Universitatea POLITEHNICA din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Proiect de semestru

Clasificarea tipurilor de vinuri folosind PCA și metode de clasificare non-neurale

Student
Alex-Costin Țițu

Anul 2024

Cuprins

Sumar: Se pune problema clasificării tipurilor de vinuri în trei categorii, pe baza celor 13 caracteristici definitorii ale acestora. Setul de date pus la dispoziție conține 178 de eșantioane etichetate, fiecăruia fiindu-i atribuit soiul (clasa) de vin din care provine. Cu ajutorul acestor informații, se propune utilizarea algoritmului de analiză a componentelor principale (en. Principal Component Analysis – PCA) în vederea extragerii trăsăturilor semnificative ale vinurilor și clasificarea vinurilor cu un algoritm de clasificare non-neural. Algoritmii de clasificare non-neurali propuși sunt următorii: Nearest Neighbor, 3-Nearest Neighbor, Nearest Prototype (K-Means Supervizat). Prin utilizarea acestor metode, se urmărește atingerea celor mai bune rezultate în determinarea soiului de vin la un cost minim de resurse computaționale. Rezultatele finale vor fi comparate atât între cei trei clasificatori propuși, cât și cu alți clasificatori neurali.

1 Fundamente teoretice

1.1 Selecția caracteristicilor folosind modele stohastice

Selecția caracteristicilor este un proces de transformare liniară sau neliniară a spațiului inițial al observațiilor (setul de date de intrare) într-un spațiu de dimensionalitate redusă în care algoritmul de clasificare a datelor funcționează. Transformarea este necesară în cazul anumitor algoritmi de clasificare, aceștia fiind mult mai eficienți într-un spațiu cu un număr de dimensiuni redus. De asemenea, anumite aplicații necesită luarea deciziilor în timp real, astfel prin utilizarea unei dimensionalități reduse a datelor de intrare, volumul de calcule necesar este semnificativ redus și procesul accelerat.

„Dacă considerăm setul de date analizat ca aparținând unei singure clase, criteriul de selecție a caracteristicilor se referă la eficiența transformării în reducerea dimensiunii vectorilor privind conservarea informației originale. […] În cazul mai multor clase, selecția caracteristicilor este funcție de eficiența transformării privind realizarea simultană a reducerii numărului de dimensiuni și maximizarea separabilității claselor. În aceasta categorie se încadrează analiza discriminatorie liniară (en. Linear Discriminant Analysis-LDA).”

În cadrul soluției propuse s-a considerat că setul de date analizat aparține unei singure clase, criteriul de selecție a caracteristicilor fiind acela de reducere a dimensiunii în timp ce informația originală este conservată. Astfel, algoritmii de clasificare non neurali propuși vor lua deciziile mult mai rapid și mai eficient, fără a utiliza și informația redundantă prezentă în cadrul datelor.

Metoda de selecție a caracteristicilor utilizată ce realizează ceea ce mi-am propus este analiza componentelor principale (en. Principal Component Analysis – PCA)

1.2 Analiza componentelor principale (PCA)

Analiza componentelor principale (en. Principal Component Analysis – PCA) cunoscută și sub numele de Transformata Karhunen–Loève este o metodă statistică de selecție a caracteristicilor cu un grad ridicat de eficiență, care în ultimele decenii a devenit o metodă de referință pentru aplicațiile de data mining și pattern recognition.

Fiecare clasă prezentă în setul de date poate fi caracterizată de densitatea de probabilitate specifică. Analiza componentelor principale este adesea utilizată în cazul în care selecția caracteristicilor este realizată fără a mai lua în considerare clasele diferite ale vectorilor setului de date, presupunând că toți vectorii au aceleași proprietăți statistice, indiferent de clasa de care aparțin. (histogramele claselor?)

