机器学习DAY05

支持向量机(SVM)

支持向量机原理

1. **寻求最优分类边界**

   正确：对大部分样本可以正确地划分类别。

   泛化：最大化持向量间距。

   公平：与支持向量等距。

   简单：线性，直线或平面，分割超平面。

2. **基于核函数的升维变换**

   通过名为核函数的特征变换，增加新的特征，使得低维度空间中的线性不可分问题变为高维度空间中的线性可分问题。

   线性核函数：linear，不通过核函数进行维度提升，仅在原始维度空间中寻求线性分类边界。

   基于线性核函数的SVM分类相关API：

```
model = svm.SVC(kernel='linear')
model.fit(train_x, train_y)
```

   案例：对simple2.txt中的数据进行分类。

```
import numpy as np
import sklearn.model_selection as ms
import sklearn.svm as svm
import sklearn.metrics as sm
import matplotlib.pyplot as mp
x, y = [], []
data = np.loadtxt('../data/multiple2.txt', delimiter=',', dtype='f8')
x = data[:, :-1]
y = data[:, -1]
train_x, test_x, train_y, test_y = \
    ms.train_test_split(x, y, test_size=0.25, random_state=5)
# 基于线性核函数的支持向量机分类器
model = svm.SVC(kernel='linear')
model.fit(train_x, train_y)
n = 500
l, r = x[:, 0].min() - 1, x[:, 0].max() + 1
b, t = x[:, 1].min() - 1, x[:, 1].max() + 1
grid_x = np.meshgrid(np.linspace(l, r, n),
                     np.linspace(b, t, n))
flat_x = np.column_stack((grid_x[0].ravel(), grid_x[1].ravel()))
flat_y = model.predict(flat_x)
grid_y = flat_y.reshape(grid_x[0].shape)
pred_test_y = model.predict(test_x)
cr = sm.classification_report(test_y, pred_test_y)
print(cr)
mp.figure('SVM Linear Classification', facecolor='lightgray')
mp.title('SVM Linear Classification', fontsize=20)
```

```
mp.xlabel('x', fontsize=14)
mp.ylabel('y', fontsize=14)
mp.tick_params(labelsize=10)
mp.pcolormesh(grid_x[0], grid_x[1], grid_y, cmap='gray')
mp.scatter(test_x[:, 0], test_x[:, 1], c=test_y, cmap='brg', s=80)
mp.show()
```

多项式核函数：poly，通过多项式函数增加原始样本特征的高次方幂

$$y = x_1 + x_2$$
$$y = x_1^2 + 2x_1x_2 + x_2^2$$
$$y = x_1^3 + 3x_1^2x_2 + 3x_1x_2^2 + x_2^3$$

案例，基于多项式核函数训练sample2.txt中的样本数据。

```
# 基于线性核函数的支持向量机分类器
model = svm.SVC(kernel='poly', degree=3)
model.fit(train_x, train_y)
```

径向基核函数：rbf，通过高斯分布函数增加原始样本特征的分布概率

案例，基于径向基核函数训练sample2.txt中的样本数据。

```
# 基于径向基核函数的支持向量机分类器
# C：正则强度
# gamma：如果gamma越大，标准差会很小，标准差很小的高斯分布长得又高又瘦， 会造成只会作用于支持向量样本附近，容易过拟合。
model = svm.SVC(kernel='rbf', C=600, gamma=0.01)
model.fit(train_x, train_y)
```

网格搜索

获取一个最优超参数的方式可以绘制验证曲线，但是验证曲线只能每次获取一个最优超参数。如果多个超参数有很多排列组合的话，就可以使用网格搜索寻求最优超参数组合。

针对超参数组合列表中的每一个超参数组合，实例化给定的模型，做cv次交叉验证，将其中平均f1得分最高的超参数组合作为最佳选择，实例化模型对象。

网格搜索相关API：

```
import sklearn.model_selection as ms
params = [{'kernel':['linear'], 'C':[1, 10, 100, 1000]},
    {'kernel':['poly'], 'C':[1], 'degree':[2, 3]},
    {'kernel':['rbf'], 'C':[1,10,100], 'gamma':[1, 0.1, 0.01]}]
model = ms.GridSearchCV(模型, params, cv=交叉验证次数)
model.fit(输入集，输出集)
# 获取网格搜索每个参数组合
model.cv_results_['params']
# 获取网格搜索每个参数组合所对应的平均测试分值
model.cv_results_['mean_test_score']
# 获取最好的参数
model.best_params_
model.best_score_
model.best_estimator_
```

