].reshape(-1, 1) #
        # 提取每行正确评判结果-->转换成batchSize行1列的正确评估值数据矩阵
        hingeValue = np.maximum(0, scores - correctScore + delta) #
        # (s_j-s_i+delta)折页损失计算结果中小于0的结果用0代替
        hingeValue[range(batchSize), list(batchLabels)] = 0 #
        # 纠正评估为正确的结果值为0
        loss += np.sum(hingeValue)/batchSize #
        # 折页损失
        loss += alpha * np.sum(self.weight * self.weight) #
        # 添加L2正则化
        # 计算梯度
        gradient = 0.0 #
        # 存储梯度结果矩阵 --> 3072*10
        maskArr = np.zeros(hingeValue.shape) #
        # hingeValue-->batchSize*10 的结果数组
        maskArr[hingeValue > 0] = 1 #
        # 所有不为0的位置变为1-->即所有得分与标准得分差大于delta的评估结果
        maskArr[range(batchSize), list(batchLabels)] = -np.sum(maskArr, axis=1) #
        # 求出所有需要计算的非零评估位个数-->num(-x_i^Tw_y_i)
        gradient += batchData.T.dot(maskArr)/batchSize #
        # 产生3073*10的求导结果矩阵
        gradient += 2 * alpha * self.weight #
        # 添加正则化求导结果
        return loss, gradient

    # @function : svm开始训练
    # @detail : 采用==>SVM损失函数+L2正则化+微分分析计算梯度+随机梯度下降
    # : 采用==>采用随机梯度下降方法进行实现
    def svmTrain(self, trainDataIn, trainLabels, learningRate=1e-3, delta=1.0,
alpha=1.0, epoch=2000, batchSize=200):
        # 数据初始化
        trainData = np.hstack((trainDataIn, np.ones((trainDataIn.shape[0], 1)))) #
        # 将W和b融合 --> n*1全1矩阵 --> 每个数据添加一个1
        trainNum, featureNum = trainData.shape
        classesNum = np.max(trainLabels) + 1
        # 权值初始化
        if self.weight == None:
            self.weight = 0.001 * np.random.randn(featureNum, classesNum) # 生成
            # 3073*10的二维数组
        # 开始训练
        lossHistory = [] # 用于
        # 存储迭代情况-->以展示随时函数迭代情况

```

```

        for i in range(epoch):                                # 显示
进度条
            indexBatch = np.random.choice(trainNum, batchSize, replace=False) # 在训
练集中随机抽取batchSize个数据的索引值(互不相同)
            trainBatch = trainData[indexBatch]
            labelsBatch = trainLabels[indexBatch]
            loss, gradient = self.svmLossGradient(trainBatch, labelsBatch, alpha,
delta) # 调用SVM类中函数 获取损失值及梯度值
            self.weight -= learningRate * gradient
            # 沿梯度下降方向改变权值weight
            lossHistory.append(loss)
            # 存储迭代信息情况
            # 展示迭代情况
            if (i+1) % 100 == 0:
            # 每迭代100次 显示迭代情况
                print('迭代次数为: %d / %d 损失函数值为: %f' % (i+1, epoch, loss))
            # 展示迭代情况
            return lossHistory
            # 返回迭代过程损失函数值变化情况 用于可视化

