ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ ΣΤΗΝ ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ

Εργασία 2

KPCA+LDA

Υπολογιστική Νοημοσύνη-Στατιστική Μάθηση

Σουράνης Παναγιώτης

ΑΕΜ:17

Στο παρακάτω κείμενο ακολουθεί η περιγραφή των αλγορίθμων KPCA plus LDA, ακολουθεί πρώτα ο αλγόριθμος **KPCA**

```
from scipy.linalg import eigh
import numpy as np


class KPCA(object):

    def __init__(self, kernel=None, n_components=None,percentage=None):
        self.kernel = kernel
        self.n_components= n_components
        self.percentage=percentage
        if self.n_components is not None: self.n_components=int(self.n_components)
        if self.percentage is not None:self.percentage=float(self.percentage)


    def fit_transform(self,X):

        n_samples, n_features = X.shape
        self.x_fit=X
        self.array=n_samples
        self.features=n_features

        K = np.zeros((n_samples, n_samples)) #construct our Gram Matrix dimension NXN
        for i in range(n_samples):
            for j in range(n_samples):
                K[i,j] = self.kernel(X[i], X[j])

        self.K_train = K #we keep the gram matrix of train from projections of test

        One_ = np.ones((n_samples,n_samples))/n_samples #construct our 1_N matrices
        self.K = K -  One_.dot(K) -K.dot(One_) + One_.dot(K).dot(One_) #Centerizing our Gram Matrix

        eigenvalues , eigenvectors = eigh(self.K) #Find Eigenvalues and Eigenvectors
        eigenvalues=np.abs(eigenvalues)
        self.eig = eigenvalues
        self.projections = eigenvectors.dot(np.diag(np.sqrt(eigenvalues) / eigenvalues))

        eigenvalues = np.sqrt(eigenvalues)
        eigenvectors = eigenvectors.dot(np.diag(eigenvalues))

        if self.n_components is None:
            self.explain_variance_ratio()
            self.n_components=self.get_percentage()


        #We keep our components that correspond to the higher eigenvectors
        #They are in descenting order so we need to reverse them

        self.pcomponents=np.column_stack(eigenvectors[:,-i] for i in range(1,self.n_components+1))

        return(self.pcomponents)
```

Στο κομμάτι αυτό ορίζουμε την κλάση μας KPCA,η συνάρτηση __init__ θα παιρνει τα ορίσματα kernel,n_components,percentage οπου kernels οι πυρήνες που έχουμε ορίσει εμείς ποιοι θέλουμε να είναι ανεξάρτητα από το KPCA,n_components θα είναι το πλήθος των χαρακτηριστικών που θέλουμε να κρατήσουμε και percentage θα είναι το επιθυμητο ποσοστό που θέλουμε να κρατήσουμε αν δεν ξέρουμε πόσα components χρειαζόμαστε για να έχουμε διατηρήσει συγκεκριμένο ποσοστό πληροφορίας.

Παρακάτω η συνάρτηση fit_transform δέχεται ως όρισμα έναν πίνακα X του οποίου κρατάει τις διαστάσεις ως **n_samples** και **n_features** (τα οποία διατηρούμε) ,επίσης διατηρούμε τον X ώς self.x_fit προκειμένου να τον χρησιμοποίησουμε στην συνέχεια.

Στη συνέχεια υπολογίζουμε των Gram Matrix ο οποίος θα έχει διάσταση NxN δηλάδη όσο το πλήθος των δειγμάτων που έχουμε και θα είναι ενας πίνακας που σε κάθε στοιχείο i,j θα περιέχει το $\Phi(X_i)\Phi(X_j)$ όπου Φ η συνάρτηση πυρήνας.

$$K = \begin{pmatrix} \Phi(X_1)\Phi(X_1) & \dots & \Phi(X_1)\Phi(X_n) \\ \dots & \dots & \dots \\ \Phi(X_n)\Phi(X_1) & \dots & \Phi(X_n)\Phi(X_n) \end{pmatrix}$$

Τον πίνακα K τον διατηρούμε στην μεταβλητή self.K_train προκειμένου και αυτός να χρησιμοποιηθεί στην συνέχεια όταν θα χρειαστεί να υπολογίσουμε τα projections από τα test δείγματα.

