

13.1 降维技术

13.1.1 为何要对数据进行简化?

对数据进行简化还有如下一系列的原因:

- 使得数据集更易使用;
- 降低很多算法的计算开销;
- 去除噪声;
- 使得结果易懂.

13.1.2 降维技术的适用范围

- 在**已标注与未标注**的数据上都有降维技术.
- 这里我们将主要关注未标注数据上的降维技术, 将技术同样也可以应用于已标注的数据

13.1.3 三种主要降维技术简介

一 主成分分析 (Principal Component Analysis , PCA)

在PCA中, 数据从原来的坐标系转换到了新的坐标系, 新坐标系的选择是由数据本身决定的.

- 第一个新坐标轴选择的是原始数据中方差最大的方向,
- 第二个新坐标轴的选择和第一个坐标轴正交且具有最大方差的方向.
- 该过程一直重复, 重复次数为原始数据中特征的数目.
- 我们会发现, 大部分方差都包含在最前面的几个新坐标轴中. 因此, 我们可以忽略余下的坐标轴, 即对数据进行了降维处理.

二 因子分析 (Factor Analysis)

在因子分析中, 我们假设在观察数据的生成中有一些**观察不到的隐变量 (latent variable)**. 假设观察数据是这些隐变量和某些噪声的线性组合**. 那么隐变量的数据可能比观察数据的数目少, 也就是说通过找到隐变量就可以实现数据的降维. 因子分析已经应用于社会科学、金融和其他领域中了.

三 独立成分分析 (Independent Component Analysis, ICA)

ICA假设数据是从N个数据源生成的, 这一点和因子分析有些类似. 假设数据为多个数据源的混合观察结果, 这些数据源之间在统计上是**相互独立**的, 而在**PCA中只假设数据是不相关的**. 同因子分析一样, 如果数据源的数目少于观察数据的数目, 则可以实现降维过程.

13.2 PCA

主成分分析(Principal Component Analysis, PCA) 通俗理解: 就是找出一个最主要的特征, 然后进行分析.

13.2.1 PCA原理

1. 找出第一个主成分的方向, 也就是数据**方差最大**的方向。

2. 找出第二个主成分的方向，也就是数据**方差次大**的方向，并且该方向与第一个主成分方向**正交** (orthogonal 如果是二维空间就叫垂直)
3. 通过这种方式计算出所有的主成分方向。
4. 通过数据集的协方差矩阵及其特征值分析，我们就可以得到这些主成分的值。
5. 一旦得到了协方差矩阵的特征值和特征向量，我们就可以保留最大的 N 个特征。这些特征向量也给出了 N 个最重要特征的真实结构，我们就可以通过将数据乘上这 N 个特征向量 从而将它转换到新的空间上。

PCA的优缺点:

- 优点：降低数据的复杂性，识别最重要的多个特征。
- 缺点：不一定需要，且可能损失有用信息。
- 适用数据类型：数值型数据。

13.2.2 在NumPy中实现PCA

讲数据转换成前N个主成分的伪代码大致如下:

- 去除平均值
- 计算协方差矩阵
- 计算协方差矩阵的特征值和特征向量
- 将特征值从大到小排序
- 保留最上面的N个特征向量
- 将数据转换到上述N个特征向量构建的新空间中