案例：修改置信概率案例，基于网格搜索得到最优超参数。

```
# 基于径向基核函数的支持向量机分类器
params = [{'kernel':['linear'], 'C':[1, 10, 100, 1000]},
    {'kernel':['poly'], 'C':[1], 'degree':[2, 3]},
    {'kernel':['rbf'], 'C':[1,10,100,1000], 'gamma':[1, 0.1, 0.01, 0.001]}]
model = ms.GridSearchCV(svm.SVC(probability=True), params, cv=5)
model.fit(train_x, train_y)
for p, s in zip(model.cv_results_['params'],
        model.cv_results_['mean_test_score']):
    print(p, s)
# 获取得分最优的的超参数信息
print(model.best_params_)
# 获取最优得分
print(model.best_score_)
# 获取最优模型的信息
print(model.best_estimator_)
```

情感分析

文本情感分析又称意见挖掘、倾向性分析等。简单而言,是对带有情感色彩的主观性文本进行分析、处理、归纳和推理的过程。互联网产生了大量的诸如人物、事件、产品等有价值的评论信息。这些评论信息表达了人们的各种情感色彩和情感倾向性,如喜、怒、哀、乐和批评、赞扬等。基于此,潜在的用户就可以通过浏览这些主观色彩的评论来了解大众舆论对于某一事件或产品的看法。

```
预测酒店评论是好评还是差评：
1. 房间真棒，离大马路很近，非常方便。不错。                好评  0.9954
2. 房间有点脏，厕所还漏水，空调不制冷，下次再也不来了。    差评  0.99
3. 地板不太干净，电视没信号，但是空调还可以，总之还行。    好评  0.56

```

先针对训练文本进行分词处理，统计词频，通过词频-逆文档频率算法获得该词对样本语义的贡献，根据每个词的贡献力度，构建有监督分类学习模型。把测试样本交给模型处理，得到测试样本的情感类别。

```
pip install nltk -i https://pypi.tuna.tsinghua.edu.cn/simple/
```

文本分词

分词处理相关API：

```
import nltk.tokenize as tk
# 把样本按句子进行拆分  sent_list:句子列表
sent_list = tk.sent_tokenize(text)
# 把样本按单词进行拆分  word_list:单词列表
word_list = tk.word_tokenize(text)
#  把样本按单词进行拆分 punctTokenizer：分词器对象
punctTokenizer = tk.WordPunctTokenizer()
word_list = punctTokenizer.tokenize(text)
```

案例：

```
import nltk.tokenize as tk
doc = "Are you curious about tokenization? " \
      "Let's see how it works! " \
      "We need to analyze a couple of sentences " \
```

```
      "with punctuations to see it in action."
print(doc)
tokens = tk.sent_tokenize(doc)
for i, token in enumerate(tokens):
    print("%2d" % (i + 1), token)
print('-' * 15)
tokens = tk.word_tokenize(doc)
for i, token in enumerate(tokens):
    print("%2d" % (i + 1), token)
print('-' * 15)
tokenizer = tk.WordPunctTokenizer()
tokens = tokenizer.tokenize(doc)
for i, token in enumerate(tokens):
    print("%2d" % (i + 1), token)
```

词袋模型

一句话的语义很大程度取决于某个单词出现的次数，所以可以把句子中所有可能出现的单词作为特征名，每一个句子为一个样本，单词在句子中出现的次数为特征值构建数学模型，称为词袋模型。

This hotel is very bad. The toilet in this hotel smells bad. The environment of this hotel is very good.

