Lab1-SVM

520021910957 刘骏霖

一、 SVM using gradient descent

首先将数据载入，并对数据做处理：

1. 将 x 映射到【0，1】
2. 将 y 进行 0->-1，1->1 的映射

```
X_train = pd.read_csv("./data/X_train.csv")
# X_train.head(5)
normalizedTrainX = MinMaxScaler().fit_transform(X_train.values)
trainX = pd.DataFrame(normalizedTrainX)
# trainX.head(5)
```

```
X_test = pd.read_csv("./data/X_test.csv")
# X_test.head(5)
normalizedTestX = MinMaxScaler().fit_transform(X_test.values)
testX = pd.DataFrame(normalizedTestX)
# testX.head(5)
```

```
Y_train = pd.read_csv("./data/Y_train.csv")
Y = Y_train.to_numpy()
trainY = np.zeros(len(Y))
# change 0 in trainY to -1
for idx in range(len(Y)):
    if Y[idx] == 0 :
        trainY[idx] = -1
    else :
        trainY[idx] = 1
# print(trainY)
```

```
Y_test = pd.read_csv("./data/Y_test.csv")
Y = Y_test.to_numpy()
testY = np.zeros(len(Y))
# change 0 in trainY to -1
for idx in range(len(Y)):
    if Y[idx] == 0 :
        testY[idx] = -1
    else :
        testY[idx] = 1
# print(testY)
```

然后是我们 SVM 的实现，这里我参考了老师上课所讲解的例子进行实现：

```
class SVM:

    def init(self, learning_rate=0.001,lambda_param=0.02,n_iters=800):
        self.lr = learning_rate #a
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self,dataX,dataY):
        n_samples, n_features = dataX.shape

        self.w = np.zeros(n_features)
        self.b = 0

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(dataX):
                # print(x_i,dataY[idx])
                self.update(x_i,dataY[idx])

    def update(self,x,y):
        distance = 1-(y*(np.dot(x,self.w)+self.b))
        hinge_loss = max(0,distance)
        if(hinge_loss == 0):
            self.w = self.w-self.lr*(2*self.lambda_param*self.w)
        else:
            # print(x,"\n",y)
            self.w = self.w-self.lr*(2*self.lambda_param*self.w-np.dot(x,y))
            self.b = self.b+self.lr*y

    def predict(self,dataX):
        eq = np.dot(dataX,self.w)+self.b
        return np.sign(eq)
```

对于代码及参数给出如下解释：

1. Init 函数

Init 函数用于初始化我们的 SVM

```
def init(self, learning_rate=0.001, lambda_param=0.02, n_iters=800):
    self.lr = learning_rate #α
    self.lambda_param = lambda_param
    self.n_iters = n_iters
    self.w = None
    self.b = None
```

Learning rate 为我们上课所学到的学习率，同时，取损失函数如下：

$$L(w, w_0 \mid D) = \frac{1}{N}\sum_{\ell=1}^{N} \max(0, 1-y^{(\ell)}(w^{\mathrm{T}}x^{(\ell)}+w_0)) + \frac{\lambda}{2}||w||^2$$

Lambda_param 即为上式中的λ/2，n_iters 为迭代次数，当到达所设定的值时，停止训练。

2. Fit 函数

Fit 函数用于 SVM 的学习（训练）过程

```
def fit(self,dataX,dataY):
    n_samples, n_features = dataX.shape

    self.w = np.zeros(n_features)
    self.b = 0

    for _ in range(self.n_iters):
        for idx, x_i in enumerate(dataX):
            # print(x_i,dataY[idx])
            self.update(x_i,dataY[idx])
```

w，b 对应我们公式中的 w，b，初始值设置为 0。采用两层循环完成训练，第一层是我们的训练次数，第二层是我们对 data 库中吗每一组 x，y 进行训练。我们取出单独的每一条 x_i 以及对应的 y（dataY[idx]）传入 update 函数中更新 w，b 的值。