具体代码如下:

```
import numpy as np

def loadDataSet(fileName, delim = '\t'):
    fr = open(fileName)
    stringArr = [line.strip().split(delim) for line in fr.readlines()]
    #map(a,b)表示每次调用a函数来处理b
    dataArr = [list(map(float, line)) for line in stringArr]
    return np.mat(dataArr)

def pca(dataMat, topNfeat=99999999):
    """
    函数说明：主成分分析
    :param dataMat: 原数据集
    :param topNfeat: 应用的N个特征
    :return: lowDataMat 降维后的数据集
            reconMat 新的数据集空间
    """
    #计算每一列的均值
    meanVals = np.mean(dataMat, axis=0)
    meanRemoved = dataMat - meanVals
    #协方差矩阵
    covMat = np.cov(meanRemoved, rowvar=0)
    #计算特征值和特征向量
    #eigVals为特征值， eigVects为特征向量
    eigVals, eigVects = np.linalg.eig(np.mat(covMat))
    # 对特征值，进行从小到大的排序，返回从小到大的index序号
```

```

eigValInd = np.argsort(eigVals)
#-1表示倒序，返回topN的特征值[最后一个 到 -(topNfeat+1) 但是不包括-(topNfeat+1)本身
的倒叙]
eigValInd = eigValInd[-(topNfeat+1):-1]
#重组eigVects，从大到小
redEigVects = eigVects[:,eigValInd]
#将数据转换到新空间
lowDataMat = meanRemoved * redEigVects
#不怎么明白了
reconMat = (lowDataMat * redEigVects.T) + meanVals
return lowDataMat, reconMat

if __name__ == '__main__':
    dataMat = loadDataSet('testSet.txt')
    lowDataMat, reconMat = pca(dataMat, 1)
    print(lowDataMat)
    print(reconMat)

```

结果如下:

```

PCA_HeatonHsu x
[ 4.94030110e+00]
[-1.37359603e+00]
[-5.01662249e-01]
[-5.89871235e-02]
[-1.89787138e-01]]
[[10.37044569 11.23955536]
[10.55719313 11.54594665]
[ 9.01323877 9.01282393]
...
[ 9.32502753 9.52436704]
[ 9.0946364 9.14637075]
[ 9.16271152 9.2580597 ]]

Process finished with exit code 0

```

13.2.3 对半导体数据进行降维处理

半导体是在一些极为先进的工厂中制造出来的。设备的生命早期有限，并且花费极其巨大。虽然通过早期测试和频繁测试来发现有瑕疵的产品，但仍有一些存在瑕疵的产品通过测试。如果我们通过机器学习技术用于发现瑕疵产品，那么它就会为制造商节省大量的资金。

具体来讲，它拥有590个特征。我们看看能否对这些特征进行降维处理。

对于数据的缺失值的问题，我们有一些处理方法(参考第5章)

