统计机器学习实验报告

实验名称：多种模型的葡萄酒品种分类任务

姓名：	王恒
学院：	数学与统计学院
专业：	计算数学
学号：	220220934161

2023 年 6 月 12 日

摘要

本文中, 我们实现了朴素贝叶斯、逻辑斯谛回归、线性支持向量机、基于高斯核函数的非线性支持向量机、多层非线性感知机 (神经网络) 模型及其相应的学习算法, 并用以上模型对葡萄酒品种进行分类, 六种模型分别得到了 95.83%, 97.22%, 94.44%, 95.83%, 97.22% 的准确率.

具体的, 我们将数据集按照 $6:4$ 的比例划分为训练集和测试集, 并进行归一化. 其中, 由于该数据的所有特征都是连续型数据, 我们在实现朴素贝叶斯模型时, 将所有特征归一化到 $[-1,1]$, 并将其划分为固定的小区间, 模型中的概率均为特征值所处区间的概率. 对于其他四种回归模型, 我们在输入数据后先采用了批量归一化技术, 自动学习归一化参数, 相应的, 均采用小批量随机梯度下降算法在训练集上进行优化, 在测试集上测试准确率. 我们将本实验报告的所有内容开源:
`https://github.com/WANGH950/Statistical-Machine-Learning/tree/main/2ND`.

关键词: 朴素贝叶斯, 逻辑斯谛回归, 支持向量机, 神经网络, 葡萄酒品种分类

目录

1 开始实验

首先加载需要用到的包

```
import torch
import torch.nn as nn
from collections import OrderedDict
import copy
import time
from sklearn import datasets
import matplotlib.pyplot as plt
torch.manual_seed(100)
```

我们的所有实验都基于 PyTorch 神经网络框架实现, 该框架不仅包含了数值计算包 numpy 中的几乎所有工具, 还提供了编写神经网络的基本工具, 起到了类似于建房子时提供” 砖头” 和” 水泥” 的作用. 本次实验中所有需要计算梯度的地方都通过 PyTorch 中提供的自动微分算法实现, 需要使用的优化算法也由 PyTorch 框架提供. 我们将随机数种子设置为 100, 以保证实验结果可以复现.

我们使用 sklearn 中的 datasets 自动下载并加载数据, 之后将其转化为 torch.tensor 类型的数据, 以便后续使用 torch 的工具接口进行处理.

```
# 加载数据
data =  datasets.load_wine()
data_features = torch.tensor(data.data)
data_labels = torch.tensor(data.target)[:,None]
data_set = torch.cat([data_features,data_labels],dim=1)
```

我们使用 torch 中提供的划分数据集的工具, 将数据按照 6:4 随机划分为训练集和测试集.

```
length = data_set.shape[0]
alpha = 0.6
train_len = int(length*alpha)
test_len = length - train_len
train_data, test_data = torch.utils.data.random_split(data_set
    ,[train_len,test_len])
train_data = torch.tensor([item.numpy() for item in train_data
    ])
test_data = torch.tensor([item.numpy() for item in test_data])
```

2 实验结果分析

算法源码参见附录A, 本项目报告及实验结果开源: https://github.com/WANGH950/Statistical-Machine-Learning/tree/main/2ND.

2.1 朴素贝叶斯模型结果分析

葡萄酒品种分类数据集中的 13 个特征都为连续型数据, 因此我们在模型训练时保存了训练集特征的最小值和最大值, 并将所有特征值归一化到 $[-1,1]$, 确保它们的尺度相同.

由于归一化参数仅通过训练集得到, 用这些参数对测试数据进行归一化后, 其可能会分布到 $[-1,1]$ 以外, 因此我们将 $[-1,1]$ 等距划分为 $S-2$ 个小区间, 并分别计算了各特征在 $(-\infty,-1),[1,+\infty),[x_k,x_{k+1}),x_k=-1+\frac{2(k-1)}{S-2},k=1,2,\cdots S-1$ 个小区间内的条件概率

$$P_\lambda(X^j\in S_{jk}|Y=c_k)=\frac{\sum_{i=1}^N I(x_j\in S_{jk},y_i=c_k)+\lambda}{\sum_{i=1}^N I(y_i=c_k)+S\lambda} \tag{1}$$

和先验概率

$$P_\lambda(Y=c_k)=\frac{\sum_{i=1}^N I(y_i=c_k)+\lambda}{N+K\lambda}, \tag{2}$$

其中, $S_{jk}=[x_k,x_{k+1}),k=1,\cdots,S-1,S_{j0}=(-\infty,-1),S_{jS}=[1,+\infty)$, N 为训练样本数量, λ 为 Laplacian 平滑参数, K 为类别个数. 注意, 为方便起见, 我们将每个特征的区间划分数量设置为同一值 S.