3. Update 函数

Update 函数用于每一次单独训练的细化

```
def update(self,x,y):
    distance = 1-(y*(np.dot(x,self.w)+self.b))
    hinge_loss = max(0,distance)
    if(hinge_loss == 0):
        self.w = self.w-self.lr*(2*self.lambda_param*self.w)
    else:
        # print(x,"\n",y)
        self.w = self.w-self.lr*(2*self.lambda_param*self.w-np.dot(x,y))
        self.b = self.b+self.lr*y
```

distance 作为中间值用于计算 hinge_loss 也就是损失函数值，并根据情况判断来更新 w，b 的值，计算如图：

$$L(\boldsymbol{w}, w_0 \mid D) = \begin{cases} \frac{\lambda}{2}\|\boldsymbol{w}\|^2 & \text{if } y^{(\ell)}(\boldsymbol{w}^{\mathrm{T}}x^{(\ell)}+w_0) \geq 1 \\ \frac{1}{N}\sum_{\ell=1}^{N} 1-y^{(\ell)}(\boldsymbol{w}^{\mathrm{T}}x^{(\ell)}+w_0) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 & \text{otherwise} \end{cases}$$

For each w_j ($j=0,\ldots,d$):

$$\frac{\partial L}{\partial w_j} = \begin{cases} \lambda w_j & \text{if } y^{(\ell)}(\boldsymbol{w}^{\mathrm{T}}x^{(\ell)}+w_0) \geq 1 \\ \lambda w_j - y^{(\ell)}x^{(\ell)} & \text{o.w.} \end{cases}$$

$$\frac{\partial L}{\partial w_0} = \begin{cases} 0 & \text{if } y^{(\ell)}(\boldsymbol{w}^{\mathrm{T}}x^{(\ell)}+w_0) \geq 1 \\ -y^{(\ell)} & \text{o.w.} \end{cases}$$

4. Predict 函数

Predict 函数用于对 test 集进行预测

```
def predict(self, dataX):
    eq = np.dot(dataX, self.w)+self.b
    return np.sign(eq)
```

传入测试数据 dataX，计算出结果并返回 sign

下面介绍我们的学习与测试实际运用过程：

```
clf = SVM()
clf.init()
start_time = time.time()
clf.fit(trainX.to_numpy(), trainY)
end_time = time.time()
predictedY = clf.predict(testX.to_numpy())
```

首先实例化 SVM 类，取名 clf（老师的例子中用这个名字我就直接用了），调用 init 函数用于初始化，随后调用 fit 函数开始训练（两头加上时间记录函数），最后使用 predict 函数进行预测

关于评测指标，我选择使用 sklearn.metrics 中的 accuracy_score 函数计算评估准确率（所有样本中有多少被准确预测了），使用 recall_score 函数计算评估召回率（正例样本中预测正确了多少），使用 precision_score 函数计算评估精确率（预测为正例的里面有多少是对的）。

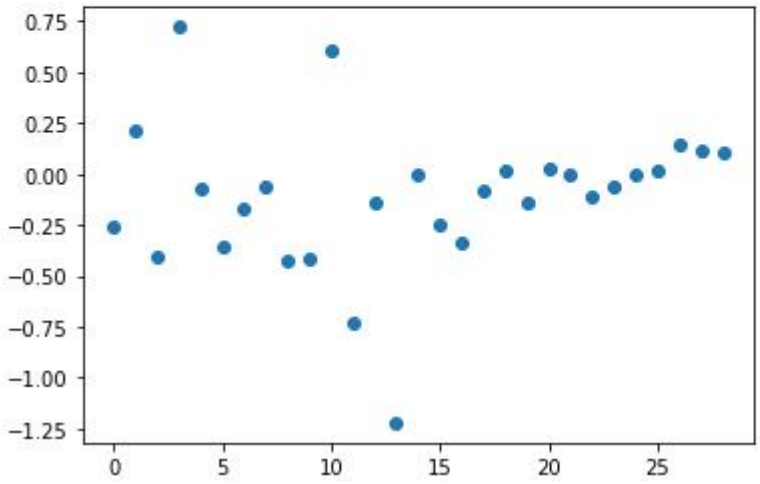
结果如图：

```
print("fit time: {}".format(end_time-start_time))
print("accuracy on test dataset: {}".format(accuracy_score(testY, predictedY)))
print("recall on test dataset: {}".format(recall_score(testY, predictedY)))
print("precision on test dataset: {}".format(precision_score(testY, predictedY)))
```

```
fit time: 9.501799583435059
accuracy on test dataset: 0.85
recall on test dataset: 0.6842105263157895
precision on test dataset: 1.0
```