1 This hotel is very bad.
2 The toilet in this hotel smells bad.
3 The environment of this hotel is very good.

This	hotel	is	very	bad	The	toilet	in	smells	environment	of	good
1	1	1	1	1	0	0	0	0	0	0	0
1	1	0	0	1	1	1	1	1	0	0	0
1	1	1	1	0	1	0	0	0	1	1	1

词袋模型化相关API：

```
import sklearn.feature_extraction.text as ft

# 构建词袋模型对象
cv = ft.CountVectorizer()
# 训练模型，把句子中所有可能出现的单词作为特征名，每一个句子为一个样本，单词在句子中出现的次数为特征值。
bow = cv.fit_transform(sentences).toarray()
print(bow)
# 获取所有特征名
words = cv.get_feature_names()
```

案例：

```
import nltk.tokenize as tk
import sklearn.feature_extraction.text as ft
doc = 'This hotel is very bad. The toilet in this hotel smells bad. The environment of this hotel is very good.'
print(doc)
sentences = tk.sent_tokenize(doc)
print(sentences)
cv = ft.CountVectorizer()
bow = cv.fit_transform(sentences).toarray()
print(bow)
words = cv.get_feature_names()
print(words)

```

词频（TF）

单词在句子中出现的次数除以句子的总词数称为词频。即一个单词在一个句子中出现的频率。词频相比单词的出现次数可以更加客观的评估单词对一句话的语义的贡献度。词频越高，对语义的贡献度越大。对词袋矩阵归一化即可得到词频。

案例：对词袋矩阵进行归一化

```
import nltk.tokenize as tk
import sklearn.feature_extraction.text as ft
import sklearn.preprocessing as sp
doc = 'This hotel is very bad. The toilet in this hotel smells bad. The environment of this hotel is very good.'
print(doc)
sentences = tk.sent_tokenize(doc)
print(sentences)
cv = ft.CountVectorizer()
bow = cv.fit_transform(sentences).toarray()
print(bow)
words = cv.get_feature_names()
print(words)
tf = sp.normalize(bow, norm='l1')
print(tf)
```

文档频率（DF）

$$DF = \frac{含有某个单词的文档样本数}{总文档样本数}$$

逆文档频率（IDF）

$$IDF = log(\frac{总样本数}{1+含有某个单词的样本数})$$

词频-逆文档频率(TF-IDF)

词频矩阵中的每一个元素乘以相应单词的逆文档频率，其值越大说明该词对样本语义的贡献越大，根据每个词的贡献力度，构建学习模型。

获取词频逆文档频率（TF-IDF）矩阵相关API：

```
# 获取词袋模型
cv = ft.CountVectorizer()
bow = cv.fit_transform(sentences).toarray()
# 获取TF-IDF模型训练器
tt = ft.TfidfTransformer()
tfidf = tt.fit_transform(bow).toarray()
```

案例：获取TF_IDF矩阵：

```
import nltk.tokenize as tk
import sklearn.feature_extraction.text as ft

doc = 'This hotel is very bad. The toilet in this hotel smells bad. The environment of this hotel is very good.'
print(doc)
sentences = tk.sent_tokenize(doc)
print(sentences)
cv = ft.CountVectorizer()
bow = cv.fit_transform(sentences).toarray()
print(bow)
words = cv.get_feature_names()
print(words)
tt = ft.TfidfTransformer()
tfidf = tt.fit_transform(bow).toarray()
print(tfidf)
```

文本分类(主题识别)

使用给定的文本数据集进行主题识别训练，自定义测试集测试模型准确性。

案例：

```
import sklearn.datasets as sd
import sklearn.feature_extraction.text as ft
import sklearn.naive_bayes as nb

train = sd.load_files('../data/20news', encoding='latin1',
    shuffle=True, random_state=7)
# 20news 下的文件夹名即是相应子文件的主题类别名
# train.data 返回每个文件的字符串内容
# train.target 返回每个文件的父目录名（主题类别名）
train_data = train.data
train_y = train.target
categories = train.target_names
cv = ft.CountVectorizer()
train_bow = cv.fit_transform(train_data)
tt = ft.TfidfTransformer()
train_x = tt.fit_transform(train_bow)
model = nb.MultinomialNB()
model.fit(train_x, train_y)
test_data = [
    'The curveballs of right handed pitchers tend to curve to the left',
    'Caesar cipher is an ancient form of encryption',
    'This two-wheeler is really good on slippery roads']
test_bow = cv.transform(test_data)
test_x = tt.transform(test_bow)
```

```
pred_test_y = model.predict(test_x)
for sentence, index in zip(test_data, pred_test_y):
    print(sentence, '->', categories[index])