Listing 1: 朴素贝叶斯模型训练算法部分源码

```
...
for i in range(self.K):
    labels_i = labels == self.full_labels[i]
    self.pre_prob[i] = (labels_i.sum() + self.lamb) / (N + self
        .K*self.lamb)
    for j in range(self.n):
        self.cond_prob[i,j,0] = 1 / self.S
        for k in range(self.S-1):
            l = -1 + k * delta_x
            r = l + delta_x
            features_ij = features[labels_i[:,0],j]
            features_ijk = features_ij[(features_ij>=l)*(
                features_ij<r)]
```

```
            self.cond_prob[i,j,k+1] = (features_ijk.shape[0] +
                self.lamb) / (labels_i.sum() + self.S*self.lamb)
...
```

我们实例化模型, 训练模型, 并在测试集上测试, 得到了 95.83% 的准确率.

2.2 Logistic 回归模型结果分析

我们采用多项逻辑斯谛回归模型

$$
\begin{aligned}
P(Y=k|x) &= \frac{exp(w_k \cdot x + b_k)}{1+\sum_{i=1}^{K-1} exp(w_k \cdot x + b_k)}, k = 1, 2, \cdots, K-1 \\
P(Y=K|x) &= \frac{1}{1+\sum_{i=1}^{K-1} exp(w_k \cdot x + b_k)}.
\end{aligned} \tag{3}
$$

以处理本文的三分类 ($K = 3$) 任务.

为学习模型参数, 我们实现了多类别的负对数似然损失函数

$$
L(\theta) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} log(\pi_{\theta_k}(x_i)). \tag{4}
$$

```
# 负对数似然损失函数
class NLLLoss(nn.Module):
    def __init__(self) -> None:
        super().__init__()

    def forward(self,y_pre,y_rel):
        assert y_pre.shape == y_rel.shape
        loss = -torch.log(y_pre)*y_rel
        return torch.sum(loss)
```

我们使用小批量梯度下降方法训练数据, 每次在训练数据中均匀随机采样 batch 条数据进行训练. 我们还在训练模型时将数据的标签进行 one-hot 编码, 以使用该方法.

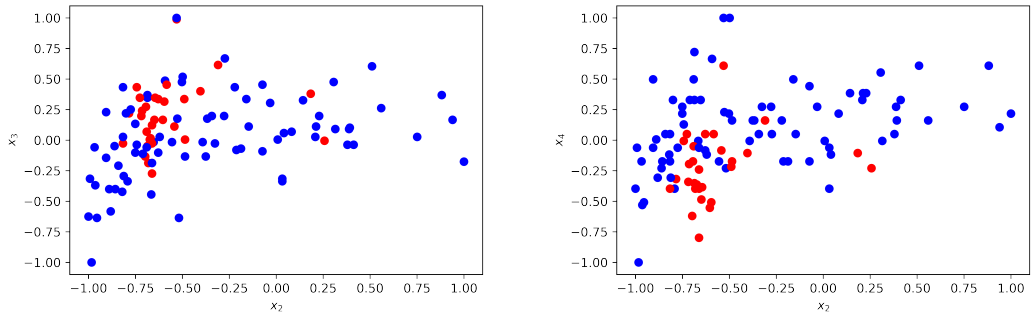
```
index = torch.randint(0,N,[batch_size]) # 随机选取batch条数据
data_i = data_set[index,:]
x_i = data_i[:,:-1].to(torch.float32)
y_i = nn.functional.one_hot(data_i[:,-1].to(torch.int64),
   num_classes=model.class_num)
```

在数据经过线性层以前, 我们先对其进行了批量归一化, 自动学习相应的归一化参数.

我们实例化模型, 选择批处理大小为 20, 在 $1e-3$ 的学习率下在训练数据上迭代了 $2,000$ 次, 得到了最终的模型. 我们使用训练好的模型在测试集上进行测试, 得到了 97.22% 的准确率. 该非线性模型效果略好于朴素贝叶斯模型.

2.3 支持向量机模型结果分析

我们分别实现了线性和非线性支持向量机模型. 如图1, 简单观察数据, 我们可以得到第二个特征和第三个特征以及第二个特征和第四个特征的非线性可分关系. 因此, 针对非线性支持向量机模型, 我们简单的考虑高斯核函数得到的特征



(a) 第二个特征和第三个特征关于标签 0 和非 0 绘制的散点图

(b) 第二个特征和第四个特征关于标签 0 和非 0 绘制的散点图

图 1: 非线性可分数据现象图像.