目前该章节处理的方案是：将缺失值NaN(Not a Number缩写)，全部用平均值来替代(如果用0来处理的策略就太差劲了)。

完整代码:

```
import numpy as np
```

```

def loadDataSet(fileName, delim = '\t'):
    fr = open(fileName)
    stringArr = [line.strip().split(delim) for line in fr.readlines()]
    #map(a,b)表示每次调用a函数来处理b
    dataArr = [list(map(float, line)) for line in stringArr]
    return np.mat(dataArr)

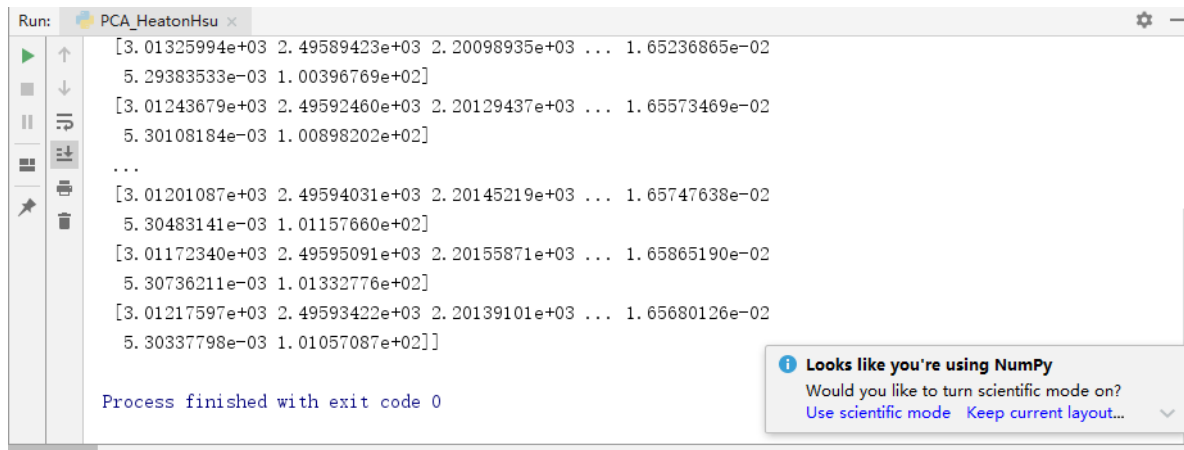
def pca(dataMat, topNfeat=99999999):
    """
    函数说明：主成分分析
    :param dataMat: 原数据集
    :param topNfeat: 应用的N个特征
    :return: lowDataMat 降维后的数据集
            reconMat 新的数据集空间
    """
    #计算每一列的均值
    meanVals = np.mean(dataMat, axis=0)
    meanRemoved = dataMat - meanVals
    #协方差矩阵
    covMat = np.cov(meanRemoved, rowvar=0)
    #计算特征值和特征向量
    #eigVals为特征值, eigVects为特征向量
    eigVals, eigVects = np.linalg.eig(np.mat(covMat))
    # 对特征值, 进行从小到大的排序, 返回从小到大的index序号
    eigvalInd = np.argsort(eigVals)
    #-1表示倒序, 返回topN的特征值[最后一个 到 -(topNfeat+1) 但是不包括-(topNfeat+1)本身
    的倒叙]
    eigvalInd = eigvalInd[::-1-(topNfeat+1)]
    #重组eigVects, 从大到小
    redEigVects = eigVects[:, eigvalInd]
    #将数据转换到新空间
    lowDataMat = meanRemoved * redEigVects
    #不怎么明白了
    reconMat = (lowDataMat * redEigVects.T) + meanVals
    return lowDataMat, reconMat

def replaceNaNwithMean():
    """
    :return: dataMat: 处理好的dataMat
    """
    dataMat = loadDataSet('secom.data', ' ')
    #特征数量
    numFeat = np.shape(dataMat)[1]
    for i in range(numFeat):
        #对value部位NaN的求均值
        #.A表示将矩阵转换成对应的数组(array)
        meanVal = np.mean(dataMat[np.nonzero(~np.isnan(dataMat[:, i].A))[0], i])
        #将value为NaN的值赋值为均值
        dataMat[np.nonzero(np.isnan(dataMat[:, i].A))[0], i] = meanVal
    return dataMat

if __name__ == '__main__':
    dataMat = replaceNaNwithMean()
    lowDataMat, reconMat = pca(dataMat, 1)
    print(lowDataMat)

```

```
print(reconMat)
```

A screenshot of a Jupyter Notebook interface. The top bar shows 'Run: PCA_HeatonHsu'. The main area displays the output of a print statement, showing a 5x5 matrix of scientific notation values. The values are: Row 1: [3.01325994e+03, 2.49589423e+03, 2.20098935e+03, ..., 1.65236865e-02]; Row 2: [5.29383533e-03, 1.00396769e+02]; Row 3: [3.01243679e+03, 2.49592460e+03, 2.20129437e+03, ..., 1.65573469e-02]; Row 4: [5.30108184e-03, 1.00898202e+02]; Row 5: [3.01201087e+03, 2.49594031e+03, 2.20145219e+03, ..., 1.65747638e-02]; Row 6: [5.30483141e-03, 1.01157660e+02]; Row 7: [3.01172340e+03, 2.49595091e+03, 2.20155871e+03, ..., 1.65865190e-02]; Row 8: [5.30736211e-03, 1.01332776e+02]; Row 9: [3.01217597e+03, 2.49593422e+03, 2.20139101e+03, ..., 1.65680126e-02]; Row 10: [5.30337798e-03, 1.01057087e+02]]. The bottom status bar says 'Process finished with exit code 0'. A notification bubble on the right says 'Looks like you're using NumPy. Would you like to turn scientific mode on? Use scientific mode Keep current layout...'.

```
[3.01325994e+03 2.49589423e+03 2.20098935e+03 ... 1.65236865e-02
 5.29383533e-03 1.00396769e+02]
[3.01243679e+03 2.49592460e+03 2.20129437e+03 ... 1.65573469e-02
 5.30108184e-03 1.00898202e+02]
...
[3.01201087e+03 2.49594031e+03 2.20145219e+03 ... 1.65747638e-02
 5.30483141e-03 1.01157660e+02]
[3.01172340e+03 2.49595091e+03 2.20155871e+03 ... 1.65865190e-02
 5.30736211e-03 1.01332776e+02]
[3.01217597e+03 2.49593422e+03 2.20139101e+03 ... 1.65680126e-02
 5.30337798e-03 1.01057087e+02]]

Process finished with exit code 0
```