Fie $\underline{\boldsymbol{X}}$ un vector aleator n-dimensional. Se caută o transformare ortogonală care să permită reprezentarea optimă (decorelare) a vectorului $\underline{\boldsymbol{X}}$ în raport cu criteriul erorii medii pătratice minime. Această eroare se referă la diferența între vectorul original și vectorul reconstruit după transformarea prin PCA într-un spațiu cu dimensionalitate redusă și apoi revenirea în spațiul original.

Fie $\underline{\boldsymbol{K}}$ transformarea KLT (PCA) căutată:

$$\underline{\boldsymbol{K}} = (\underline{\Phi}_1, \underline{\Phi}_2, \dots, \underline{\Phi}_n)^t \tag{1.1}$$

unde Φ_i sunt vectori n-dimensionali, deocamdată necunoscuți.

Se impune ca vectorii coloană ai lui $\underline{\boldsymbol{K}}$ să formeze un sistem ortonormat:

$$\underline{\Phi}_i^t \cdot \underline{\Phi}_j = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \tag{1.2}$$

Fiecare vector $\underline{\boldsymbol{X}}$ se transformă în

$$\underline{\boldsymbol{Y}} = \underline{\boldsymbol{K}}\,\underline{\boldsymbol{X}} = (\mathrm{y}_1, \mathrm{y}_2, \dots, \mathrm{y}_n)^t \tag{1.3}$$

cu $y_i = \Phi_i^t \cdot \underline{\boldsymbol{X}}$.

Relațiile (1.1) și (1.2) conduc la concluzia că matricea K este ortogonală având relația:

$$\underline{K} \cdot \underline{K}^t = \underline{K}^t \cdot \underline{K} = \underline{I}_n \tag{1.4}$$

unde $\underline{I}_n$este matrice unitate de dimensiune *n x n.* Rezultată:

$$\underline{X} = \underline{K}^t \cdot \underline{Y} = \sum_{i=1}^{n} y_i \cdot \underline{\Phi}_i \tag{1.5}$$

Se vor reține numai m < n componente ale lui $\underline{\boldsymbol{Y}}$, restul de n – m componente fiind înlocuite cu constantele preselectate b_i, astfel că se estimează $\underline{\boldsymbol{X}}$ ca:

$$\underline{\hat{X}}(m) = \sum_{i=1}^{m} y_i \cdot \underline{\Phi}_i + \sum_{i=m+1}^{n} b_i \cdot \underline{\Phi}_i$$

Eroare corespunzătoare acestei estimări este:

$$\Delta\underline{X}(m) = \underline{X} - \underline{\hat{X}}(m) = \sum_{i=m+1}^{n} (y_i - b_i) \cdot \underline{\Phi}_i$$

Se consideră criteriul minimizării erorii pătratice medii:

$$\overline{\varepsilon^2}(m) = E\left\{\left\|\Delta\underline{X}(m)\right\|^2\right\} = E\left\{\sum_{i=m+1}^{n}\sum_{j=m+1}^{n} (y_i - b_i)(y_j - b_j) \cdot \underline{\Phi}_i^t \cdot \underline{\Phi}_j\right\}$$

unde E{ } semnifică operatorul de mediere statistică. Ținând seama de relația (1.2), rezultă:

$$\overline{\varepsilon^2}(m) = \sum_{i=m+1}^{n} E\{(y_i - b_i)^2\} \tag{1.6}$$

Funcția de eroare $\overline{\varepsilon^2}(m)$ se minimizează alegând în mod adecvat vectorii $\underline{\Phi}_i$ care definesc matricea K, precum și constantele b_i.

1.2.1 Algoritmul PCA

Fie setul de vectori n-dimensionali

$$\xi = \{X_1 \dots X_L\}$$

1. Se calculează matricea de covariație:

$$\underline{\Sigma}_x = \frac{1}{L}\sum_{i=1}^{L}(\underline{X}_i - \underline{\mu}_x)(\underline{X}_i - \underline{\mu}_x)^t \tag{1.7}$$

unde $\underline{X}_i \epsilon R^n$ sunt vectorii de intrare n-dimensionali ($i = 1,\dots,L$) și:

$$\underline{\mu}_x = \frac{1}{L}\sum_{i=1}^{L} \underline{X}_i \tag{1.8}$$

este media vectorilor de intrare.

