7.4.1 有监督学习算法网络

7.4.2 非监督学习算法网络

7.5 实战练习

请尝试使用神经网络方法对logit表单数据进行建模预测，并进行参数调优。

提示：参数调优操作请参见第8章相应内容。

对boston数据使用神经网络回归进行分析，先拆分为训练集和测试集，然后在其他参数固定不变的情况下，进行如下参数调整，观察结果变化。

将单隐含层的神经元数量设定为1~100。
将网络层数设定为1~20。
将连接函数设定为identity、logistic、tanh、relu。
将alpha设定为0.01~100。

8 支持向量机

8.1 支持向量机的基本原理

8.2 SVM分类

sklearn中的SVM分类方法：

SVC和NuSVC：是相似的方法，但参数设定不同，数学表达式也有差异。
LinearSVC：线性核函数的支持向量分类

class sklearn.svm.SVC(

```
C = 1.0 : float，错分案例的惩罚参数
    本质上是在错分样本和分界面的简单性之间进行权衡
    低的C值使分界面平滑，而高的C值则通过增加模型自由度给出更复杂的分界面

kernel = 'rbf' : 算法中使用的核函数
    'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable

degree = 3 : 多项式核函数时使用的阶次
gamma = 'auto' : 'rbf', 'poly'和'sigmoid'使用的核系数
    实际上定义了单个样本对模型的影响大小，值越小影响越大，值越大影响越小
    可以看作被模型选中作为支持向量的样本的影响半径的倒数
    'auto'时为1/n_features

coef0 = 0.0, shrinking = True

probability = False : 是否要求进行概率的估计，该选项会增加拟合时间

tol = 0.001, cache_size = 200, class_weight = None, verbose = False
max_iter = -1, decision_function_shape = 'ovr', random_state = None
```

)

sklearn.svm.SVC类的属性：

```
    support_ : array-like, shape = [n_SV], 支持向量的索引
    support_vectors_ : array-like, shape = [nSV, n_features], 支持向量
    n_support_ : array-like, dtype=int32, shape = [n_class], 每个类的sv数量
    dual_coef_ : array, shape = [n_class-1, n_SV], 决策函数中的支持向量系数
    coef_ : array, shape = [n_class-1, n_features], 各属性的权重/系数
    intercept_ : array, shape = [1,], 决策函数的常数项
```

In [ ]:

```
# 对自变量做标准化
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
irisZX = scaler.fit_transform(iris.data)
```

In [ ]:

```
from sklearn.preprocessing import binarize

tmpy = binarize(iris.target.reshape(-1, 1))
```

In [ ]:

```
from sklearn.svm import SVC

# 构建两分类预测模型
clf = SVC()
clf.fit(irisZX, tmpy)
```

In [ ]:

```
clf.score(irisZX, tmpy)
```

In [ ]:

```
clf.support_
```

In [ ]:

```
clf.support_vectors_[:5]
```

In [ ]:

```
clf.n_support_
```

In [ ]:

```
clf.dual_coef_
```

In [ ]:

```
# 构建多分类预测模型，sklearn为1 vs 1模式
clf = SVC()
clf.fit(irisZX, iris.target)
```

In [ ]:

```
clf.score(irisZX, iris.target)
```

In [ ]:

```
clf.support_
```

In [ ]:

```
clf.support_vectors_[:5]
```

In [ ]:

```
clf.n_support_
```

In [ ]:

```
clf.dual_coef_
```

In [ ]:

```
# 不同核函数的效果比较
clf = SVC(kernel = 'linear')
clf.fit(irisZX, iris.target)
clf.score(irisZX, iris.target)
```

In [ ]:

```
clf = SVC(kernel = 'poly')
clf.fit(irisZX, iris.target)
clf.score(irisZX, iris.target)
```

In [ ]:

```
clf = SVC(kernel = 'sigmoid')
clf.fit(irisZX, iris.target)
clf.score(irisZX, iris.target)
```

