lab_gene_partial

May 22, 2019

1 Lab: PCA, LDA and Logistic Regression for Gene Expression Data

In this lab, we use logistic regression to predict biological characteristics ("phenotypes") from gene expression data. In addition to the concepts in breast cancer demo, you will learn to:

- Handle missing data
- Compute and visualize PCA and LDA coefficients
- Combine PCA and LDA with scaling
- Perform multi-class logistic classification on PCA and LDA outputs.
- Evaluate multi-class logistic classification with K-fold validation

1.1 Background

Genes are the basic unit in the DNA and encode blueprints for proteins. When proteins are synthesized from a gene, the gene is said to "express". Micro-arrays are devices that measure the expression levels of large numbers of genes in parallel. By finding correlations between expression levels and phenotypes, scientists can identify possible genetic markers for biological characteristics.

The data in this lab comes from:

https://archive.ics.uci.edu/ml/datasets/Mice+Protein+Expression

In this data, mice were characterized by three properties:

- Whether they had down's syndrome (trisomy) or not
- Whether they were stimulated to learn or not
- Whether they had a drug memantine or a saline control solution.

With these three choices, there are 8 possible classes for each mouse. For each mouse, the expression levels were measured across 77 genes. We will see if the characteristics can be predicted from the gene expression levels. This classification could reveal which genes are potentially involved in Down's syndrome and if drugs and learning have any noticeable effects.

1.2 Load the Data

We begin by loading the standard packages.

```
In [25]: # Imports
         import numpy as np
         import matplotlib.pyplot as plt
         import matplotlib
```

```
matplotlib.rcParams.update({'font.size':16})
%matplotlib inline
import matplotlib.image as mpimg
from pylab import rcParams

import random
import math
from numpy.linalg import inv

import pandas as pd
from sklearn import linear_model, preprocessing

from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

Use the `pd.read_excel` command to read the data from https://archive.ics.uci.edu/ml/machine-learning-databases/00342/Data_Cortex_Nuclear.xls into a dataframe `df`. Use the `index_col` option to specify that column 0 is the index. Use the `df.head()` to print the first few rows.

```
In [26]: # Import the data
         # (Data from: https://archive.ics.uci.edu/ml/datasets/Mice+Protein+Expression)

         io = 'https://archive.ics.uci.edu/ml/machine-learning-databases/00342/\
         Data_Cortex_Nuclear.xls'
         df = pd.read_excel(io, index_col=0)
         df.head()
```

```
Out[26]:          DYRK1A_N   ITSN1_N    BDNF_N     NR1_N    NR2A_N    pAKT_N   pBRAF_N  \
         MouseID
         309_1    0.503644  0.747193  0.430175  2.816329  5.990152  0.218830  0.177565
         309_2    0.514617  0.689064  0.411770  2.789514  5.685038  0.211636  0.172817
         309_3    0.509183  0.730247  0.418309  2.687201  5.622059  0.209011  0.175722
         309_4    0.442107  0.617076  0.358626  2.466947  4.979503  0.222886  0.176463
         309_5    0.434940  0.617430  0.358802  2.365785  4.718679  0.213106  0.173627

                  pCAMKII_N   pCREB_N    pELK_N   ...     pCFOS_N     SYP_N  H3AcK18_N  \
         MouseID                                  ...
         309_1     2.373744  0.232224  1.750936   ...    0.108336  0.427099   0.114783
         309_2     2.292150  0.226972  1.596377   ...    0.104315  0.441581   0.111974
         309_3     2.283337  0.230247  1.561316   ...    0.106219  0.435777   0.111883
         309_4     2.152301  0.207004  1.595086   ...    0.111262  0.391691   0.130405
         309_5     2.134014  0.192158  1.504230   ...    0.110694  0.434154   0.118481

                    EGR1_N  H3MeK4_N    CaNA_N  Genotype  Treatment  Behavior   class
         MouseID
         309_1    0.131790  0.128186  1.675652   Control  Memantine       C/S  c-CS-m
```

```
309_2    0.135103  0.131119  1.743610   Control  Memantine       C/S   c-CS-m
309_3    0.133362  0.127431  1.926427   Control  Memantine       C/S   c-CS-m
309_4    0.147444  0.146901  1.700563   Control  Memantine       C/S   c-CS-m
309_5    0.140314  0.148380  1.839730   Control  Memantine       C/S   c-CS-m

[5 rows x 81 columns]
```

This data has missing values. The site:
http://pandas.pydata.org/pandas-docs/stable/missing_data.html
has an excellent summary of methods to deal with missing values. Following the techniques there, create a new data frame `df1` where the missing values in each column are filled with the mean values from the non-missing values.