# @function : svm开始预测
def svmPredict(self, testDataIn):
    testData = np.hstack((testDataIn, np.ones((testDataIn.shape[0], 1)))) # 将w
和b融合 --> n*1全1矩阵 --> 每个数据添加一个1
    scores = testData.dot(self.weight)                                     # 根
据训练权重计算各类别得分
    res = np.argmax(scores, axis=1)                                       # 返
回max值的索引即得分最高类的类标签
    return res

```

#### ■ 交叉验证获取最优超参数:

SVM的验证过程中，只需要对测试数据通过一次权重 $w$ ，预测过程较快。主要时间集中在训练过程中，由于可以人为规定训练轮数，且当损失函数值收敛时候便可以不在进行训练，总体而言所需时间较短。cifar10训练集规模为 $10000 * 5$ 。因此，在交叉验证过程中，训练集采用  $10000 * 4$ ，验证集采用  $10000 * 1$  的规模，不进行压缩。

SVM中比较重要的超参数有：学习率learningRate，正则化系数alpha，loss阈值delta等，其中delta对训练结果影响较小，在本次实验过程中人为规定delta=1.0，主要关注对learningRate和alpha的验证，进而寻找最优超参数选择。同一个标准下，分别进行五次测试，每次选取训练集1-5中的一个作为验证集，剩余四个作为训练集。实现超参数的验证。

##### 1. 学习率learningRate交叉验证

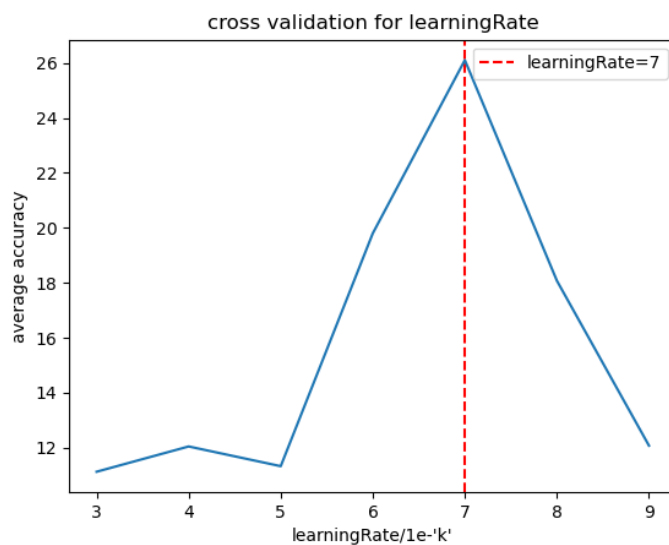
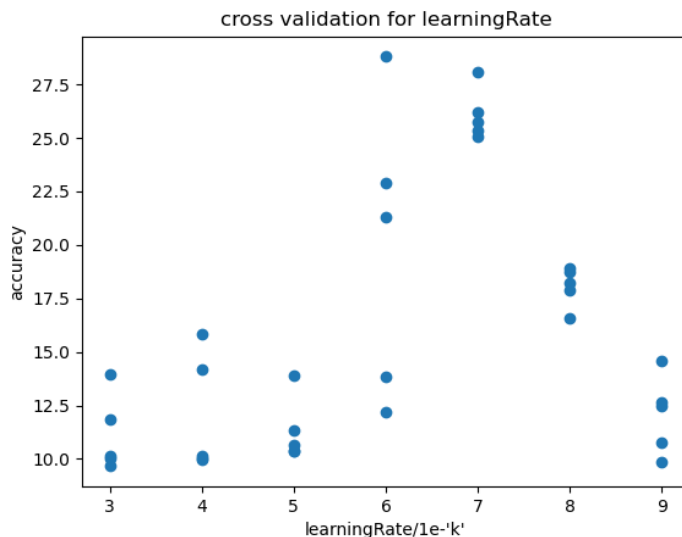
经过查阅资料，将alpha定义为 $5e2$ ，控制learningRate作为自变量，固定等间隔的量级作为评价指标，选定为 $1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3$ 作为学习率进行测试，在不同learningRate下进行交叉验证。

```

for i in range(len(learning)):
    for j in range(5):
        trainData, trainLabels, testData, testLabels, verifyData,
verifyLabels = tools.datasetGet(ratioTest=1000, verifyNum=j+1) # 获得
数据集
        svm = gonSVM() # 实例化SVM对象
        train_data = tools.preProcessing(trainData) # 数据预处理
        verify_data = tools.preProcessing(verifyData) # 数据预处理
        svm.svmTrain(train_data, trainLabels, learningRate=learning[i],
alpha=5e2) # 记录损失情况
        res = svm.svmPredict(verify_data)
        get = (np.mean(res == verifyLabels)*100)
        y.append(get)

```

根据预测结果绘制散点图，并将五次accuracy取平均得到折线图如下：



根据散点图可知，当学习率learningRate取 $1e-7$ 时效果较好，稳定性高，五次交叉验证准确率均较高。根据average accuracy折线图也可以看出，当learningRate当取 $1e-7$ 效果最好。因此，在此后的实验过程中学习率取值为 $1e-7$ 。

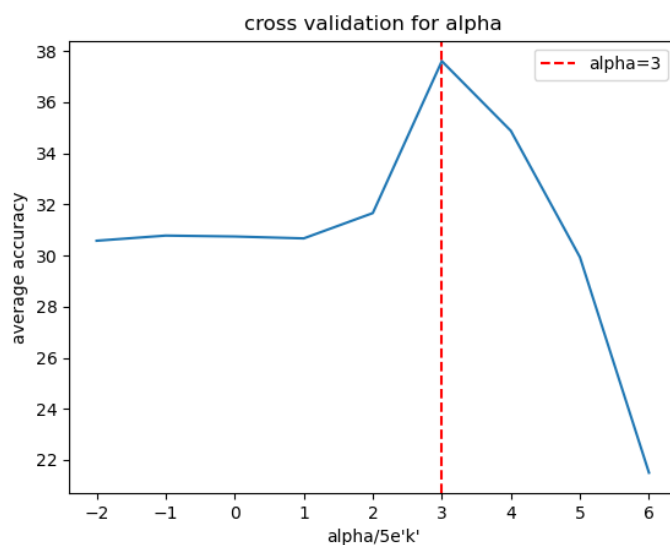
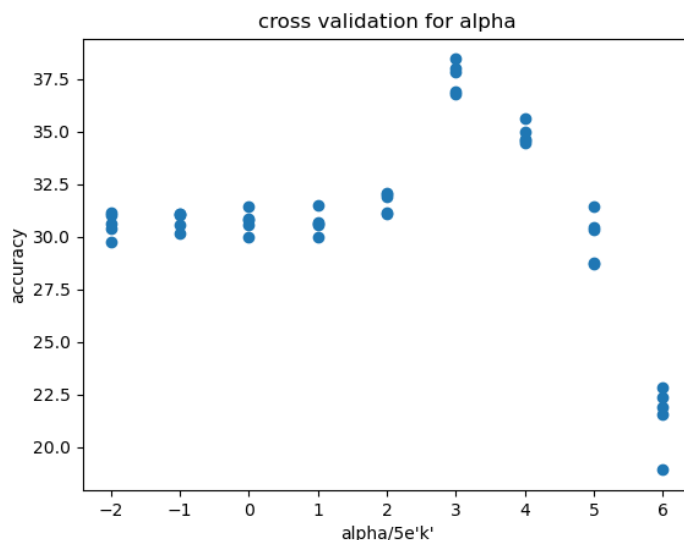
## 2. 正则化系数alpha交叉验证

根据上述实验得到最优学习率继续开展针对正则化系数的交叉验证。将learningRate固定为 $1e-7$ ，控制正则化系数alpha为自变量，固定等间隔的量级作为评价指标，选定为 $5e6$ ,  $5e5$ ,  $5e4$ ,  $5e3$ ,  $5e2$ ,  $5e1$ ,  $5e0$ ,  $5e-1$ ,  $5e-2$ 作为alpha进行测试，在不同alpha标准下进行交叉验证。