8.3 SVM回归

SVM回归在python中使用SVR、NuSVR和LinearSVR类实现，此时因变量应当为连续变量（浮点类型）

class sklearn.svm.SVR(

```
    kernel = 'rbf', degree = 3, gamma = 'auto', coef0 = 0.0
    tol = 0.001, C = 1.0, epsilon = 0.1, shrinking = True
    cache_size = 200, verbose = False, max_iter = -1
```

)

In [ ]:

```
# 直接使用原始数据建模
from sklearn.svm import SVR

clf = SVR()
clf.fit(boston.data, boston.target)
```

In [ ]:

```
clf.score(boston.data, boston.target)
```

In [ ]:

```
# 对自变量做标准化
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
bostonZX = scaler.fit_transform(boston.data)
```

In [ ]:

```
from sklearn.svm import SVR

clf = SVR()
clf.fit(bostonZX, boston.target)
```

In [ ]:

```
clf.score(bostonZX, boston.target)
```

In [ ]:

```
clf2 = SVR(kernel = 'poly', degree = 3)
clf2.fit(bostonZX, boston.target)
clf2.score(bostonZX, boston.target)
```

In [ ]:

```
clf2 = SVR(kernel = 'linear')
clf2.fit(bostonZX, boston.target)
clf2.score(bostonZX, boston.target)
```

In [ ]:

```
clf2 = SVR(kernel = 'sigmoid')
clf2.fit(bostonZX, boston.target)
clf2.score(bostonZX, boston.target)
```

8.4 新奇值（novelty）检测

class sklearn.svm.OneClassSVM(

```
    kernel = 'rbf' : 算法中使用的核函数
        'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable

    degree = 3 : 多项式核函数时使用的阶次
    gamma = 'auto' : 'rbf', 'poly'和'sigmoid'使用的核系数
        'auto'时为1/n_features

    nu = 0.5 : 训练样本集最后允许被划分为负类（新奇值）的比例

    coef0 = 0.0, tol = 0.001, shrinking = True
    cache_size = 200, verbose = False, max_iter = -1, random_state = None
```

)

sklearn.svm.OneClassSVM类的属性：

```
    support_ : array-like, shape = [n_SV], 支持向量的索引
    support_vectors_ : array-like, shape = [nSV, n_features], 支持向量
    dual_coef_ : array, shape = [1, n_SV], 决策函数中的支持向量系数
    coef_ : array, shape = [1, n_features], 各属性的权重/系数
    intercept_ : array, shape = [1,], 决策函数的常数项
```

In [ ]:

```
import numpy as np
from sklearn import svm

xx, yy = np.meshgrid(np.linspace(-5, 5, 500), np.linspace(-5, 5, 500))
# Generate train data
X = 0.3 * np.random.randn(100, 2)
X_train = np.r_[X + 2, X - 2]
# Generate some regular novel observations
X = 0.3 * np.random.randn(20, 2)
X_test = np.r_[X + 2, X - 2]
# Generate some abnormal novel observations
X_outliers = np.random.uniform(low=-4, high=4, size=(20, 2))
```

In [ ]:

```
# 数据分布示意图
plt.scatter(X_train[:,0], X_train[:,1])
plt.scatter(X_test[:,0], X_test[:,1])
plt.scatter(X_outliers[:,0], X_outliers[:,1])
```

In [ ]:

```
# 使用默认模型参数拟合一类SVM模型
clf = svm.OneClassSVM()
clf.fit(X_train)
```

In [ ]:

```
len(clf.support_vectors_), clf.support_vectors_
```

In [ ]:

```
clf.predict(X_train)
```

In [ ]:

```
pd.DataFrame(clf.predict(X_train))[0].value_counts()
```

In [ ]:

```
clf.predict(X_test)
```

In [ ]:

```
pd.DataFrame(clf.predict(X_test))[0].value_counts()
```

In [ ]:

```
# 调整模型参数
clf = svm.OneClassSVM(nu = 0.01)
clf.fit_predict(X_train)
```