```
In [27]: # pd.set_option('display.max_rows', -1)
         # pd.reset_option(pat='display')

         col = df.columns # Column labels
         xnames = np.array(col[0:-4]) # Array of gene names
         onames = np.array(col[-4:]) # Array of object-type column names
         dfa = df[xnames]
         dfb = df[onames]
         # New data frame with missing values filled by linear interpolation
         df0 = dfa.astype(float).interpolate(axis=0)
         # NaNs in first rows weren't filled
         df0 = df0.astype(float).interpolate(axis=0,limit_direction='backward')
         # print(df0)
         # print(np.isnan(df0).any())
```

We next get the data as `numpy` arrays. For the predictors, `X`, we will the expression levels of the `ngene=77` genes. The expression levels are stored in the first 77 columns of the dataframe `df1`.

- Set `xnames` = the names of genes (you can get them from `df1.columns`)
- Set `X = a numpy` array with the values of the expression levels. (you can get this from `df1[xnames].values`)

```
In [28]: ngene = 77 # number of genes
         df1 = pd.concat([df0,dfb],axis=1) # New data frame
         X = df1[xnames].values # Array of the values of the expression levels
```

Now run the following code which will extract the `class` of each measurement into a vector y. The values y will have values 0 to 7 corresponding to the 8 classes. Our goal will be to predict y from `X`.

```
In [29]: # Extract the class of each measurement into a vector y
         ystr = df1['class'].values
         vals, y = np.unique(ystr, return_inverse=True)
```

Next, split the data into training and test. You can use the `train_test_split` function. Set `shuffle=True` and `test_size=0.5`.

```
In [30]: # Split the data into training and test sets
         from sklearn.model_selection import train_test_split
         Xtr, Xts, ytr, yts = train_test_split(X, y, shuffle=True,test_size=0.5)
```

1.3 PCA on the Data

We will first try to perform PCA. With PCA, it is import to first scale the data matrix to remove the mean and normalize the features by their variance. We can do the scaling and PCA in two steps using routines from `sklearn` package:

- Create a scaling object, `scaler = StandardScaler(...)` and `fit` and `transform` the scaler on the training data, `Xtr`.
- Create a PCA object, `pca = PCA(...)` and `fit` the PCA coefficients on the scaled data. In order that we can visualize the results, set `n_components=2`.

```
In [31]: # Does the contain contain an NaN value?
         print(np.isnan(Xtr).any())
         print(np.isnan(ytr).any())
         print(np.isnan(X).any())
```

```
False
False
False
```

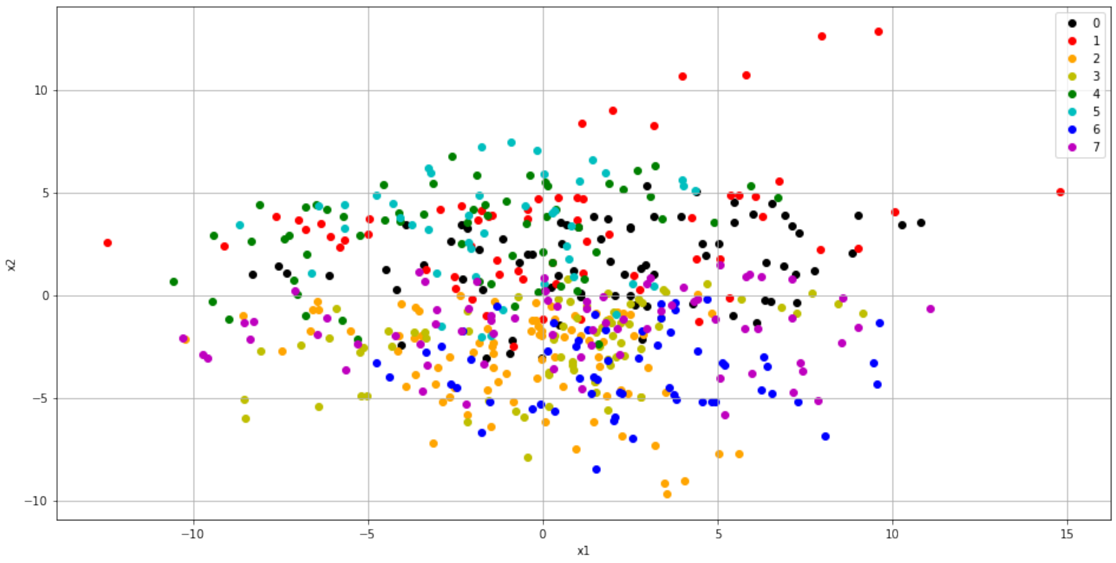
```
In [32]: from sklearn.decomposition import PCA
         from sklearn.preprocessing import StandardScaler

