人工智能原理第四次作业

一、EM算法

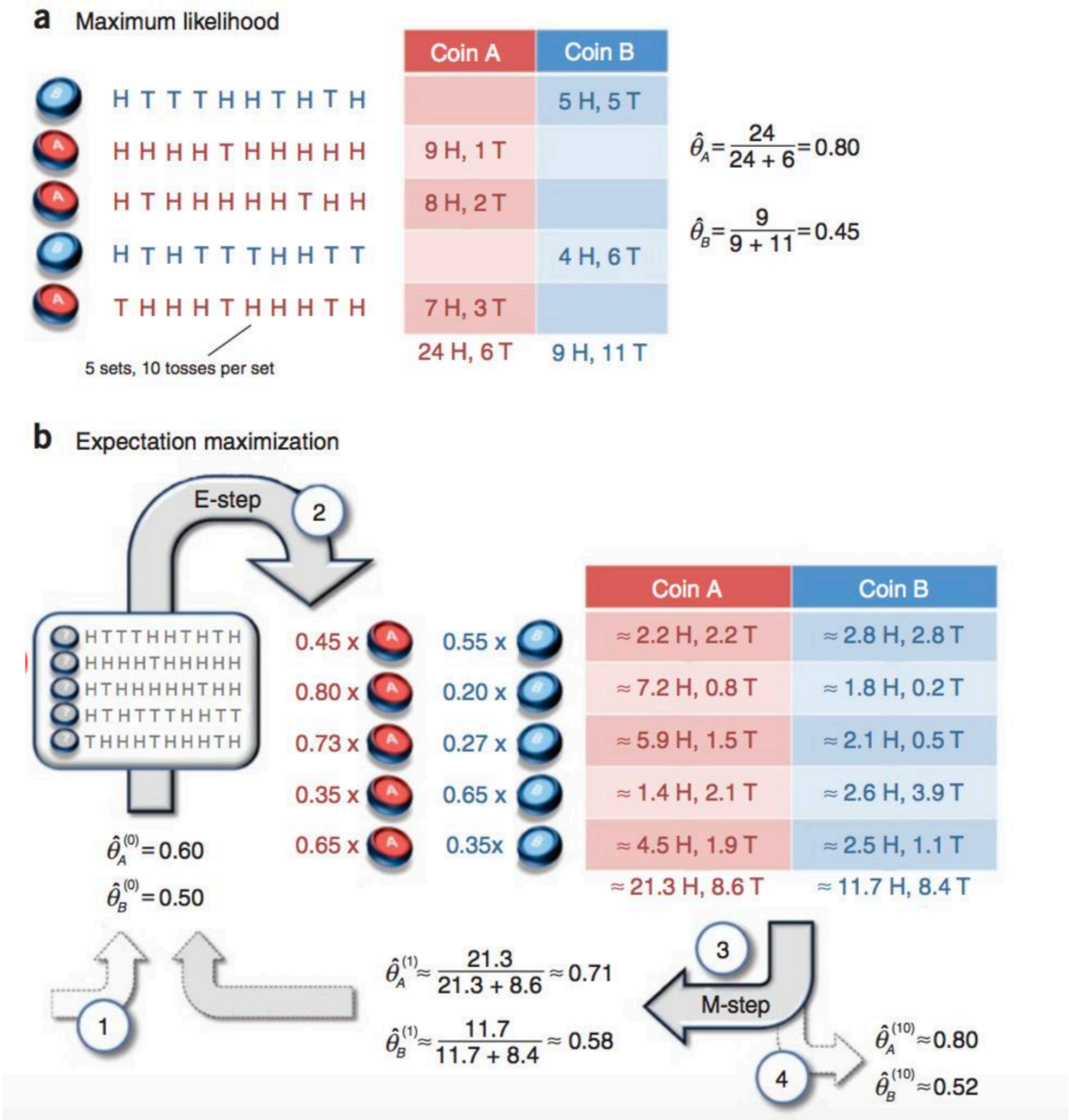
> EM: expectation maximization，最大期望算法

这个算法属于最大似然估计范畴。当我们不知道每个样本是从哪个分布来的，这个时候就可以使用EM算法思想。假设有两个样本A、B，并且都服从正态分布。那么就可以设$\theta a \theta b$分别为某个值，**然后随机取出一个个体，只需要判断它的特征更靠近哪一个样本就可以划分到对应的样本中，判断的依据就是θ值**（这个过程称为单步迭代）。经过划分后，我们可以得到AB样本中的数据了，那么就可以求出来新的$\theta a \theta b$，对于这两个值，我们再次进行迭代，可以得到新的值。当$\theta a \theta b$满足条件，或者趋于平衡时，这两个值就是我们要求的值。