和原始特征结合, 再经过线性分类器对模型进行分类.

我们实现了合页损失函数

$$L(\theta) = \sum_{i=1}^{N} [1 - y_i(w \cdot x_i + b)]_+ + \lambda \|w\|^2, \tag{5}$$

以用于优化模型, 其中 $\theta = \{w, b\}$.

```
# 合页损失函数
class HingeLoss(nn.Module):
    def __init__(self,lamb) -> None:
        super(HingeLoss,self).__init__()
        self.lamb = lamb

    def forward(self,res_pre_linear_values,res_rel,parameters):
```

```
8         return torch.sum(torch.relu(1 - res_rel*
              res_pre_linear_values)) + self.lamb*torch.norm(
              parameters)**2
```

由于支持向量机是二分类模型, 我们将原始三分类问题分解为两个二分类问题, 并针对这两类二分类问题各训练了一个模型.

对于线性模型, 我们选取批处理大小 20, 正则化参数 $\lambda = 0.1$, 以 $1e-3$ 的学习率在训练数据上迭代了 $2,000$ 次得到了最终的模型. 对于非线性模型, 我们选择核函数参数 $N_s = 20$, 其余参数与线性模型相同. 我们使用训练好的线性模型和非线性模型, 在测试数据上分别得到了 $94.44\%, 95.83\%$ 的准确率. 结果表明, 非线性模型的效果略好于线性模型, 但都不如逻辑斯谛回归模型.

2.4 神经网络模型结果分析

多层感知机, 又名全连接前馈神经网络是现在深度学习模型中最基础也是最重要的模型之一. 其通过交替堆叠线性函数和非线性函数 (激活函数) 而达到对任意函数的万能逼近的作用. 全连接神经网络的结构如图2.

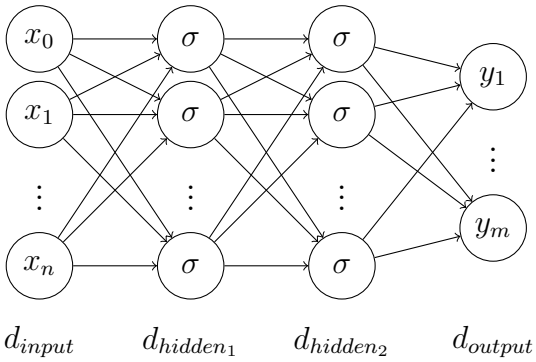


图 2: 两层隐藏层的全连接前馈神经网络结构, 其中带箭头的实线表示线性连接, 每条实线都对应一个权重, σ 是非线性函数, 为网络提供非线性因素.

我们搭建了以 Sigmoid Linear Unit(SiLU)

$$SiLU(x) = \frac{xexp(x)}{1 + exp(x)} \tag{6}$$

为激活函数的神经网络模型, 并在输出层使用 Softmax 函数

$$Softmax(y_i) = \frac{exp(y_i)}{\sum_{i=1}^{K} exp(y_i)} \tag{7}$$

归一化.

我们实例化了一个 3 层隐藏层, 每个隐藏层维度为 30 的全连接前馈神经网络. 该模型以 $1e-3$ 的学习率, 20 的批处理大小, 在训练数据上经过 $3,000$ 次迭代, 得到了最终的模型. 模型在测试数据上得到了 97.22% 的准确率, 精度与逻辑斯谛回归模型相似, 要优于其他模型.