2. Se determină valorile proprii λ_i ($i = 1,..,n$) ale matricei $\underline{\Sigma}_X$ ca soluții ale ecuației:

$$\left|\underline{\Sigma}_x - \lambda \cdot \underline{I}_n\right| = 0 \tag{1.9}$$

unde $\underline{I}_n$ este matricea unitate n-dimensională și λ este necunoscuta. Se ordonează descrescător rădăcinile ecuației, reprezentând valorile proprii și se rețin cele mai mari m valori proprii dintre ele.

3. Se deduc vectorii proprii din ecuațiile:

$$\underline{\Sigma}_x \cdot \underline{\Phi}_i = \lambda_i \cdot \underline{\Phi}_i \tag{1.10}$$

unde

$$\underline{\Phi}_i^t \cdot \underline{\Phi}_i = 1 \tag{1.11}$$

cu $\underline{\Phi}_1, .., \underline{\Phi}_m, \underline{\Phi}_{m=1}, .., \underline{\Phi}_n$ vectorii proprii ai matricei $\underline{\Sigma}_X$ corespunzând valorilor proprii considerate în ordine crescătoare.

Se deduce matricea ortogonală a transformării KLT:

$$\underline{K} = \left(\underline{\Phi}_1 | \ .. \ | \underline{\Phi}_n\right)^t \tag{1.12}$$

În urma reducerii la *m* vectori proprii, corespunzători primelor m valori proprii, se obține matricea KLT trunchiată denumită mai departe PCA:

$$PCA = \left(\underline{\Phi}_1 | \ .. \ | \underline{\Phi}_n\right)^t \tag{1.13}$$

Aplicând transformarea PCA cu ajutorul matricei definite anterior, se obțin vectorii cu dimensionalitate redusă în spațiul m-dimensional ($m < n$).

$$\underline{Z}_i = PCA \cdot \underline{X}_i \tag{1.14}$$

Factorul de conservare a informației prin aplicarea PCA se poate exprima cu formula:

$$\eta = \frac{\sum_{i=1}^{m} \lambda_i}{\sum_{i=1}^{n} \lambda_i} \times 100\% \tag{1.15}$$

unde $\lambda_i$sunt valorile proprii ordonate descrescător, iar η este notația factorului de conservare a informației.

1.3 Clasificatori non-neurali supervizați

1.3.1 Clasificatorul celui mai apropiat vecin (Nearest Neighbor - NN)

Algoritmul celui mai apropiat vecin (en. Nearest Neighbor - NN) este un clasificator neparametric, care alocă vectorul de intrare de clasificat *X*, acelei clase care corespunde celui mai apropiat vecin al lui *X* din lotul vectorilor de referință (etichetați).

Algoritmul se desfășoară în felul următor:

Fie setul de N vectori de referință etichetați:

$$\chi = \{X_1 \dots X_N\}$$

Cu etichetele de apartenență la una din cele M clase corespunzătoare

$$\Omega_{(\chi)} = \left\{\omega_{(X_1)} \dots \omega_{(X_N)}\right\}$$

unde

$$\omega_{(X_i)} \in \{\omega_1 \dots \omega_m\}, \text{ i = 1...N,}$$

Dacă se aplică la intrare un vector X care trebuie clasificat, se determină distanța minimă de la X la vectorii de referință etichetați. Se presupune că:

$$d(X, X_j) = min\{d(X, X_j), i = 1 \dots N\}$$

ceea ce înseamnă că X_j este cel mai apropiat vecin al lui X. Dacă

$$\omega(X_j) = \omega_h$$

atunci decidem că

$$\omega(X) = \omega_h$$

Altfel spus, se alocă vectorului X clasa celui mai apropiat vecin din lotul de vectori etichetați (în acest caz *h*).