13.3 sklearn中的PCA库

13.3.1 sklearn中PCA介绍

在sklearn中, 与PCA相关的类都在`sklearn.decomposition`包中, 其中, 最常用的PCA类就是`sklearn.decomposition.PCA`。

除了PCA类以外, 还有以下几种常用的PCA类, 具体如下:

- **KernelPCA类:** 除了PCA类以外, 最常用的PCA相关类还有KernelPCA类, 它主要用于非线性数据的降维, 需要用到核技巧。因此在使用的时候需要选择合适的核函数并对核函数的参数进行调参。
- **IncrementalPCA类:** 另外一个常用的PCA相关类是IncrementalPCA类, 它主要是为了解决单机内存限制的。有时候样本量可能是百万+, 维度可能也是上千, 直接去拟合数据可能会让内存爆掉, 此时我们可以用IncrementalPCA类来解决这个问题。IncrementalPCA先将数据分成多个batch, 然后对每个batch依次递增调用`partial_fit`函数, 这样一步步的得到最终的样本最优降维。
- **SparsePCA和MiniBatchSparsePCA类:** 他们和上面讲到的PCA类的区别主要是使用了**L1的正则化**, 这样可以将很多非主要成分的影响度降为0, 这样在PCA降维的时候我们仅仅需要对那些相对比较主要的成分进行PCA降维, 避免了一些噪声之类的因素对我们PCA降维的影响。

SparsePCA和MiniBatchSparsePCA之间的区别则是MiniBatchSparsePCA通过使用一部分样本特征和给定的迭代次数来进行PCA降维, 以解决在大样本时特征分解过慢的问题, 当然, 代价就是PCA降维的精确度可能会降低。使用SparsePCA和MiniBatchSparsePCA需要对L1正则化参数进行调参。

sklearn.decomposition: Matrix Decomposition

The `sklearn.decomposition` module includes matrix decomposition algorithms, including among others PCA, NMF or ICA. Most of the algorithms of this module can be regarded as dimensionality reduction techniques.

« **User guide:** See the [Decomposing signals in components \(matrix factorization problems\)](#) section for further details.

<code>decomposition.DictionaryLearning</code> ([...])	Dictionary learning
<code>decomposition.FactorAnalysis</code> ([n_components, ...])	Factor Analysis (FA)
<code>decomposition.FastICA</code> ([n_components, ...])	FastICA: a fast algorithm for Independent Component Analysis.
<code>decomposition.IncrementalPCA</code> ([n_components, ...])	Incremental principal components analysis (IPCA).
<code>decomposition.KernelPCA</code> ([n_components, ...])	Kernel Principal component analysis (KPCA)
<code>decomposition.LatentDirichletAllocation</code> ([...])	Latent Dirichlet Allocation with online variational Bayes algorithm
<code>decomposition.MiniBatchDictionaryLearning</code> ([...])	Mini-batch dictionary learning
<code>decomposition.MiniBatchSparsePCA</code> ([...])	Mini-batch Sparse Principal Components Analysis
<code>decomposition.NMF</code> ([n_components, init, ...])	Non-Negative Matrix Factorization (NMF)
<code>decomposition.PCA</code> ([n_components, copy, ...])	Principal component analysis (PCA)
<code>decomposition.SparsePCA</code> ([n_components, ...])	Sparse Principal Components Analysis (SparsePCA)
<code>decomposition.SparseCoder</code> (dictionary[, ...])	Sparse coding
<code>decomposition.TruncatedSVD</code> ([n_components, ...])	Dimensionality reduction using truncated SVD (aka LSA).
<code>decomposition.dict_learning</code> (X, n_components, ...)	Solves a dictionary learning matrix factorization problem.
<code>decomposition.dict_learning_online</code> (X[, ...])	Solves a dictionary learning matrix factorization problem online.
<code>decomposition.fastica</code> (X[, n_components, ...])	Perform Fast Independent Component Analysis.
<code>decomposition.non_negative_factorization</code> (X)	Compute Non-negative Matrix Factorization (NMF)
<code>decomposition.sparse_encode</code> (X, dictionary[, ...])	Sparse coding