In [ ]:

```
clf.predict(X_test)
```

In [ ]:

```
clf.predict(X_outliers)
```

In [ ]:

```
# 调整模型参数
clf = svm.OneClassSVM(nu = 0.1)
clf.fit(X_train)
clf.predict(X_train)
```

In [ ]:

```
clf.predict(X_test)
```

In [ ]:

```
clf.predict(X_outliers)
```

8.5 模型参数的优化

class sklearn.model_selection.GridSearchCV(

estimator : 考虑优化的估计器
param_grid : dict or list of dictionaries, 希望进行搜索的参数阵
scoring = None : string/callable/list/tuple/dict/None, 模型评分方法

fit_params = None, n_jobs = 1, cv = None
iid = True : 数据是否在各fold间均匀分布，此时将直接最小化总样本的损失函数
refit = True : 是否使用发现的最佳参数重新拟合估计器
verbose = 0, pre_dispatch = '2*n_jobs', error_score = 'raise'
return_train_score = True : 是否返回训练集的评分

)

GridSearchCV类的属性：

cv_results_ : 字典格式的参数列表，可被直接转换为pandas数据框
best_estimator_ : 网格搜索得出的最佳模型
best_score_ : 最佳模型的平均交互验证得分
best_params_ : dict，最佳模型的参数设定
best_index_ : int，最佳模型对应的索引值
scorer_ : function or a dict，用于选择最佳模型的评分函数
n_splits_ : int，交叉验证的拆分数

GridSearchCV类的方法：

decision_function(*args, **kwargs) : 调用筛选出的最佳模型并返回预测结果
其余标准API接口函数

In [ ]:

```
from sklearn import svm, datasets
from sklearn.model_selection import GridSearchCV

parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
svc = svm.SVC(probability = True)

clf = GridSearchCV(svc, parameters)
clf.fit(iris.data, iris.target)
```

In [ ]:

```
# 显示所有拟合模型的参数设定
pd.DataFrame(clf.cv_results_)
```

In [ ]:

```
clf.best_estimator_
```

In [ ]:

```
clf.decision_function(iris.data[:10])
```

In [ ]:

```
clf.predict_proba(iris.data[:10])
```

In [ ]:

```
clf.best_estimator_.predict_proba(iris.data[:10])
```

8.6 实战练习

如果在模型训练完毕后，在数据中删除所有非支持向量，然后再重新建模，得到的模型结果是否会和原来完全一样？请用具体数据尝试一下，并思考原因。

请尝试对boston数据拟合SVM回归模型，并进行参数调优，找到最优模型。

提示：需要将数据拆分为训练集和验证集进行结果验证。

9 主成分分析与因子分析

9.1 主成分分析

9.1.1 主成分分析的基本原理

9.1.2 主成分分析的statsmodels实现

class statsmodels.multivariate.pca.PCA(

```
    data
    ncomp = None : 希望返回的主成分数，为None时全部返回

    standardize = True : 是否对数据做标准化，等价于用相关系数阵做PCA
    demean = True : 是否移除均数，standardize = True时该参数无效
        只使用该参数相当于基于协方差阵提取主成分
    normalize = True : 是否对提取出的主成分做标化

    gls = False : 是否使用两步GLS估计
    weights = None : 各变量在计算中的权重
    method = 'svd' : 具体使用的主成分提取方法
        'svd' 普通的SVD
        'eig' eigenvalue decomposition
        'nipals' 要提取的特征根数远少于变量数时，速度要快于SVD法

    missing = None : 对缺失值的处理方式
       {'drop-row'/'drop-col'/'drop-min' (删除行/列中较少的) /'fill-em'}
    tol = 5e-08, max_iter = 1000, tol_em = 5e-08, max_em_iter = 100
```

)

属性：