A 模型及算法源码

A.1 朴素贝叶斯模型及其训练算法

Listing 2: 朴素贝叶斯模型及其训练算法实现

```
# 定义模型
class NaiveBayes(nn.Module):
    def __init__(self, n, full_labels, S, lamb) -> None:
        super(NaiveBayes,self).__init__()
        # 归一化参数
        self.max = None
        self.min = None
        self.n = n # 特征数量
        self.full_labels = full_labels # 所有标签
        self.K = len(full_labels) # 标签数量
        self.lamb = lamb # 贝叶斯估计参数lambda
        self.S = S # 每个特征分划区间数，这里默认都为S
        self.cond_prob = torch.zeros([self.K,self.n,S]) # 条件概率
        self.pre_prob = torch.zeros([self.K]) # 先验概率

    def forward(self, features):
        B,n = features.shape
        assert n == self.n
        post_prob = torch.ones([B,1])*self.pre_prob
        # 归一化
        features = (features - self.min) / (self.max - self.min) * 2 - 1
        delta_x = 2 / (self.S - 2)
        for i in range(B):
            for j in range(self.K):
                for k in range(n):
                    # (-\infty,-1)和(1,\infty)的概率
                    if features[i,k] < -1: post_prob[i,j] *= self.cond_prob[j,k,0]
                    elif features[i,k] >= 1: post_prob[i,j] *= self.cond_prob[j,k,-1]
```

```
                    else:
                        for h in range(self.S-2):
                            l = -1 + h * delta_x
                            r = l + delta_x
                            if features[i,k] >= l and features[
                               i,k] < r:
                                post_prob[i,j] *= self.
                                   cond_prob[j,k,h+1]
                                break
        return self.full_labels[torch.argmax(post_prob,dim=1)]

    def fit(self, train_data):
        # 计算先验概率
        N,_ = train_data.shape
        self.max = torch.max(train_data[:,:-1],dim=0).values
        self.min = torch.min(train_data[:,:-1],dim=0).values
        train_data[:,:-1] = (train_data[:,:-1] - self.min) / (
           self.max - self.min) * 2 - 1
        features = train_data[:,:-1] # 特征
        labels = train_data[:,-1:].int() # 标签
        delta_x = 2 / (self.S - 2)
        for i in range(self.K):
            labels_i = labels == self.full_labels[i]
            self.pre_prob[i] = (labels_i.sum() + self.lamb) / (
               N + self.K*self.lamb)
            for j in range(self.n):
                self.cond_prob[i,j,0] = 1 / self.S
                for k in range(self.S-1):
                    l = -1 + k * delta_x
                    r = l + delta_x
                    features_ij = features[labels_i[:,0],j]
                    features_ijk = features_ij[(features_ij>=l)
                       *(features_ij<r)]
                    self.cond_prob[i,j,k+1] = (features_ijk.
                       shape[0] + self.lamb) / (labels_i.sum()
                       + self.S*self.lamb)
        return self.pre_prob, self.cond_prob
```

A.2 Logistic 模型及相应学习算法

Listing 3: 逻辑斯谛回归模型及损失函数和训练方法

```
# 定义模型
class Logistic(nn.Module):
    def __init__(self,feature_num,class_num) -> None:
        super(Logistic,self).__init__()
        self.feature_num = feature_num
        self.class_num = class_num
        self.linear = nn.Sequential(
            nn.BatchNorm1d(feature_num), # 批量归一化
            nn.Linear(feature_num,class_num-1,bias=False)
        )

    def forward(self, x):
        B,d = x.shape
        assert d == self.feature_num and B > 0
        y = torch.cat([torch.exp(self.linear(x)),torch.ones([B
            ,1])],dim=1)
        y = y / torch.sum(y,dim=1,keepdim=True)
        return y

# 负对数似然损失函数
class NLLLoss(nn.Module):
    def __init__(self) -> None:
        super().__init__()

    def forward(self,y_pre,y_rel):
        assert y_pre.shape == y_rel.shape
        loss = -torch.log(y_pre)*y_rel
        return torch.sum(loss)

# 定义模型训练函数
def train(model, data_set, batch_size, epoch = 1000,
    learning_rate = 1e-3):
    N,_ = data_set.shape
    criterion = NLLLoss()
```

```
    optim = torch.optim.SGD(model.parameters(),learning_rate,
        momentum=0) # 动量设置为0

    start = time.time()
    loss_values = torch.zeros(epoch)
    for i in range(epoch):
        model.train()
        optim.zero_grad()
        index = torch.randint(0,N,[batch_size]) # 随机选取batch
            条数据
        data_i = data_set[index,:]
        x_i = data_i[:,:-1].to(torch.float32)
        y_i = nn.functional.one_hot(data_i[:,-1].to(torch.int64
            ),num_classes=model.class_num)
        outputs = model(x_i)
        loss = criterion(outputs,y_i)
        loss.backward()
        optim.step()

        model.eval()
        loss_values[i] = loss.item()

        print('\r%5d/{}|{}{}|{:.2f}s  [Loss: %e]'.format(
            epoch,
            "#"*int((i+1)/epoch*50),
            " "*(50-int((i+1)/epoch*50)),
            time.time() - start) %
            (i+1,
            loss_values[i]), end = ' ', flush=True)
    print("\nTraining has been completed.")
    return loss_values
```