13.3.2 sklearn.decomposition.PCA

sklearn.decomposition.PCA

```
class sklearn.decomposition.PCA(n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0,
                                iterated_power='auto', random_state=None) \[source\]
```

Principal component analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD.

It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko et al. 2009, depending on the shape of the input data and the number of components to extract.

It can also use the scipy.sparse.linalg ARPACK implementation of the truncated SVD.

Notice that this class does not support sparse input. See [TruncatedSVD](#) for an alternative with sparse data.

Read more in the [User Guide](#).

Parameters: **n_components** : *int, float, None or string*

Number of components to keep. if n_components is not set all components are kept:

```
n_components == min(n_samples, n_features)
```

If `n_components == 'mle'` and `svd_solver == 'full'`, Minka's MLE is used to guess the dimension. Use of `n_components == 'mle'` will interpret `svd_solver == 'auto'` as `svd_solver == 'full'`.

If `0 < n_components < 1` and `svd_solver == 'full'`, select the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by n_components.

If `svd_solver == 'arpack'`, the number of components must be strictly less than the minimum of n_features and n_samples.

Hence, the None case results in:

```
n_components == min(n_samples, n_features) - 1
```

copy : *bool (default True)*

If False, data passed to fit are overwritten and running `fit(X).transform(X)` will not yield the expected results, use `fit_transform(X)` instead.

whiten : *bool, optional (default False)*

When True (False by default) the `components_` vectors are multiplied by the square root of n_samples and then divided by the singular values to ensure uncorrelated outputs with unit component-wise variances.

Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making their data respect some hard-wired assumptions.

svd_solver : *string {'auto', 'full', 'arpack', 'randomized'}*

auto :

the solver is selected by a default policy based on `X.shape` and `n_components`: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient 'randomized' method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

full :

run exact full SVD calling the standard LAPACK solver via `scipy.linalg.svd` and select the components by postprocessing

arpack :

run SVD truncated to n_components calling ARPACK solver via `scipy.sparse.linalg.svds`. It requires strictly `0 < n_components < min(X.shape)`

randomized :

run randomized SVD by the method of Halko et al.

New in version 0.18.0.


```

tol : float >= 0, optional (default .0)
    Tolerance for singular values computed by svd_solver == 'arpack'.

    New in version 0.18.0.

iterated_power : int >= 0, or 'auto', (default 'auto')
    Number of iterations for the power method computed by svd_solver == 'randomized'.

    New in version 0.18.0.

random_state : int, RandomState instance or None, optional (default None)
    If int, random_state is the seed used by the random number generator; If RandomState instance,
    random_state is the random number generator; If None, the random number generator is the
    RandomState instance used by np.random. Used when svd_solver == 'arpack' or
    'randomized'.

    New in version 0.18.0.

```

参数介绍:

- **n_components:** 指定希望PCA降维后的特征维度数目, int, float, None or string 类型. 如果未设置n_components, 则保留所有特征: `n_components == min(n_samples, n_features)`
 - 如果 `n_components == 'mle'` 并且 `svd_solver == 'full'`, 则使用Minka'MLE来估计维度. 使用 `n_components == 'mle'` 会将 `svd_solver == 'auto'` 解释为 `svd_solver == 'full'`.
 - 如果 `0 < n_components < 1` 并且 `svd_solver == 'full'`, 则选择特征维度数, 以使需要解释的方差量大于n_components指定的百分比。
 - 如果 `svd_solver == 'arpack'`, 则特征维度的数量必须严格小于n_features和n_samples的最小值。
 - 因此, 取 None 将是: `n_components == min(n_samples, n_features) - 1`

- **copy:** 表示是否在运行算法时, 将原始数据复制一份, bool类型, 默认为True.