```

朴素贝叶斯分类

朴素贝叶斯分类是一种依据统计概率理论而实现的一种分类方式。观察这组数据：

天气情况	穿衣风格	约女朋友	==>	心情
0（晴天）	0（休闲）	0（约了）	==>	0（高兴）
0	1（风骚）	1（没约）	==>	0
1（多云）	1	0	==>	0
0	2（破旧）	1	==>	1（郁闷）
2（下雨）	2	0	==>	0
…	…	…	==>	…
0	1	0	==>	？

通过上述训练样本如何预测：晴天、穿着休闲、没有约女朋友时的心情？可以整理相同特征值的样本，计算属于某类别的概率即可。但是如果在样本空间没有完全匹配的数据该如何预测？

1）晴天、穿着休闲、没有约女朋友的条件下高兴的概率

2）晴天、穿着休闲、没有约女朋友的条件下郁闷的概率

贝叶斯定理：P(A|B)=P(B|A)P(A)/P(B)　　<==　　*P(A, B) = P(A) P(B|A) = P(B) P(A|B)*

例如：

假设一个学校里有60%男生和4 0%女生.女生穿裤子的人数和穿裙子的人数相等,所有男生穿裤子.一个人在远处随机看到了一个穿裤子的学生.那么这个学生是女生的概率是多少?

```
P(女) = 0.4
P(裤子|女) = 0.5
P(裤子) = 0.6 + 0.2 = 0.8
P(女|裤子) = P(裤子|女) * P(女) / P(裤子) = 0.5 * 0.4 / 0.8 = 0.25

```

根据贝叶斯定理，如何预测：晴天、穿着休闲、没有约女朋友时的心情？

```
P(高兴|晴天,休闲,没约)
= P(晴天,休闲,没约|高兴) P(高兴) / P(晴天,休闲,没约)
P(郁闷|晴天,休闲,没约)
= P(晴天,休闲,没约|郁闷) P(郁闷) / P(晴天,休闲,没约)

即比较：
P(晴天,休闲,没约|高兴) P(高兴)
P(晴天,休闲,没约|郁闷) P(郁闷)

朴素：根据条件独立假设，特征值之间没有因果关系，则：
P(晴天|高兴) P(休闲|高兴) P(没约|高兴)P(高兴)
P(晴天|郁闷) P(休闲|郁闷) P(没约|郁闷)P(郁闷)
```

由此可得，统计总样本空间中晴天、穿着休闲、没有约女朋友时高兴的概率，与晴天、穿着休闲、没有约女朋友时不高兴的概率，择其大者为最终结果。

高斯贝叶斯分类器相关API：

```
import sklearn.naive_bayes as nb
# 创建高斯分布朴素贝叶斯分类器
model = nb.GaussianNB()
model = nb.MultinomialNB()
model.fit(x, y)
result = model.predict(samples)
```

案例：

```
import numpy as np
import sklearn.naive_bayes as nb
import matplotlib.pyplot as mp

data = np.loadtxt('../data/multiple1.txt', unpack=False, dtype='U20',
delimiter=',')
print(data.shape)
x = np.array(data[:, :-1], dtype=float)
y = np.array(data[:, -1], dtype=float)

# 创建高斯分布朴素贝叶斯分类器
model = nb.GaussianNB()
model.fit(x, y)
l, r = x[:, 0].min() - 1, x[:, 0].max() + 1
b, t = x[:, 1].min() - 1, x[:, 1].max() + 1
n = 500
grid_x, grid_y = np.meshgrid(np.linspace(l, r, n), np.linspace(b, t, n))
samples = np.column_stack((grid_x.ravel(), grid_y.ravel()))
grid_z = model.predict(samples)
grid_z = grid_z.reshape(grid_x.shape)

mp.figure('Naive Bayes Classification', facecolor='lightgray')
mp.title('Naive Bayes Classification', fontsize=20)
mp.xlabel('x', fontsize=14)
mp.ylabel('y', fontsize=14)
mp.tick_params(labelsize=10)
mp.pcolormesh(grid_x, grid_y, grid_z, cmap='gray')
mp.scatter(x[:, 0], x[:, 1], c=y, cmap='brg', s=80)
mp.show()
```