Αφότου έχουμε υπολογίσει τον πίνακα K στην συνέχεια τον κεντράρουμε σύμφωνα με την παρακάτω σχέση.

$$\widetilde{K} = K - 1_N K - K 1_N + 1_N K 1_N$$

Όπου ο πίνακας $1_N = \left(\frac{1}{N}\right)_{NxN}$

Υπολογίζουμε τα ιδιοδιανύσματα $\gamma_1, \gamma_2, \dots., \gamma_N$ τα οποία αντιστοιχούν στις ιδιοτιμές $\lambda_1 \le \lambda_2 \le \lambda_3 \le \cdots \le \lambda_N$ καθώς οι ιδιοτιμές υπολογίζονται με φθίνουσα σειρά και για αυτό τον λόγο πραγματοποιούμε sorting έτσι ώστε να έχουμε τα ιδιοδιανύσματα $\gamma_1, \gamma_2, \dots., \gamma_N$ που αντιστοιχούν στις μεγαλύτερες ιδιοτιμές.

Ακόμη κρατάμε σε έναν πίνακα self.projections έναν διαγώνιο πίνακα

$$\Lambda = E \begin{pmatrix} \frac{\sqrt{|\lambda_1|}}{|\lambda_1|} & \dots & 0 \\ \dots & \frac{\sqrt{|\lambda_2|}}{|\lambda_2|} & \dots \\ \dots & \dots & \dots \\ 0 & \dots & \frac{\sqrt{|\lambda_N|}}{|\lambda_N|} \end{pmatrix} E = (e_1, \dots e_N)\,, e_i \; το \; i \; ιδιοδιάνυσμα$$

Προκειμένου να χρησιμοποιηθεί στην προβολή των test δεδομένων.

Αν ο χρήστης δεν έχει δώσει πλήθος χαρακτηριστικών που θέλει να κρατήσει καλείται η συνάρτηση self.explain_variance_ratio() προκειμένου να υπολογίσει το ποσοστό

πληροφορίας που κρατάει η κάθε ιδιοτιμή και στην συνέχεια το πλήθος των χαρακτηριστικών που χρειάζονται για συγκεκριμένο percentage δίνεται από την συνάρτηση self.get_percentage.

Εφόσον έχουμε πραγματοποιήσει αυτά δεν έχουμε παρα μόνο να επιστρέψουμε τα ιδιοδιανύσματα σε φθίνουσα σειρά μέχρι το σημείο n_components.

Αυτή θα είναι η προβολή των δεδομένων μας στους x_i $axes$

```
def explain_variance_ratio(self):
    sum_eigenvalues=np.sum(self.eig)
    self.eig=self.eig/sum_eigenvalues #explain variance ratio
    non_trivial=[]
    for i in self.eig:
        if i>1e-5: #threshold
            non_trivial.append(i)
    self.non_trivial=np.array(sorted(non_trivial,reverse=True)) #sort them in descenting order
    return(np.round(self.non_trivial,5))

def get_percentage(self):
    count=0
    sum_percent = 0.0
    for i in self.non_trivial:
        if sum_percent < self.percentage:
            sum_percent += i
            count += 1
        else:
            break
    return(count)

def transform(self,X): #transform our test data

    n_samples,n_features = X.shape
    K = np.zeros((n_samples,self.array))
    for i in range(n_samples):
        for j in range(self.array):
            K[i,j] = self.kernel(X[i],self.x_fit[j])


    One_ = np.ones((n_samples,self.array)) / self.array #1'M will have dimension LXN and each element 1/N
    Ones = np.ones((self.array,self.array))/self.array #1M will have dimension NXN and each element 1/N


    K = K -  One_.dot(self.K_train) -K.dot(Ones) + One_.dot(self.K_train).dot(Ones) #centerizing the matrix

    pc_new = K.dot(self.projections)
    self.pc_new=np.column_stack(pc_new[:,-i] for i in range(1,self.n_components+1))
    return(self.pc_new)

def get_eigens(self):
    return(self.eig)

def cummulative_percentage(self):
    cumperce=np.array(self.explain_variance_ratio())
    for i in range(1,len(cumperce)):
        cumperce[i]=cumperce[i]+cumperce[i-1]
    return(cumperce)
```