如果为False, 则传递给 fit 的数据将被覆盖, 并且运行 `fit(X).transform(X)` 将不会产生预期的结果, 请改用`fit_transform (X)`。

- **whiten :** 判断是否进行白化. bool类型, 可选, 默认为False

所谓白化, 就是对降维后的数据的每个特征进行归一化, 让方差都为1.对于PCA降维本身来说, 一般不需要白化。如果你PCA降维后有后续的数据处理动作, 可以考虑白化。默认值是False, 即不进行白化。

- **svd_solver:** 指定奇异值分解SVD的方法, 可以选择的有: {'auto', 'full', 'arpack', 'randomized'}, 默认为 'auto'.
 - 'auto': 即PCA类会自己去在前面讲到的三种算法里面去权衡, 选择一个合适的SVD算法来降维。一般来说, 使用默认值就够。
 - 'randomized': 一般适用于数据量大, 数据维度多同时主成分数目比例又较低的PCA降维, 它使用了一些加快SVD的随机算法。
 - 'full': 则是传统意义上的SVD, 使用了scipy库对应的实现。
 - 'arpack' 和randomized的适用场景类似, 区别是randomized使用的是scikit-learn自己的SVD实现, 而arpack直接使用了scipy库的sparse SVD实现。当svd_solver设置为'arpack'时, 保留的成分必须少于特征数, 即不能保留所有成分。

- **tol:** svd_solver == 'arpack'计算的奇异值的公差, float类型, 且大于等于0, 可选, 默认为0

- **iterated_power:** 迭代次数, int类型, 且大于等于0, 或者'auto', 默认为'auto'

- **random_state:** 随机状态, int或者 随机状态常数, 或者 None. 可选, 默认为None

如果为int, 则random_state是随机数生成器使用的种子; 否则为false。如果是RandomState实例, 则random_state是随机数生成器; 如果为None, 则随机数生成器是np.random使用的RandomState实例。在svd_solver == 'arpack'或'randomized'时使用。

当然, 还有一些**方法**:

Methods

<code>fit (self, X[, y])</code>	Fit the model with X.
<code>fit_transform (self, X[, y])</code>	Fit the model with X and apply the dimensionality reduction on X.
<code>get_covariance (self)</code>	Compute data covariance with the generative model.
<code>get_params (self[, deep])</code>	Get parameters for this estimator.
<code>get_precision (self)</code>	Compute data precision matrix with the generative model.
<code>inverse_transform (self, X)</code>	Transform data back to its original space.
<code>score (self, X[, y])</code>	Return the average log-likelihood of all samples.
<code>score_samples (self, X)</code>	Return the log-likelihood of each sample.
<code>set_params (self, **params)</code>	Set the parameters of this estimator.
<code>transform (self, X)</code>	Apply dimensionality reduction to X.

官方API中的例子:**

```
>>> import numpy as np
>>> from sklearn.decomposition import PCA
>>> X = np.array([[ -1, -1], [ -2, -1], [ -3, -2], [ 1, 1], [ 2, 1], [ 3, 2]])
>>> pca = PCA(n_components=2)
>>> pca.fit(X)
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
     svd_solver='auto', tol=0.0, whiten=False)
>>> print(pca.explained_variance_ratio_)
#可以看到, 第一个占比较高, 后一个几乎很少的占比
[0.9924... 0.0075...]
>>> print(pca.singular_values_)
[6.30061... 0.54980...]
```

```
>>> pca = PCA(n_components=2, svd_solver='full')
>>> pca.fit(X)
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
     svd_solver='full', tol=0.0, whiten=False)
>>> print(pca.explained_variance_ratio_)
[0.9924... 0.00755...]
>>> print(pca.singular_values_)
[6.30061... 0.54980...]
```

```
>>> pca = PCA(n_components=1, svd_solver='arpack')
>>> pca.fit(X)
PCA(copy=True, iterated_power='auto', n_components=1, random_state=None,
     svd_solver='arpack', tol=0.0, whiten=False)
>>> print(pca.explained_variance_ratio_)
[0.99244...]
>>> print(pca.singular_values_)
[6.30061...]
```