题目：

假设有两枚硬币 A、B，以相同的概率随机选择一个硬币，进行如下的抛硬币实验:共做5次实验，每次实验独立的抛十次，结果如图中 a 所示，例如某次实验产生了 H、T、T、T、H、H、T、H、T、H，H 代表正面朝上。

假设试验数据记录员可能是实习生，业务不一定熟悉，造成如下图的 a 和 b 两种情况:



对于a种情况

由于扔硬币属于二项分布，所以只需要适应二项分布的共识即可求出来对应的概率，因为每个样本的来源是清楚的。

对于b种情况

对于每次一次E步的迭代，他的任务就是算出来一个新的$\theta a \theta b$，然后用于代替原来的。这里需要建立一个数据集：

```
observations = np.array([[1,0,0,0,1,1,0,1,0,1],
                         [1,1,1,1,0,1,1,1,1,1],
                         [1,0,1,1,1,1,1,0,1,1],
                         [0,1,1,1,0,1,1,1,0,1]])
```

抛硬币是一个二项分布，可以使用python自带的库。对于每一次的实验，可以根据公式求出来这一行属于A（或者B）的概率

```
contributionA = stats.binom.pmf(numH, obLen, thetaA)
contributionB = stats.binom.pmf(numT, obLen, thetaB)
```

将概率正规化就可以得到来自A（或者B）的概率。

有了这些值，就可以求出来本次E步迭代的对应来自A、B的总数了

```
counts['A']['H'] += weightA * numH
counts['A']['T'] += weightA * numT
counts['B']['H'] += weightB * numH
counts['B']['T'] += weightB * numT
```

那么就可以根据`counts`来求出来新的θ了。

```
    newThetaA = counts['A']['H'] / (counts['A']['H'] + counts['A']['T'])
    newThetaB = counts['B']['H'] / (counts['B']['H'] + counts['B']['T'])
```

上面叙述的就是EM的单次迭代。对于它的总迭代函数，我们只需要设置好循环的结束条件即可。以因为接下来就是调用单次迭代来得到新的θ的过程。

结果截图

- $\theta a = 0.5\ \theta b = 0.6$

```
>>> em(observations, [0.5,0.6])

[[0.796577914622738, 0.5601755372420011], 14]
>>>
```

  迭代了十四次，得到的结果和a很吻合。

- $\theta a = 0.3\ \theta b = 0.4$

```
>>> em(observations, [0.3,0.4])

[[0.7965779338296328, 0.5601756954441562], 14]
>>>
```

- $\theta a = 0.0000001\ \theta b = 0.88888888$

```
>>> em(observations, [0.0000001,0.88888888])

[[0.7965776260353056, 0.5601731983418142], 13]
>>>
```

可以看的出来EM算法还是很厉害的，当两个初始概率相差很大的时候，得到的结果还是很正确的。

二、图片分类

1、CIFAR-10数据集

运行以下代码，可以得出来这个数据集的一些特征

```python
import pickle

def load_file(filename):
    with open(filename, 'rb') as fo:
        data = pickle.load(fo, encoding='bytes')
    return data

# print(load_file('./data/data_batch_1'))
data = load_file('./data/data_batch_1')
# print(data)
print(data[b'data'].shape)
# print(data[b'labels'])
print(len(data[b'filenames']))
```

```
(10000, 3072)
10000
```

说明对于一个文件而言有10000个文件，每个文件像素大小为3072。一共有五个训练文件，一个测试文件。同时，`b'data'`是一个数组，而`b'labels'`是一个列表。

所以需要预先将训练数据和测试数据提取出来。也就是对读取几个文件，然后将得到的数据存放到数组中。整合后的数据规模如下：

```
训练数据规模：  (50000, 3072)
训练标签规模：  (50000,)
测试数据规模：  (10000, 3072)
测试标签规模：  (10000,)
bash-3.2$
```

2、K近邻算法

- 基本原理

  对于不同位置类别属性数据集中的点

  1. 计算已知类型数据集中两个点之间的距离，使用欧式距离公式

     ```
     distances = np.sum(np.abs(self.Xtr - self.x[i,:]) ** 2, axis=1) ** 0.5
     ```

  2. 按照距离递增的次序排序

     ```
     indexes = distances.argsort()
     ```

  3. 选取与当前点距离最小的K个点

```
#前k个
            for j in range(k):
                countY = self.Ytr[indexes[j]]

                #不存在则为0
                countDic[countY] = countDic.get(countY, 0) + 1
            sortedDic = sorted(countDic.items(),
key=operator.itemgetter(1), reverse=True)
```

4. 确定前K个点所在类别的出现频率
5. 返回前K个点出现频率最高的类别作为当前点的预测分类

由于如果对一个数据的预测，需要遍历整个数据集，所以耗费时间太长，所以就挑选数据内的部分内容进行计算来得到相应的结果，这样做的后果就是会导致准确率下降。运行后的结果如下：

```
11.50794/pythonFiles/ptvsd_
KNN accuracy:  0.26
运行时间21.63秒
bash-3.2$
```