         # Standard scaling function
         def f_StandardScale(Xtr,Xts):
             scaler = StandardScaler()
             Xtr_ = scaler.fit_transform(Xtr)
             Xts_ = scaler.transform(Xts)
             return Xtr_,Xts_
```

Now use the `transform` method to transform the test data `Xts` through the `scaler` and `pca` objects. Create a scatter plot using the `plt.scatter` function of the points from the two classes. Use different colors for each class. You will see that the PCA representation does not differentiate the classes well along the two components

```
In [33]: # Create a scaling object, and fit and transform
         # the scaler on the training data.
         [Xtr_,Xts_] = f_StandardScale(Xtr,Xts)

         # Create a PCA object and fit the
         # PCA coefficients on the scaled data.
         pca = PCA(n_components=2)
         Xtr_pca = pca.fit_transform(Xtr_)
         # Transform the test data through the scalar and PCA objects
         Xts_pca = pca.transform(Xts_)
```

```
In [35]: # Scatter plot of the data
```

```python
num_classes = 8
fig, ax = plt.subplots()
cdict = ['k', 'r', 'orange', 'y', 'g', 'c', 'b', 'm'] # Color dictionary

datax = Xts_pca.T[0]
datay = Xts_pca.T[1]

for cc in np.arange(num_classes):
    ivec = np.where(yts == cc)
    # Generate 'num_classes' tuples equally spread
    # in hue space, then convert to RGB.
    # color = np.array(colorsys.hsv_to_rgb(cc*1.0/num_classes, 0.5, 0.5))
    # (doesn't work too well)
    color = cdict[cc]
    ax.scatter(datax[ivec],datay[ivec],c=color,label=cc)

plt.xlabel('x1')
plt.ylabel('x2')