A.3 支持向量机

Listing 4: 线性支持向量机

```
# 定义模型
class SVMLinear(nn.Module):
```

```
3       def __init__(self,feature_num) -> None:
4           super(SVMLinear,self).__init__()
5           self.feature_num = feature_num
6           self.batch_norm = nn.BatchNorm1d(feature_num) # 特征批
                量归一化，学习归一化参数
7           self.linear = nn.Linear(feature_num,1)
8
9       def forward(self, x, signed = True):
10          B,d = x.shape
11          assert B > 0 and d == self.feature_num
12          normed = self.batch_norm(x)
13          outputs = self.linear(normed)
14          if signed:
15              return torch.sign(outputs)
16          else:
17              return outputs
18
19  # 合页损失函数
20  class HingeLoss(nn.Module):
21      def __init__(self,lamb) -> None:
22          super(HingeLoss,self).__init__()
23          self.lamb = lamb
24
25      def forward(self,res_pre_linear_values,res_rel,parameters):
26          return torch.sum(torch.relu(1 - res_rel*
                res_pre_linear_values)) + self.lamb*torch.norm(
                parameters)**2
27
28  # 定义训练函数
29  def train_svm(model, data_set, batch_size, lamb=1, epoch =
        1000, learning_rate = 1e-3):
30      N,_ = data_set.shape
31      criterion = HingeLoss(lamb)
32      optim = torch.optim.SGD(model.parameters(),learning_rate,
            momentum=0) # 动量设置为0
33
34      start = time.time()
```

```
    loss_values = torch.zeros(epoch)
    for i in range(epoch):
        model.train()
        optim.zero_grad()
        index = torch.randint(0,N,[batch_size]) # 随机选取batch
            条数据
        data_i = data_set[index,:]
        x_i = data_i[:,:-1].to(torch.float32)
        y_i = data_i[:,-1:].to(torch.int32)
        outputs = model(x_i,signed=False)
        loss = criterion(outputs,y_i,model.linear.weight)
        loss.backward()
        optim.step()

        model.eval()
        loss_values[i] = loss.item()

        print('\r%5d/{}|{}{}|{:.2f}s  [Loss: %e]'.format(
            epoch,
            "#"*int((i+1)/epoch*50),
            " "*(50-int((i+1)/epoch*50)),
            time.time() - start) %
            (i+1,
            loss_values[i]), end = ' ', flush=True)
    print("\nTraining has been completed.")
    return loss_values
```

Listing 5: 非线性支持向量机

```
# 高斯核非线性SVM
class SVMNonLinear(nn.Module):
    def __init__(self,feature_num,kernel_data) -> None:
        super(SVMNonLinear,self).__init__()
        self.feature_num = feature_num
        self.kernel_features = kernel_data[:,:-1].to(torch.
            float32)
        self.kernel_labels = kernel_data[:,-1].to(torch.int32)
        self.batch_norm = nn.BatchNorm1d(feature_num)
```

```
9         self.linear = nn.Linear(feature_num+kernel_data.shape
              [0],1)
10        self.kernel_params = nn.Parameter(torch.rand(1)+5,
              requires_grad=True) # 核函数标准差
11
12    def forward(self, x, signed = True):
13        B,d = x.shape
14        assert B > 0 and d == self.feature_num
15        # 归一化特征
16        normed_data = self.batch_norm(x)
17        normed_kernel_data = self.batch_norm(self.
              kernel_features)
18        kernel_features = torch.exp(-torch.mm(normed_data,
              normed_kernel_data.T) / self.kernel_params**2) *
              self.kernel_labels
19        outputs = self.linear(torch.cat([normed_data,
              kernel_features],dim=1)) # 核方法特征和原特征结合
20        if signed:
21            return torch.sign(outputs)
22        else:
23            return outputs
```

A.4 全连接神经网络

Listing 6: 全连接神经网络模型实现

```
1  class MLP(nn.Module):
2  def __init__(self,feature_num,class_num,hidden_dim=20,layer_num
       =2) -> None:
3      super(MLP,self).__init__()
4      self.feature_num = feature_num
5      self.class_num = class_num
6      self.hidden_dim = hidden_dim
7      self.layer_num = layer_num
8      self.model = nn.Sequential(
9          OrderedDict(
10             [("input_layer",
11                 nn.Sequential(
```

```
                    nn.BatchNorm1d(feature_num),
                    nn.Linear(feature_num,hidden_dim),
                    nn.Tanh()
                ))] +
            [("hidden_layer_"+str(i+1),
                nn.Sequential(
                    nn.Linear(hidden_dim,hidden_dim),
                    nn.Tanh()
                )) for i in range(layer_num-1)] +
            [("output_layer",
                nn.Sequential(
                    nn.Linear(hidden_dim,class_num),
                    nn.Softmax(dim=1)
                ))]
        )
    )

# 前向传播
def forward(self,x):
    return self.model(x)
```