```
for i in range(len(alphaValue)):  
    for j in range(5):  
        trainData, trainLabels, testData, testLabels, verifyData, verifyLabels =  
            tools.datasetGet(ratioTest=1000, verifyNum=j+1)           # 获得数据集  
        svm = gonSVM()                                                # 实例化SVM对象  
        train_data = tools.preProcessing(trainData)                  # 数据预处理  
        verify_data = tools.preProcessing(verifyData)                # 数据预处理  
        svm.svmTrain(train_data, trainLabels, learningRate=1e-7,  
            alpha=alphaValue[i])                                     # 记录损失情况  
        res = svm.svmPredict(verify_data)  
        get = (np.mean(res == verifyLabels)*100)  
        y.append(get)
```

根据预测结果绘制散点图，并将五次accuracy取平均得到折线图如下：





根据散点图可知，当正则化系数 $\alpha$ 取 $5e3$ 时效果较好，稳定性高，五次交叉验证准确率均较高。根据average accuracy折线图也可以看出，当 $\alpha$ 当取 $5e3$ 效果最好。因此，在此后的实验过程中正则化系数 $\alpha$ 取值为 $5e3$ 。

根据交叉验证，我们得到最优超参数为：**学习率**learningRate =  $1e-7$ ；**正则化系数** $\alpha = 5e3$

## 4. HOG

### ■ HOG使用说明：

HOG是一个对图像数据进行特征处理的方法，因此只需要实现对应的特征提取的HOG类，返回对应图像的特征信息即可。使用过程中，只需在分类器使用之前对训练集和测试集数据进行特征提取，并将特征向量喂入分类器，使用测试集特征向量进行预测即可。

### ■ 代码实现细节：

```
# @function : 提取图像HOG特征
class HOG:
    def __init__(self):
        print('hello HOG')

    # @function : 对一个灰度图像image提取hog信息
    # @graph:
    #   Y--
    # X *****
    # | *
    # | *
```

```

#      *
#      *
#      *
#      *****
def hogImage(self, image, pixelsPerCell=(8, 8), cellsPerClock=(2,2), stride=8):

    # 数据初始化
    cellX, cellY = pixelsPerCell    # hog直方图提取过程中每个cell的规模
    blockX, blockY = cellsPerClock  # hog直方图提取过程中每个block的规模
    imageX, imageY = image.shape    # 提取特征的图像情况--> X表示行 Y表示列
    cellsNumX = int(np.floor(imageX // cellX))    # cell在x方向的数量
    cellsNumY = int(np.floor(imageY // cellY))    # cell在y方向的数量
    blocksNumX = cellsNumX - blockX + 1          # 滑动窗口取block, x方向的block数量
    blocksNumY = cellsNumY - blockY + 1          # 滑动窗口取block, y方向的block数量
    gradientX = np.zeros((imageX, imageY), dtype='float32') # 初始化X方向梯度存储
数组
    gradientY = np.zeros((imageX, imageY), dtype='float32') # 初始化Y方向梯度存储
数组
    gradient = np.zeros((imageX, imageY, 2), dtype='float32') # 存储各个方向的梯度
方向和梯度幅值

    # 整体梯度计算
    eps = 1e-5                                # 避免出现分母为0的误差值
    for i in range(1, imageX-1):              # X方向遍历
        for j in range(1, imageY-1):          # Y方向遍历
            gradientX[i][j] = image[i][j-1] - image[i][j+1] # X方向水平梯度
            gradientY[i][j] = image[i+1][j] - image[i-1][j] # Y方向垂直梯度
            gradient[i][j][0] = np.arctan(gradientY[i][j]/(gradientX[i][j]+eps))*180/math.pi # 算出该点的角度值
            if gradientX[i][j] < 0:
                gradient[i][j][0] += 180
    # 角度转换至0-180度的区间
        gradient[i][j][0] = int(gradient[i][j][0]+360) % 360
    # 保证梯度方向为正
        gradient[i][j][1] = np.sqrt(gradientY[i][j]**2+gradientX[i][j]**2)
    # 计算梯度幅值