ax.grid()
ax.legend(loc='best', bbox_to_anchor=(1,1))
rcParams['figure.figsize'] = 16, 8
plt.show()
```



1.4 LDA on the Data

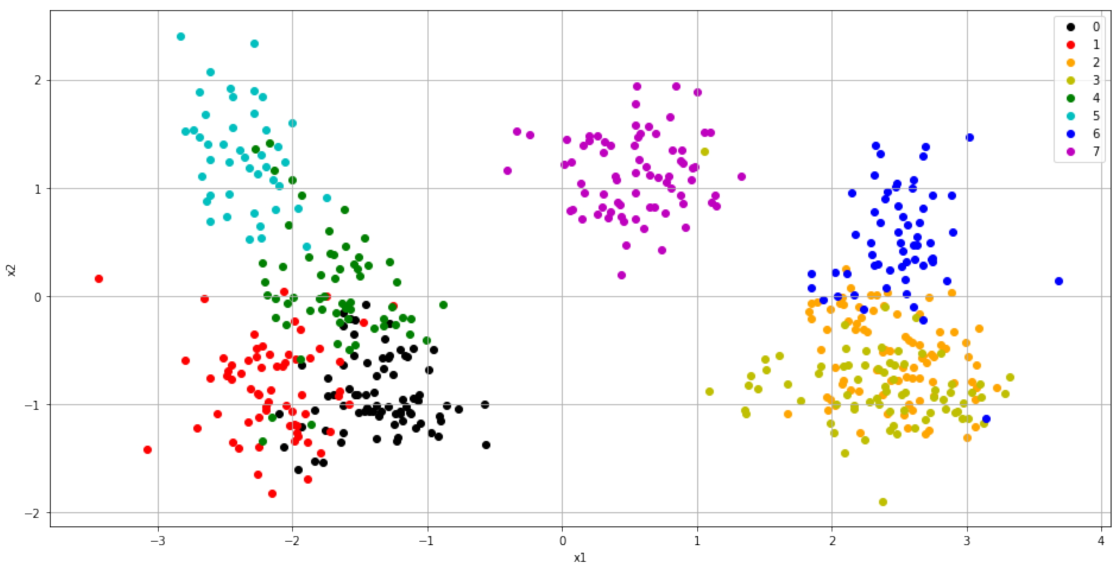
A better way to transform data in a way that separates classes is LDA. The `sklearn` has excellent routines for LDA.

- As in the PCA case, create a scaling object, `scaler = StandardScaler(...)` and `fit` and `transform` the scaler on the training data, `Xtr`.
- Next create an LDA object, `lda = LinearDiscriminantAnalysis(...)`. To avoid ill-conditioning, set `shrinkage='auto'` and `solver='eigen'`. Fit the LDA transform from the scaled output of `scaler`.

```
In [36]: # Create a scaling object, and fit and transform
         # the scaler on the training data.
         [Xtr_,Xts_] = f_StandardScale(Xtr,Xts)

         # Create an LDA object and fit the LDA transform.
         lda = LinearDiscriminantAnalysis(n_components=2, \
                                          shrinkage='auto', solver='eigen')
         Xtr_lda = lda.fit_transform(Xtr_, ytr)
         # Transform the test data through the scalar and LDA objects
         Xts_lda = lda.transform(Xts_)

         # print(Xts_lda)
```

Now transform the test data `Xts` through the `scaler` and `lda` objects. Create a scatter plot using the `plt.scatter` function of the points from the two classes. Use different colors for each class. You will see that LDA results in much better separation.

```
In [37]: # Scatter plot of the data

         num_classes = 8
         fig, ax = plt.subplots()
         cdict = ['k', 'r', 'orange', 'y', 'g', 'c', 'b', 'm'] # Color dictionary

         datax = Xts_lda.T[0]
         datay = Xts_lda.T[1]

         for cc in np.arange(num_classes):
             ivec = np.where(yts == cc)
             color = cdict[cc]
             ax.scatter(datax[ivec],datay[ivec],c=color,label=cc)

         plt.xlabel('x1')
         plt.ylabel('x2')

         ax.grid()
         ax.legend(loc='best', bbox_to_anchor=(1,1))
         rcParams['figure.figsize'] = 16, 8
         plt.show()
```



1.5 Logistic Regression on the LDA Data

We will now build a linear classifier from the LDA outputs. To fit the classifier, we use a three step pipeline:

- As above, create a scaling object, `scaler = StandardScaler(...)` and `fit` and `transform` the scaler on the training data, `Xtr`.

- Also, as above create an LDA object, `lda = LinearDiscriminantAnalysis(...)` and `fit` and `transform` the scaled training data. Call the transformed output `Ztr`.
- Create a logistic regression object, `logreg = linear_model.LogisticRegression(...)`. Set `solver='lbfgs'`, and `multi_class='auto'`. Fit the model on the transformed training data.

```
In [38]: # Create a scaling object, and fit and transform the scaler on the data.
         [Xtr_,Xts_] = f_StandardScale(Xtr,Xts)

         # Create an LDA object and fit the LDA transform.
         lda = LinearDiscriminantAnalysis(n_components=8, \
                                          shrinkage='auto', solver='eigen')
         Ztr = lda.fit_transform(Xtr_, ytr)
         # Transform the test data through the scalar and LDA objects
         Xts_lda = lda.transform(Xts_)

         # Create a logistic regression object
         logreg = linear_model.LogisticRegression(solver='lbfgs', \
                                                  multi_class = 'multinomial')
         Xtr_log = logreg.fit(Ztr,ytr)
```

Now test the model on the test data:

- Scale the test data `Xts` with the `scaler.transform()` method
- Transform the scaled test data with the `lda.transform()` method
- Predict the class labels from `logreg.predict`. Call the outputs `yhat`.
- Measure the accuracy by comparing the outputs `yhat` with `yts`.

If you did everything correctly, you should get an accuracy of around 94%.