以及打印的 w，b 的值

```
[-0.2597445   0.21244897 -0.40137655  0.72191565 -0.07285712 -0.36147921
 -0.17094976 -0.06159953 -0.43036999 -0.41979212  0.61096953 -0.73505614
 -0.14116231 -1.22081644 -0.00649399 -0.24472475 -0.34151514 -0.08139082
  0.01728005 -0.14164351  0.02179914 -0.00521123 -0.11056641 -0.06569335
 -0.00556229  0.01999888  0.14546662  0.11574942  0.10187698]
[2.681]
```



最后介绍一下训练过程中损失函数值以及准确性的变化过程
我在 800 次迭代中选取了几段:

```
[1.]
[0.99149305]
[0.98297463]
[1.00200577]
[1.01432795]
[0.99192574]
[1.01267306]
[0.98669245]
[0.9938513]
[0.97801688]
[1.01816671]
[0.97773072]
[1.02595921]
[0.97725711]
[1.02962993]
[1.00026009]
[0.98071613]
[0.97346075]
[0.99953641]
```

```
0
[0.25901797]
0
[0.15552333]
[0.42058538]
[0.25388708]
[0.02862858]
[1.27331052]
[0.17037741]
0
[0.38033023]
0
[0.28119152]
[0.33876728]
[0.15674523]
[0.22547301]
[0.14923493]
0
[1.60299272]
```

```
[0.06233148]
0
0
0
[0.39938025]
0
[0.05456596]
0
0
[1.8211025]
0
0
0
0
0
0
0
[0.69018907]
[0.01785431]
0
[0.03918289]
0
```

```
0
0
[0.00742764]
0
0
0
0
0
[0.00980172]
0
```

可以看到，一开始损失函数值在 1.0 附近，随后经过一部分迭代后开始出现 0，0 越来越多，最后，损失函数值稳定在 0 附近。
对于准确性等指标的变化，我在尝试调整迭代次数的过程中，有如下情况
分别为迭代 100、300、500、800、1000、2000

```
运行时间：0.48696351051330566秒
accuracy on test dataset: 0.82
```

```
运行时间：1.2589619159698486秒
accuracy on test dataset: 0.85
```

```
运行时间：2.2150168418884277秒
accuracy on test dataset: 0.865
```

```
运行时间：3.5002171993255615秒
accuracy on test dataset: 0.875
```

```
运行时间：4.339415788650513秒
accuracy on test dataset: 0.88
```

```
运行时间：8.564855813980103秒
accuracy on test dataset: 0.88
```

可以观察到，随着迭代次数的提高，运行时间是逐渐增长的，但是预测的准确率，在 100 到 1000 次的递增中，准确率由 0.82 提高到了 0.88；但是从 1000 次提高到 2000 次时准确率并无变化，可能是由于 w 与 b 的值在 1000 次迭代后已经处于收敛阶段，提高迭代次数对 w 和 b 的修改变化十分微小。
我也尝试了对学习率和λ的值作修改，但是对各评测值的影响并不大。
这里说明一下，如果对数据不做归一化直接进行训练，同样是 800 次迭代
结果如下

```
运行时间：3.0999901294708255秒
accuracy on test dataset: 0.76
recall on test dataset: 0.9578947368421052
precision on test dataset: 0.674074074074074
```

可以看到预测的准确率和精确率相较归一化后的结果是较差的，由此可知对数据的归一化可以很好地提高对数据利用的有效性，从而提高模型预测的准确率。

二、 SVM using sklearn

首先是载入数据部分，这里和前面差不多，不做过多赘述：

```
dataX1 = pd.read_csv('./data/X_train.csv')
dataY1 = pd.read_csv('./data/Y_train.csv')
dataXT1 = pd.read_csv('./data/X_test.csv')
dataYT1 = pd.read_csv('./data/Y_test.csv')
y_map={1:1,0:-1}
dataY1['Label']=dataY1['Label'].map(y_map)
dataYT1['Label']=dataYT1['Label'].map(y_map)
```

对于使用 sklearn 实现的 SVM，我选择使用 sklearn.svm 提供的 LinearSVC;
调用 LinearSVC 实例化线性分类支持向量机，使用 fit 函数，传入训练的数据集以实现对向量机的训练，用变量 start_time1,end_time1 分别记录开始时间和结束时间，如图：

```
linearsvc = LinearSVC(C=1e9)
start_time1=time.time()
linearsvc.fit(dataX1.to_numpy(), dataY1.to_numpy())
end_time1=time.time()
```