3、SVM分类

> support vector machine（支持向量机）

做SVM分类器的主要实现方式是用线性分类实现。SVM的核心思想是：通过一个权重矩阵与样本相乘，然后算出来多个类别，哪一个类别分数高，那么就是哪个类别。SVM中还有一个间隔最大化的步骤，就是虽然一个类别的分数大于其他类别，但是还是需要判断这个分数是否大于我们规定的一个阈值，如果不大于，说明是有损失的。那么会出现两种情况：

- 损失函数等于0

  这种情况下不需要再解

- 损失函数不为0

  采用梯度下降求解

损失情况将会用做对模型准确度的评估，而梯度讲会在训练过程中，用于与学习率结合。

对于训练函数，如果将所有的训练数据都用做梯度下降算法，那么算法的时间实在是太久了，所以才用了随机抽取小样本进行测试。

```
        #梯度下降
        for i in range(iterations):
            #随机抽取样本来进行更新
            indexes = np.random.choice(train_scale, batch_size)
            loss,grad = self.loss(train_data[indexes], train_label[indexes],
reg)
            self.matrix = self.matrix - learning_rate * grad
```

但是这样做会导致准确率稍微低一点，同时每次训练后，测试的结果不同，有的偏低。

```
11.50794/pythonFiles/ptvsd_lau
SVM的准确度为： 0.305
运行时间1.61秒
bash-3.2$
```

4、二层神经网络分类器

二层神经网络是多层神经网络的一种。多层神经网络由三部分组成：输入层、隐藏层、输出层。输入层的作用是接受训练集的特征向量，经过连接节点的权重传入下一层。也就是说这一层是的输出是下层的输入。而对于每一层的节点权重更新，需要靠前向传播、反向传播两种方式进行，前向传播就是上面去提到的方式，反向传播就是下一层影响上一层。隐藏层的个数是任意的，隐藏层为两个是则称为二层神经网络。

对于损失函数来说，处理的过程和SVM的略微不一样，就是需要计算上前向传播和反向传播的权重矩阵，以及隐藏层的权重矩阵

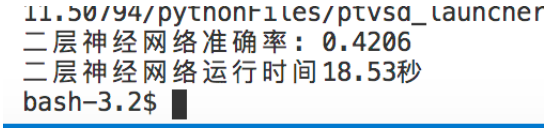
```
        #前向传播
        forward=np.maximum(0,np.dot(train_data,hidden_weight)+hidden_data)
        forward_output=np.dot(forward,output_weight)+output_data
        #计算损失
        #这里类似于SVM
        #需要根据前向传播的值来进行类别的归属判断
        loss_list = np.exp(forward_output[np.arange(data_scale), train_label])
        loss = sum(-np.log(loss_list / np.sum(np.exp(forward_output),
axis=1))) / data_scale + (np.sum(output_weight**2) * reg +
np.sum(hidden_weight**2))

        #反向传播
        #输出层梯度计算
        output_grad = np.exp(forward_output) / np.sum(np.exp(forward_output),
axis=1).reshape(data_scale, 1)
        output_grad[np.arange(data_scale), train_label] -= 1
        output_grad = output_grad / data_scale
        #权重矩阵
        output_data_grad = np.sum(output_grad, axis=0)
        output_matrix_grad = output_weight * 2 * reg  + np.dot(forward.T,
output_grad)

        #隐藏层梯度
        forward_grad = np.dot(output_grad, output_weight.T)
        forward_grad[forward<=0] = 0
```

然后就是将算出来的权重矩阵存放到`grads`中。

接着是训练函数，这个函数基本上和SVM的一样，都是需要随机取一些样本用于更新权重矩阵，同时还需要注意学习率的递减。

```
            #学习率衰减
            if i % reduce_rate == 0:
                learning_rate *= decay_rate
```

运行算结果如下：



5、三种分类器的对比

相比而言，KNN是最简单的一种分类器思想，但是正因为其简单，将会导致大量的运算，所以如果样将测试样本整体测试一遍，一个小时都不一定测的完（我就亲测过，一个小时还没运行完，只能终止），并且它的准确率只有0.26左右，这不是很高。SVM就比KNN略显高级一点，准去率和运行时间都比KNN的好，但是有一个缺陷就是，我们在训练更新数据时，才用的是随机采取小样本，这就会导致每次预测导致的结果都可能不一样，这主要是以内才用的是线性核函数，要想很完美的将两部分切割，仅仅是用线性是不够的。对于二层神经网络，解决问题上要比线性思想更具有优势，所以运行的结果总体上是要比前两种分类思想要好的。