```
In [39]: # Predict the class labels
         yhat = logreg.predict(Xts_lda)
         ydiff = np.absolute(yhat-yts)
         error = np.count_nonzero(ydiff)/np.size(ydiff)
         per_acc = (1-error)*100
         print('Accuracy:', round(per_acc,2),'%')
```

```
Accuracy: 96.67 %
```

1.6 K-Fold Cross Validation

K-Fold validation can yield better assessments of the accuracy when the training data is limited. Complete the following code to perform 5 fold validation.

```
In [21]: # A 5-fold cross validation of the data
         from sklearn.model_selection import KFold
         from sklearn.metrics import precision_recall_fscore_support
         nfold = 5
         kf = KFold(n_splits=nfold,shuffle=True)
         acc = np.zeros(nfold)

         for i, I in enumerate(kf.split(X)):
             # Get training and test data
             Itr, Its = I
             Xtr = X[Itr,:]
             ytr = y[Itr]
             Xts = X[Its,:]
             yts = y[Its]

             # Train the scaler, LDA, and logistic regression
             # model on the training data
             [Xtr_,Xts_] = f_StandardScale(Xtr,Xts)
             lda = LinearDiscriminantAnalysis(n_components=8, \
                                              shrinkage='auto', solver='eigen')
             Ztr = lda.fit_transform(Xtr_, ytr)
             # Transform the test data through the scalar and LDA objects
             Xts_lda = lda.transform(Xts_)
             logreg = linear_model.LogisticRegression(solver='lbfgs', \
                                                      multi_class = 'multinomial')
             Xtr_log = logreg.fit(Ztr,ytr)
```

```
            # Test the scaler, LDA, and regression model on the test data
            yhat = logreg.predict(Xts_lda)

            # Measure accuracy and store in acc[i]
            ydiff = np.absolute(yhat-yts)
            error = np.count_nonzero(ydiff)/np.size(ydiff)
            per_acc = (1-error)*100
            acc[i] = per_acc

        # Print the mean and SE of the accuracy
        acc_mean = np.mean(acc)
        acc_std = np.std(acc)
        print('Mean Accuracy:', round(acc_mean,2),'%')
        print('Accuracy SE:', round(acc_std,2),'%')
```

```
/Users/peterracioppo/anaconda3/lib/python3.6/site-packages/sklearn/discriminant_analysis.py:44
  UserWarning)


Mean Accuracy: 97.31 %
Accuracy SE: 0.9 %
```

1.7 More Fun

Statistical analysis of genetic analysis is a rich area and there are several simple things that you can explore as a continuation of this lab:

- Larger datasets
- Combining the K-fold validation with parameter optimization
- Using sparse LDA or sparse regression

1.7.1 K-Fold Cross Validation + Parameter Optimization

```
In [23]: # Vary the number of folds
         from sklearn.model_selection import KFold
         from sklearn.metrics import precision_recall_fscore_support

         fold_vec = np.arange(9)+2
         for nn in fold_vec:
             nfold = nn
             kf = KFold(n_splits=nfold,shuffle=True)
             acc = np.zeros(nfold)

             for i, I in enumerate(kf.split(X)):
                 # Get training and test data
                 Itr, Its = I
                 Xtr = X[Itr,:]
                 ytr = y[Itr]
```

```
                Xts = X[Its,:]
                yts = y[Its]

                # Train the scaler, LDA, and logistic
                # regression model on the training data
                [Xtr_,Xts_] = f_StandardScale(Xtr,Xts)
                lda = LinearDiscriminantAnalysis(n_components=8, \
                                                 shrinkage='auto', solver='eigen')
                Ztr = lda.fit_transform(Xtr_, ytr)
                # Transform the test data through the scalar and LDA objects
                Xts_lda = lda.transform(Xts_)
                logreg = linear_model.LogisticRegression(solver='lbfgs', \
                                                         multi_class = 'multinomial')
                Xtr_log = logreg.fit(Ztr,ytr)