    # 各个模块计算
    normalisedBlock = []
    # 存储最终结果
    for y in range(blocksNumY):
        for x in range(blocksNumX):
            block = gradient[y*stride:y*stride+blockY*stride, x*stride:
x*stride+blockX*stride] # 分离出一个block单独计算
            histogramBlock = []
            # 存储该block的直方图
            eps = 1e-5
            # 偏差参数 避免出现分母为0的情况
            for n in range(blockY):
                for m in range(blockX):
                    cell = block[n * stride:(n + 1) * stride, m * stride:(m +
1) * stride] # 在原有的block中分离出一个cell
                    histogramCell = np.zeros(8, dtype='float32')
                    # 初始化cell的梯度直方图结果
                    for p in range(cellY):
                        for q in range(cellX):
                            histogramCell[int(cell[q][p][0]/45)] += cell[q][p][1]
[1]
                    # 为所属方向上的梯度添加自己的值
                    if len(histogramBlock) == 0:
                        histogramBlock = histogramCell
                    else:

```

```

        histogramBlock = np.hstack((histogramBlock,
histogramCell))
        histogramBlock = np.array(histogramBlock)
        # 划归为矩阵
        histogramBlock = histogramBlock /
np.sqrt(histogramBlock.sum()*2+eps) # 直方图数据归一化
        if len(normalisedBlock) == 0:
            normalisedBlock = histogramBlock
        # 若为空则初始赋值
        else:
            normalisedBlock = np.hstack((normalisedBlock, histogramBlock))
        # 添加至最后的归一化数组数据中

    return normalisedBlock
    # 返回每张图像hog之后的结果数组

# @function : 对原始n*3073的数据进行处理
def hogProcess(self, originalData, fileName='data'):
    processedData=[] # 存储最终结果
    for data in tqdm(originalData): # 显示对应进度条
        image = np.reshape(data.T, (32, 32, 3)) # 一维组转化为图
片形式数据
        grayImage= cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)/255. # 转化为灰度图
        hogEach = self.hogImage(grayImage) # 提取灰度图hog特
征
        if len(processedData) == 0: # 若数组为空
            processedData = hogEach # 若为空则初始赋
值
        else: # 若数组不为空
            processedData = np.hstack((processedData, hogEach)) # 添加至最后的归
一化数组数据中
    processedData = np.array(processedData).astype('float32') # 转化为float型
矩阵
    processedData = np.reshape(processedData, (-1, 32*9)) # 转化为n*特征数
的数据集
    np.save(fileName+'.npy', processedData) # 保存在npy文件,
便于下次使用
    return processedData # 返回hog数据

# @function : 数据处理
def hogGetData(self, dataSet, fileName='0'):
    if os.path.exists(fileName+'.npy'):
        # 已经存在HOG特征提取的结果, 则直接使用即可。避免再次HOG提取占用大量时间
        print('existed') # 显示存在
        return np.load(fileName+'.npy') # 载入npy数据
    else:
        # 不存在HOG特征提取的结果, 则直接需要首先加载原数据, 再对原数据做HOG特征提取获
        得梯度特征, 最终是使用每张图片的梯度特征训练SVM并进行预测
        print('not existed') # 显示不存在
        return self.hogProcess(dataSet, fileName) # 进行计算

```

- 具体调用方式:

```

# 获取原始数据集
trainData, trainLabels, testData, testLabels, verifyData, verifyLabels =
tools.datasetGet(20, 100)
# 实例化HOG()
hog = HOG()

# 获取hog特征向量数据
test_data = hog.hogGetData(testData, '../npFile/testData_KNN_HOG')

train_data = hog.hogGetData(trainData, '../npFile/trainData_KNN_HOG')

```

注意：在第一次进行特征提取时，HOG算法将耗费大量时间。因此，在首次执行后同步将执行结果存入fileName.npy文件中，下次如需使用同样数据，直接读取文件即可，大大降低程序执行时间。

```

if os.path.exists(fileName+'.npy'):
    # 已经存在HOG特征提取的结果，则直接使用即可。避免再次HOG提取占用大量时间
    print('existed') # 显示存在
    return np.load(fileName+'.npy') # 载入numpy数据
else:
    # 不存在HOG特征提取的结果，则对原始数据进行特征提取
    print('not existed') # 显示不存在
    return self.hogProcess(dataSet, fileName) # 进行计算

```

### (三) 运行说明

- 本项目在配置有Anaconda里python3.7环境的PyCharm进行实现和运行。其中项目中所用的matplotlib、numpy、collections、matplotlib、pickle、tqdm、collections、cv2、math、os等库均已在Anaconda的对应环境中安装。运行时，直接在PyCharm IDE中运行或者直接在cmd窗口运行。
- cifar10数据集与code文件夹所处同目录，code中py程序直接调用cifar10文件夹中的数据集进行图像分类。
- 调用gonSVM\_HOG.py时，首次运行，程序会在npFile文件夹中生成对应的经过HOG的特征向量numpy文件。后续运行如果未修改程序数据集获取文件名，则将直接读取该numpy文件；否则，将生成对应文件名的numpy文件。

提交npFile文件夹中已含有hog特征提取完成的numpy文件，方便执行验证。

## (四) 结果展示及分析

### 1. KNN

根据以上交叉验证获取的最优超参数 $K=7$ 的情况下，使用cifar10训练集和测试集进行预测。程序执行结果如下：