调用相关函数获取训练所得的 w 与 b

```
w=linearsvc.coef_
b=linearsvc.intercept_
```

使用 predict 函数对测试集的 x 数据进行预测，如图

```
Py=linearsvc.predict(dataXT1.to_numpy())
```

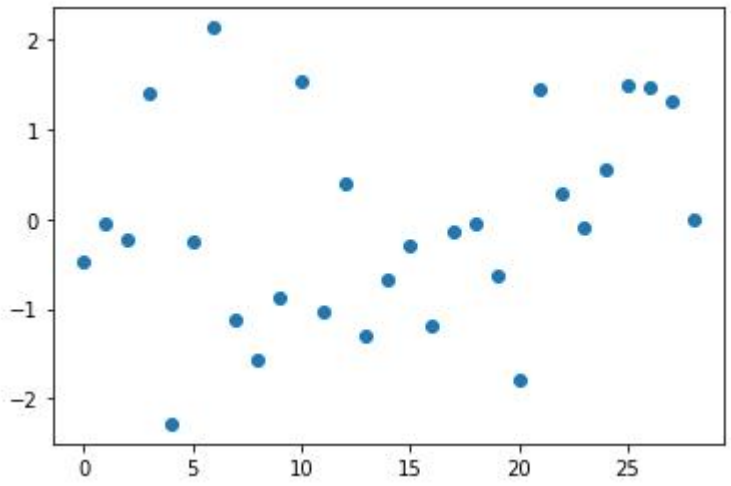
最后打印运行时间，上文提及的测量指标（准确率，召回率和精确率），如图

```
print("运行时间：{}秒".format(end_time1-start_time1))
print("accuracy on test dataset: {}".format(accuracy_score(dataYT1.to_numpy(), Py)))
print("recall on test dataset: {}".format(recall_score(dataYT1.to_numpy(), Py)))
print("precision on test dataset: {}".format(precision_score(dataYT1.to_numpy(), Py)))
```

```
运行时间：0.020160675048828125秒
accuracy on test dataset: 0.9
recall on test dataset: 0.9052631578947369
precision on test dataset: 0.8865979381443299
```

再打印 w，b，并将 w 以图形式呈现

```
[-0.48271553 -0.04895821 -0.23448403  1.40312742 -2.28014888 -0.24665846
  2.1296161  -1.11251603 -1.56759854 -0.88462836  1.52657857 -1.026641
  0.38421142 -1.29452693 -0.66913844 -0.30411337 -1.19752286 -0.14304955
 -0.05640576 -0.63428871 -1.78559425  1.45285473  0.28367293 -0.09869297
  0.54246576  1.49389257  1.45656251  1.30198715 -0.01016236]
[-3.55995828]
```



三、 二者比较

使用 sklearn 中的 LinearSVC

```
运行时间：0.020160675048828125秒
accuracy on test dataset: 0.9
recall on test dataset: 0.9052631578947369
precision on test dataset: 0.8865979381443299
```

使用自己实现的 SVM

```
运行时间：3.8320188522338867秒
accuracy on test dataset: 0.875
recall on test dataset: 0.7368421052631579
precision on test dataset: 1.0
```

可以观察到，在运行时间方面，LinearSVC 的耗时远远小于我实现的 SVM，预测的准确率和召回率也要更高。这可能是因为 LinearSVC 本身的设计就是为了支持大量数据的训练集，其运用的算法更加高效，所选用的损失函数也更加适用训练集的数据，从而可以更加迅速地得到收敛的 w 和 b，以减少迭代的次数，在更短的时间内得到高效的模型。而在精确率方面，我的模型更高，可能是由于更多次的迭代较大程度地剔除了正样本中的错误估计，同时也是召回率和精确率一般呈反比的体现。

四、 总结

在这次 lab 中，我很好地学习到了 SVM 的具体实现过程，对它也有了更深的理解。同时，我深深感慨到了 python 的方便之处，不管是各种各样的函数（比如说矩阵计算），还是各种各样的包装库（sklearn 本身）都对我们的学习有很大帮助，节省了我们学习工作中很多时间。以后也要多多学习 python 知识，让它更好地服务于机器学习课程中。