                # Test the scaler, LDA, and regression model on the test data
                yhat = logreg.predict(Xts_lda)

                # Measure accuracy and store in acc[i]
                ydiff = np.absolute(yhat-yts)
                error = np.count_nonzero(ydiff)/np.size(ydiff)
                per_acc = (1-error)*100
                acc[i] = per_acc

            # Print the mean and SE of the accuracy
            acc_mean = np.mean(acc)
            acc_std = np.std(acc)
            print('Number of Folds:', nn)
            print('Mean Accuracy:', round(acc_mean,2),'%')
```

```
/Users/peterracioppo/anaconda3/lib/python3.6/site-packages/sklearn/discriminant_analysis.py:44
  UserWarning)


Number of Folds: 2
Mean Accuracy: 96.11 %
Number of Folds: 3
Mean Accuracy: 96.57 %
Number of Folds: 4
Mean Accuracy: 96.67 %
Number of Folds: 5
Mean Accuracy: 97.41 %
Number of Folds: 6
Mean Accuracy: 97.78 %
Number of Folds: 7
Mean Accuracy: 97.69 %
Number of Folds: 8
Mean Accuracy: 97.87 %
```

```
Number of Folds: 9
Mean Accuracy: 97.5 %
Number of Folds: 10
Mean Accuracy: 97.59 %
```

1.7.2 Sparse LDA

In [24]:
```
# Vary the number of LDA components
# A 5-fold cross validation of the data
from sklearn.model_selection import KFold
from sklearn.metrics import precision_recall_fscore_support
nfold = 5
kf = KFold(n_splits=nfold,shuffle=True)
acc = np.zeros(nfold)

comp_vec = np.arange(10)+1
for nc in comp_vec:

    for i, I in enumerate(kf.split(X)):
        # Get training and test data
        Itr, Its = I
        Xtr = X[Itr,:]
        ytr = y[Itr]
        Xts = X[Its,:]
        yts = y[Its]

        # Train the scaler, LDA, and logistic
        # regression model on the training data
        [Xtr_,Xts_] = f_StandardScale(Xtr,Xts)
        lda = LinearDiscriminantAnalysis(n_components=nc, \
                                        shrinkage='auto', solver='eigen')
        Ztr = lda.fit_transform(Xtr_, ytr)
        # Transform the test data through the scalar and LDA objects
        Xts_lda = lda.transform(Xts_)
        logreg = linear_model.LogisticRegression(solver='lbfgs', \
                                                multi_class = 'multinomial')
        Xtr_log = logreg.fit(Ztr,ytr)

        # Test the scaler, LDA, and regression model on the test data
        yhat = logreg.predict(Xts_lda)

        # Measure accuracy and store in acc[i]
        ydiff = np.absolute(yhat-yts)
        error = np.count_nonzero(ydiff)/np.size(ydiff)
        per_acc = (1-error)*100
        acc[i] = per_acc
```

```
            # Print the mean and SE of the accuracy
            acc_mean = np.mean(acc)
            acc_std = np.std(acc)
            print('Number of LDA Components:', nc)
            print('Mean Accuracy:', round(acc_mean,2),'%')
```

```
/Users/peterracioppo/anaconda3/lib/python3.6/site-packages/sklearn/discriminant_analysis.py:44
  UserWarning)


Number of LDA Components: 1
Mean Accuracy: 50.28 %
Number of LDA Components: 2
Mean Accuracy: 82.5 %
Number of LDA Components: 3
Mean Accuracy: 91.11 %
Number of LDA Components: 4
Mean Accuracy: 92.96 %
Number of LDA Components: 5
Mean Accuracy: 94.07 %
Number of LDA Components: 6
Mean Accuracy: 95.83 %
Number of LDA Components: 7
Mean Accuracy: 96.85 %
Number of LDA Components: 8
Mean Accuracy: 97.22 %
Number of LDA Components: 9
Mean Accuracy: 97.41 %
Number of LDA Components: 10
Mean Accuracy: 97.31 %
```