```
开始执行KNN预测~~  
100%|██████████| 200/200 [01:00<00:00, 3.32it/s]  
预测准确度为：23.00%
```

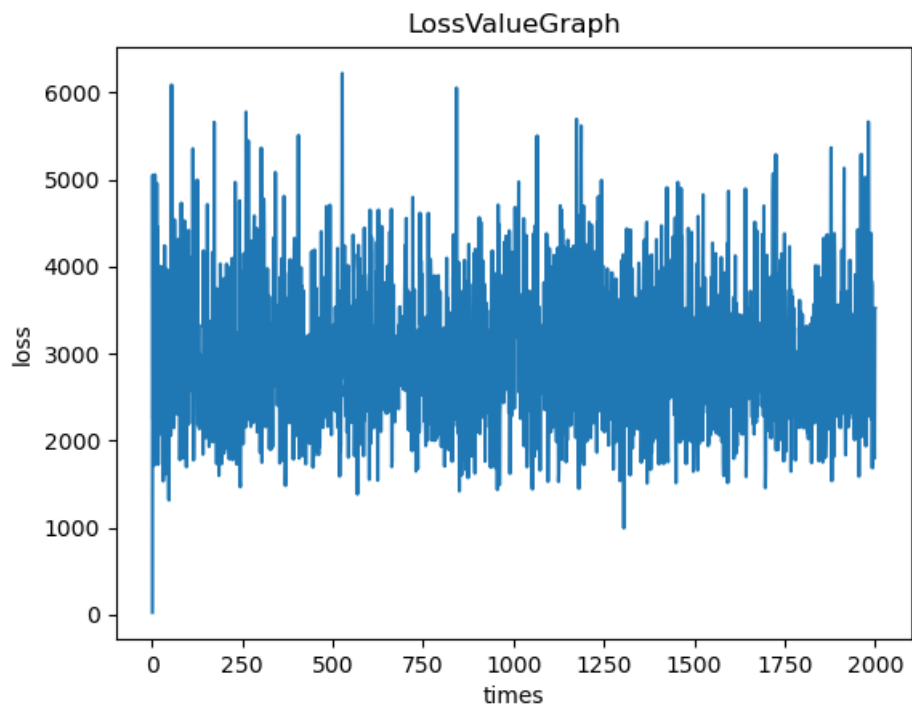
可见，使用KNN运行程序，准确率大致维持在23.00%左右。

测试集难以达到在交叉验证过程中的29.5%的较高准确率的情况。猜测可能是因为cifar10训练集和测试集之间存在一定差异而导致。由于在交叉验证过程中，训练集和验证集均在CIFAR10的10000 \* 5 原始训练数据集范围内选择，验证集和训练集之间的特征更加接近，因此预测准确度更高。而cifar10中测试集与训练集之间的特征之间差别较大，而KNN仅仅根据二者之间的距离作为衡量指标，模型鲁棒性较差。综上，在测试集上难以达到在验证集上的较高准确率。

### 2. SVM

- 首先使用随机的超参数（ $\text{learningRate}=1e-3$ ,  $\alpha=5e1$ ）的情况进行预测，观察程序运行结果，及loss函数值收敛情况，以此与后续最优超参数情况进行对比。SVM预测结果及loss损失函数图如下：