Această idee poate fi extinsă la cei mai apropiați k vectori vecini etichetați ai lui X, astfel încât X este atribuit clasei care este reprezentată printr-o majoritate a celor mai apropiați k vecini. Se obține astfel algoritmul de clasificare *k*-NN (al celor mai apropiați k vecini). Pentru k > 1 și M = 2 clase, se alege k impar.

1.3.2 Algoritmul K-means supervizat (Nearest Prototype – NP)

Se consideră un set de N vectori de referință etichetați

$$\chi = \{X_1 \dots X_N\}$$

cu etichetele claselor corespunzătoare

$$\Omega_{(\chi)} = \{\omega_{(X_1)} \dots \omega_{(X_N)}\}$$

unde

$$\omega_{(X_i)} \in \{\omega_1 \dots \omega_m\}, \text{ i = 1...N,}$$

iar notațiile au semnificațiile de la paragraful precedent.

Se calculează prototipurile μ_1, μ_2,.., μ_M ale fiecărei clase ca medii aritmetice ale vectorilor care aparțin clasei respective.

Pentru orice vector de intrare X ce nu este etichetat, regula de clasificare va fi:

$$\omega(X) = \omega_h = arg\left(\min_{k=1\dots M} \|X - \mu_k\|\right)$$

2 Desfășurarea experimentelor și rezultatele obținute

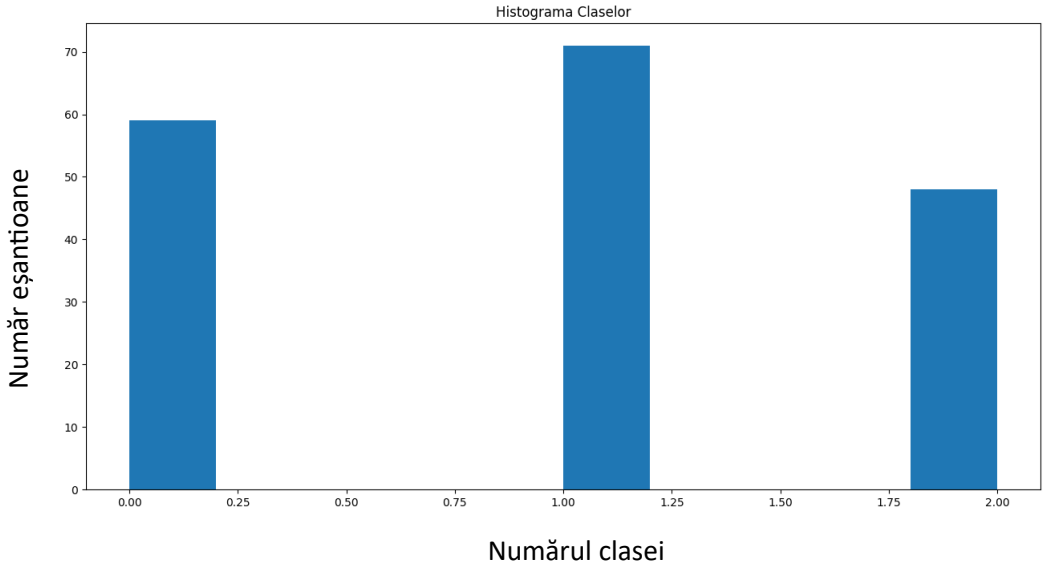
2.1 Descrierea setului de date

Setul de date de vinuri (en. Wine Dataset) este format din rezultatele unor analize chimice a vinurilor cultivate în aceeași regiune din Italia, dar derivate din trei soiuri diferite. Analiza acestora a condus la determinarea cantităților a 13 constituenți găsiți în fiecare dintre cele trei tipuri de vinuri. Constituenții ce descriu fiecare soi de vin sunt: Alcool, Acid Malic, Cenușă, Alcalinitatea cenușii, Magneziu, Total fenoli, Flavanoide, Fenoli neflavanoizi, Proantocianine, Intensitatea culorii, Nuanță, OD280/OD315 a vinurilor diluate, Proline.