Αυτό που απομένει λοιπόν να κάνουμε είναι να υπολογίσουμε την προβολή των test δεδομένων με την συνάρτηση transform.

Η συνάρτηση transform δέχεται ως όρισμα έναν πίνακα X ο οποίος μπορεί να είναι και ένα μοναδικό διάνυσμα και κρατάει τις διαστάσεις του ως n_samples και n_features,στην συνέχεια υπολογίζουμε έναν καινούριο gram matrix K^{test}ο οποίος θα έχει διασταση NxM όπου N το πλήθος των test samples και M το πλήθος των train samples,αρα περιέχει ολους τους δυνατούς συνδυασμούς με χρήση της συνάρτησης πυρήνα μεταξυ των test δειγμάτων και train

$$K^{test} = \begin{pmatrix} \Phi\left(X_{test_1}\right)\Phi(X_{train_1}) & \cdots & \Phi\left(X_{test_1}\right)\Phi(X_{train_M}) \\ \vdots & \ddots & \vdots \\ \Phi\left(X_{test_N}\right)\Phi(X_{train_1}) & \cdots & \Phi\left(X_{test_N}\right)\Phi(X_{train_M}) \end{pmatrix} K_{NxM}$$

Στην συνέχεια δημιουργούμε τους πίνακες

$$1'_M = \left(\frac{1}{M}\right) \dim NxM$$

$$1_M = \left(\frac{1}{M}\right) \dim MxM$$

Προκειμένου να μπορέσουμε να κεντράρουμε τον καινούριο πίνακα K^{test}, στην συνέχεια βάση του παρακάτω τύπου κεντράρουμε τον πινακα K^{test}

$$\widetilde{K}^{test} = K^{test} - 1'_M K^{train} - K^{test} 1_M + 1'_M K^{train} 1_M$$

Στην συνέχεια υπολογίζουμε τον πίνακα

$$P = \widetilde{K}^{test} \Lambda = \widetilde{K}^{test} ED \text{ όπου } D = diag(\frac{\sqrt{|\lambda_1|}}{|\lambda_1|}, \ldots., \frac{\sqrt{|\lambda_N|}}{|\lambda_N|})$$

Τέλος το μόνο που απομένει να κάνουμε είναι να πραγματοποιήσουμε sorting σε αύξουσα σειρά μέχρι το πλήθος των n_components που επιθυμούμε και αυτές θα είναι οι προβολές των test δειγμάτων στους x_i $axes$

References

[1] Nonlinear Component Analysis as a Kernel Eigenvalue Problem Bernhard Scholkopf Alexander Smola and KlausRobert Muller.

[2] KPCA Plus LDA: A Complete Kernel Fisher Discriminant Framework for Feature Extraction and Recognition Jian Yang, Alejandro F. Frangi, Jing-yu Yang, David Zhang, Senior Member, IEEE, and Zhong Jin.