```
迭代次数为： 100 / 2000  损失函数值为： 3299.702722  
迭代次数为： 200 / 2000  损失函数值为： 3215.777661  
迭代次数为： 300 / 2000  损失函数值为： 2616.685432  
迭代次数为： 400 / 2000  损失函数值为： 2474.686275  
迭代次数为： 500 / 2000  损失函数值为： 2807.447510  
迭代次数为： 600 / 2000  损失函数值为： 2607.608665  
迭代次数为： 700 / 2000  损失函数值为： 4245.599667  
迭代次数为： 800 / 2000  损失函数值为： 1931.545641  
迭代次数为： 900 / 2000  损失函数值为： 3610.426285  
迭代次数为： 1000 / 2000  损失函数值为： 3265.443272  
迭代次数为： 1100 / 2000  损失函数值为： 3247.427352  
迭代次数为： 1200 / 2000  损失函数值为： 2726.016879  
迭代次数为： 1300 / 2000  损失函数值为： 1919.134733  
迭代次数为： 1400 / 2000  损失函数值为： 2870.483360  
迭代次数为： 1500 / 2000  损失函数值为： 2906.723172  
迭代次数为： 1600 / 2000  损失函数值为： 2873.608456  
迭代次数为： 1700 / 2000  损失函数值为： 4132.175002  
迭代次数为： 1800 / 2000  损失函数值为： 2466.839671  
迭代次数为： 1900 / 2000  损失函数值为： 3310.232584  
迭代次数为： 2000 / 2000  损失函数值为： 3521.431252  
测试集上准确率为：20.81 %
```



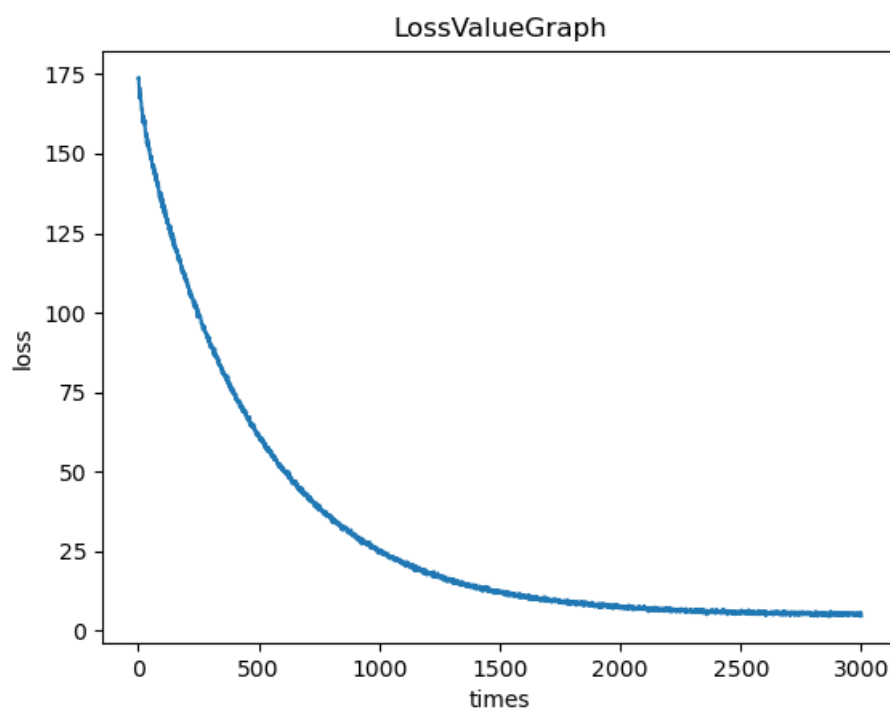
观察图像可得，损失函数无法收敛至一定的值，预测准确率20.81%也较低。因此，显然以上随机确定的超参数在预测功能的实现上存在问题，需要确定更为优秀的超参数，以实现loss的收敛，预测准确率最优。

- 根据以上交叉验证获取的最优超参数（ $\text{learningRate}=1\text{e-}7$ ,  $\alpha=5\text{e}3$ ）进行预测。并根据预测过程中的loss值绘制曲线图有：



迭代次数为：400 / 3000	损失函数值为：74.386132
迭代次数为：500 / 3000	损失函数值为：61.749531
迭代次数为：600 / 3000	损失函数值为：50.565836
迭代次数为：700 / 3000	损失函数值为：42.776032
迭代次数为：800 / 3000	损失函数值为：35.270175
迭代次数为：900 / 3000	损失函数值为：29.339545
迭代次数为：1000 / 3000	损失函数值为：25.379119
迭代次数为：1100 / 3000	损失函数值为：21.280499
迭代次数为：1200 / 3000	损失函数值为：18.064417
迭代次数为：1300 / 3000	损失函数值为：15.785586
迭代次数为：1400 / 3000	损失函数值为：13.751071
迭代次数为：1500 / 3000	损失函数值为：12.045460
迭代次数为：1600 / 3000	损失函数值为：10.495045
迭代次数为：1700 / 3000	损失函数值为：9.921037
迭代次数为：1800 / 3000	损失函数值为：8.849674
迭代次数为：1900 / 3000	损失函数值为：7.839431
迭代次数为：2000 / 3000	损失函数值为：7.500272
迭代次数为：2100 / 3000	损失函数值为：7.383610
迭代次数为：2200 / 3000	损失函数值为：7.100209
迭代次数为：2300 / 3000	损失函数值为：5.947030
迭代次数为：2400 / 3000	损失函数值为：6.436397
迭代次数为：2500 / 3000	损失函数值为：5.720489
迭代次数为：2600 / 3000	损失函数值为：5.806393
迭代次数为：2700 / 3000	损失函数值为：5.529042
迭代次数为：2800 / 3000	损失函数值为：4.550737
迭代次数为：2900 / 3000	损失函数值为：4.924192
迭代次数为：3000 / 3000	损失函数值为：4.545723

测试集上准确率为：38.18 %





使用交叉验证得到的最优超参数进行预测，可以看到loss函数能稳定收敛，并且识别准确率能达到38.18%的较高准确率。

根据以上信息我们可以看到，不同的超参数将极大的影响SVM分类器的分类结果。因此，使用交叉验证获取最优超参数的过程是及其有必要的。我们也可以看到，SVM的预测准确率相较于KNN有明显的提升，分类的执行效率也较高。

### 3. HOG

#### ■ HOG应用于KNN分类器

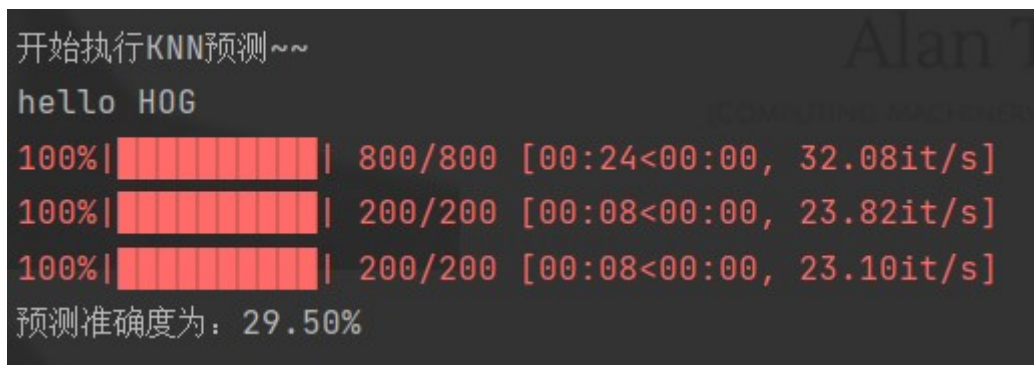
仍然使用最优的KNN的最优超参数 $K = 7$ 执行预测，数据集和测试集规模均不发生变化。

1. 执行代码为：

```
# HOG 应用于 KNN分类器
print('开始执行KNN预测~~')
# 数据集获取
trainData, trainLabels, testData, testLabels, verifyData, verifyLabels =
tools.datasetGet(ratioTrain=50, ratioTest=50, ratioVerify=50, verifyNum=2)
hog = HOG()
# 获取hog数据
train_data = hog.hogGetData(trainData, '../npyFile/trainData_KNN')

test_data = hog.hogGetData(testData, '../npyFile/testData_KNN')
# 开始预测
preLabels = gonKNN.gonKNN(7).predict(train_data, trainLabels, test_data)
# 结果展示
print('预测准确度为: %.2f%%' % (np.mean(preLabels == testLabels) * 100))
```

2. 运行结果为：



```
开始执行KNN预测~~
hello HOG
100%|██████████| 800/800 [00:24<00:00, 32.08it/s]
100%|██████████| 200/200 [00:08<00:00, 23.82it/s]
100%|██████████| 200/200 [00:08<00:00, 23.10it/s]
预测准确度为: 29.50%
```

由上图可看到，KNN的预测准确率由23.00%提升至29.50%，准确率提升较为明显。可知，HOG特征相较于原始图像RGB特征，KNN分类器上运行效果较好，准确率提升可达28.26%。

#### ■ HOG应用于SVM分类器

仍然使用最优的SVM的最优超参数 $learningRate = 1e - 7, alpha = 5e3$ 执行预测，数据集和测试集规模均不发生变化。需要对10000 \* 4个训练集数据和10000 \* 1个测试数据进行HOG特征值提取，花费时间较长，因此存储首次提取特征为npy文件是很有必要的，能极大的提升后续执行效率。

1. 执行代码为：

```
# HOG 应用于 SVM分类器
trainData, trainLabels, testData, testLabels, verifyData, verifyLabels =
tools.datasetGet()
# 实例化HOG()
hog = HOG()
# 获取hog数据
train_data = hog.hogGetData(trainData, '../npyFile/trainData_SVM_HOG')
```

```

test_data = hog.hogGetData(testData, '../npFile/testData_SVM_HOG')

# 实例化gonSVM()
svm = gonSVM.gonSVM()
# 获取损失值数据
lossHistory = svm.svmTrain(train_data, trainLabels, learningRate=1e-7,
alpha=5e3, epoch=3000)
# 进行预测
res = svm.svmPredict(test_data)

# 显示准确率结果
print('测试集上准确率为:%.2f %%' % (np.mean(res == testLabels) * 100))
# 可视化结果
tools.graphShow(np.linspace(1, 3000, 3000), lossHistory, 'times', 'loss',
'LossValueGraph')