Ακολουθεί στην συνέχεια η περιγραφή του αλγορίθμου **LDA**

```
import numpy as np

class LDA(object):
    def __init__(self,n_components=None):
        self.n_components=n_components

    def compute_S_k(self,X,meanvectors):
        n_features=X.shape[1]
        meanvectors_=meanvectors
        S_k=np.zeros((n_features,n_features))
        Z=X-meanvectors_
        S_k=np.dot(Z.T,Z)
        return(S_k)


    def fit(self,X,y):
        #Compute mean Vectors#
        classes = np.unique(y)
        means = []
        self.n_features=X.shape[1]
        for group in classes:
            X_classes = X[y == group, :]
            means.append(np.mean(X_classes,axis=0))
        self.meanvectors=np.asarray(means)


        #Compute S_within

        S_w=np.zeros((self.n_features,self.n_features))

        for i in range(len(classes)):
            X_covs_k=(X[ y==classes[i] , :])
            means=np.transpose(self.meanvectors[i])
            S_w+= self.compute_S_k(X_covs_k,means)
        self.S_w=S_w


        #Compute S_Between

        classes,counts=np.unique(y,return_counts=True)

        S_B=np.zeros((self.n_features,self.n_features))
        overall_mean=np.mean(X,axis=0)

        for i in range(len(classes)):
            N=counts[i]
            means=self.meanvectors[i]-overall_mean
            means=means.reshape(self.n_features,1)
            S_B+= N * np.dot(means,means.T)
        self.S_B=S_B


        #Finding Eigenvalues and eigenvectors#
        self.eig_vals, self.eig_vecs = np.linalg.eig(np.linalg.inv(self.S_w).dot(self.S_B))

        #Pair our eigenvalues and eigenvectors
        self.eig_pairs=[(np.abs(self.eig_vals[i]),self.eig_vecs[:,i]) for i in range(self.n_features)]
```

Η συνάρτηση __init__δέχεται ως όρισμα το πλήθος των components που επιθυμούμε να κρατήσουμε,στην συνεχεια η συναρτηση compute S_k χρησιμοποιείται ως βοηθητική συναρτηση για να μπορέσουμε να υπολογίσουμε τον πίνακα S_w οποίος ορίζεται ώς $S_w = \sum_k S_k$,όπου $S_k = (X - m_k)(X - m_k)^T$

Στην συνέχεια αφού βρούμε τον πίνακα S_w βρίσκουμε τον πίνακα

$S_B = \sum_k N_k (m_k - m)(m_k - m)^T$ όπου k είναι όσο το πλήθος των κλάσεων.

Αφού έχουμε υπολογίσει αυτούς τους δύο πίνακες στην συνέχεια βρίσκουμε τις ιδιοτιμές και τα ιδιοδιανύσματα από τον πίνακα $S_w^{-1} S_B$ τα οποία ομαδοποιούμε στην μεταβλητή eig_pairs

```
        #Finding our transform matrix W
        if self.n_components is None: self.keepcomponents = sum(self.eig_vals>1e-3)
        else: self.keepcomponents = self.n_components

        self.W=np.column_stack(self.eig_vecs[:,i] for i in range(self.keepcomponents)) #stack the columns of the eigenvectors


    def transform(self,X):
        return(X.dot(self.W))

    def variance_explained(self):
        sum_eigvals=np.sum(self.eig_vals)
        explained_ratio=np.round(self.eig_vals/sum_eigvals,3)
        return(explained_ratio)
```

Τέλος για να υπολογίσουμε τον τελικό μας πίνακα W αν δεν είχε δωθεί συγκεκριμένος αριθμός από n_components τοτε το πλήθος τους θα είναι όσες οι μη μηδενικές ιδιοτιμές.Αφού βρούμε και το πλήθος το μόνο που μένει είναι να κρατήσουμε στον πίνακα W μέχρι την στήλη που θα αντιστοιχεί στον αριθμό self.keepcomponents

Ακόμη η συνάρτηση variance_explained μας επιστρέφει το ποσοστό πληροφορίας που κρατάει η κάθε ιδιοτιμή.

Και οι δύο οι αλγόριθμοι εφαρμόστηκαν στην αρχή στα datasets Mnist handwritten digits,Iris και συγκρήθηκαν με τους αλγορίθμους του scikit learn.

Τα αποτελέσματα που προέκυψαν ήταν ακριβώς ίδια με την μόνη διαφορά ότι οι αλγόριθμοι του scikit learn έιχανε μικρότερο χρόνο εκτέλεσης κάτι που οφείλεται στο γεγονός ότι η βιβλιοθήκη scikit learn κανει εκτεταμένη χρήση των γεννητριών συναρτήσεων.