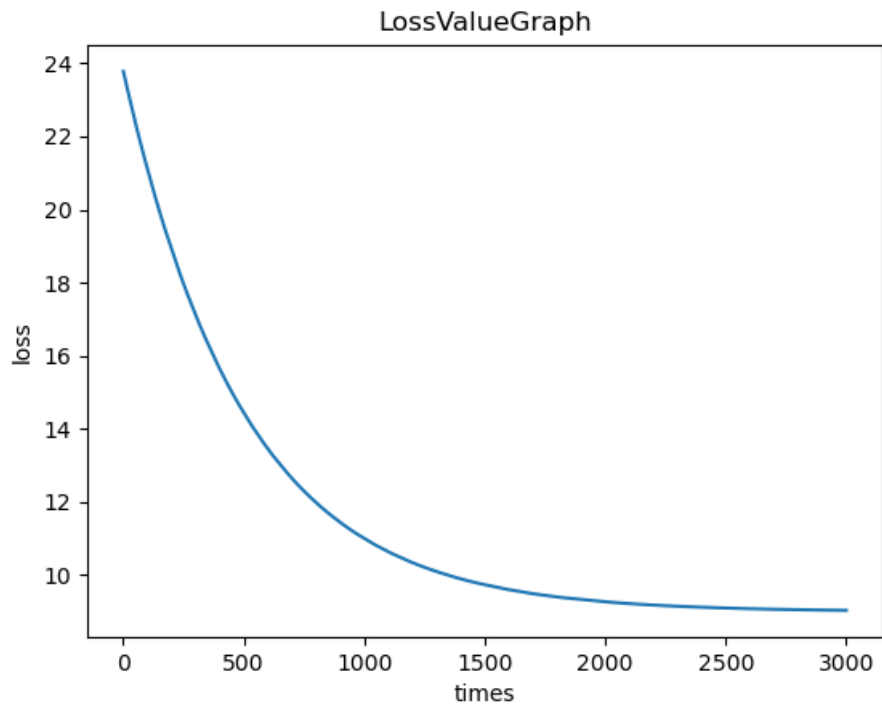
```

2. 运行结果为：

```

迭代次数为： 1000 / 3000  损失函数值为： 11.002514
迭代次数为： 1100 / 3000  损失函数值为： 10.639118
迭代次数为： 1200 / 3000  损失函数值为： 10.341826
迭代次数为： 1300 / 3000  损失函数值为： 10.098475
迭代次数为： 1400 / 3000  损失函数值为： 9.898983
迭代次数为： 1500 / 3000  损失函数值为： 9.736360
迭代次数为： 1600 / 3000  损失函数值为： 9.602655
迭代次数为： 1700 / 3000  损失函数值为： 9.493263
迭代次数为： 1800 / 3000  损失函数值为： 9.403749
迭代次数为： 1900 / 3000  损失函数值为： 9.330840
迭代次数为： 2000 / 3000  损失函数值为： 9.270717
迭代次数为： 2100 / 3000  损失函数值为： 9.221511
迭代次数为： 2200 / 3000  损失函数值为： 9.181335
迭代次数为： 2300 / 3000  损失函数值为： 9.148463
迭代次数为： 2400 / 3000  损失函数值为： 9.121600
迭代次数为： 2500 / 3000  损失函数值为： 9.099519
迭代次数为： 2600 / 3000  损失函数值为： 9.081410
迭代次数为： 2700 / 3000  损失函数值为： 9.066714
迭代次数为： 2800 / 3000  损失函数值为： 9.054621
迭代次数为： 2900 / 3000  损失函数值为： 9.044714
迭代次数为： 3000 / 3000  损失函数值为： 9.036609
测试集上准确率为：10.11 %

```



根据以上信息可以看到，使用HOG直方图作为特征送入SVM分类器进行训练，并对测试图像的HOG特征进行预测，准确率极低，仅为**10.11%**。根据loss函数曲线图可知，尽管已经收敛至一个较小值，模型已经达到理论较优情况，但是准确率极低，可能表示对于SVM分类器和CIFAR10数据集而言，HOG方向梯度直方图特征并不能很好的刻画图像特征，进而导致出现较大误差的情况。

由此可知，HOG直方图并不能很好的刻画CIFAR10数据集的特征，即使对于KNN有一点提升效果，但也影响较小；而对于SVM分类器则极大的降低预测准确率。因此，对于CIFAR10图像特征的提取，还需要进一步考虑其他方式。

## (五) 实验总结

本实验使用KNN、SVM对CIFAR10数据集进行分类预测，并辅以HOG直方图提取图像方向梯度特征，试图优化预测准确率。

1. 通过实验可以看到，KNN的分类准确率一般，仅25%左右，并且执行效率较低，每次预测均需要对训练集进行遍历计算，花费较多时间；而SVM相较于KNN有明显提升，通过训练集构建权重 $W$ ，预测过程只需要进行一次矩阵相乘即可，程序执行效率较高，并且预测准确率能接近40%。因此参数方法为代表的 **SVM 各方面明显优于 KNN**。
2. 与此同时，我们试图使用HOG对预测效果进行优化。但通过实验发现，HOG特征的代表效果不明显。对于KNN的预测准确率仅能提升至30%左右，提升幅度仅20%左右，提升效果并不明显；而对于SVM分类器，则极大地降低预测准确率，仅为15%左右，下降幅度达到60%左右。因此，使用HOG直方图作为图像特征进行训练，效果并不好。
3. 对于选择更好的特征用以代表图像，以达到提升预测准确率的目标。还需要在关注数据集特征和分类器特点的基础上，寻找更好的特征提取方式。

## (六) 未来展望

- 图像特征的提取上，可以考虑选择更贴合数据集的特征提取方式。例如：Canny边缘检测算子、形状特征(不变矩特征)、LBP特征、Haar特征等。
- 分类器的选择上，可以尝试其他更高效的分类器。例如：softmax分类器等。
- 选择更高效的图像分类方法。例如：CNN、迁移学习等。