Setul de date de vinuri este un set de date reper pentru analiza și determinarea performanțelor algoritmilor neurali sau non-neurali de clasificare supervizați. Astfel, cu ajutorul acestuia vor fi studiate performanțele celor doi algoritmi prezentați, în funcție de numărul de componente principale reținute, precum și în funcție de procentul de date de antrenare utilizate.

2.2 Rezultate experimentale

Prima etapă a fost încărcarea setului de date și studierea caracteristicilor acestuia. Setul de date a fost încărcat cu ajutorul bibliotecii sklearn, iar pentru studiul acesteia au fost afișate tipurile de atribuite, histogramele valorilor acestora, numărul de clase și histograma etichetelor atributelor. Majoritatea informațiilor aflate în această etapă au fost prezentate în paragraful 2.1.



Figură 1 Histograma claselor setului de date de vinuri

Conform histogramei, se poate observa faptul că majoritatea eșantioanelor se află în clasele 0 și 1, în clasa 2 fiind mai puține exemple față de primele două. Se va ține cont de acest lucru la separarea datelor experimentale în date de antrenare și date de testare.

Următorul pas a fost implementarea algoritmului PCA și determinarea matricei de transformare KL (Karhunen–Loève). În implementarea algoritmului a fost utilizată teoria prezentată în cadrul paragrafului 1.2, iar pentru aflarea valorilor și vectorilor proprii a fost utilizat modulul „linalg.eig" din cadrul bibliotecii NumPy (en. eig – eigen value/vector). La aflarea valorilor și vectorilor proprii pentru întregul set de date, matricea K este determinată în cazul reținerii, pe rând, a unul, doi, până la toți vectorii proprii aferenți valorilor proprii.

Număr valori proprii reținute	Factorul de conservare a informației	Scor de clasificare Nearest Neighbor	Scor de clasificare 3-Nearest Neighbor	Scor de clasificare K-Means Supervizat
1	99.809	0.666	0.703	0.740
2	99.982	0.685	0.740	0.611
3	99.992	0.833	0.759	0.722
4	99.997	0.814	0.759	0.722
5	99.998	0.796	0.814	0.722
6	99.999	0.833	0.833	0.722
7	99.999	0.833	0.814	0.722
8	99.999	0.722	0.759	0.722
9	99.999	0.722	0.759	0.722
10	99.999	0.722	0.759	0.722
11	99.999	0.722	0.759	0.722
12	99.999	0.722	0.759	0.722
13	100.00	0.722	0.777	0.722

Tabel 1 Evoluția scorului de clasificare în funcție de numărul de componente principale reținute

După cum s-a putut observa, fiecare algoritm își atinge performanțele maxime pe acest lot de date la reținerea unui număr diferit de valori proprii aferente vectorilor proprii. Cel mai performant algoritm este Nearest Neighbor cu un scor de clasificare pe lotul de test de 0.833 la reținerea doar a 3 valori proprii. Ca scor de clasificare este urmat de 3-Nearest Neighbor cu un scor de 0.833 însă pentru reținerea a 6 valori proprii, făcându-l mai ineficient computațional. În ultimul rând, algoritmul K-Means Supervizat reușește să obțină un scor de 0.74 pentru reținerea unei singure valori proprii. Deși K-Means Supervizat este cel mai puțin precis algoritm de clasificare supervizată pentru acest lot de antrenare, el este cel mai eficient, obținând cel mai mare scor la reținerea celui mai mic număr posibil de valori proprii.

Ținând cont că este o diferență semnificativă în scorul de clasificare al algoritmului K-Means Supervizat pentru o singură valoare proprie reținută și algoritmul Nearest Neighbor pentru reținerea a 3 valori proprii, voi continua cercetarea cu reținerea celor 3 valori proprii pentru a

maximiza eficiența computațională, dar și acuratețea. Reținerea celor 3 valori proprii duce la un factor de conservare de 99.99%.

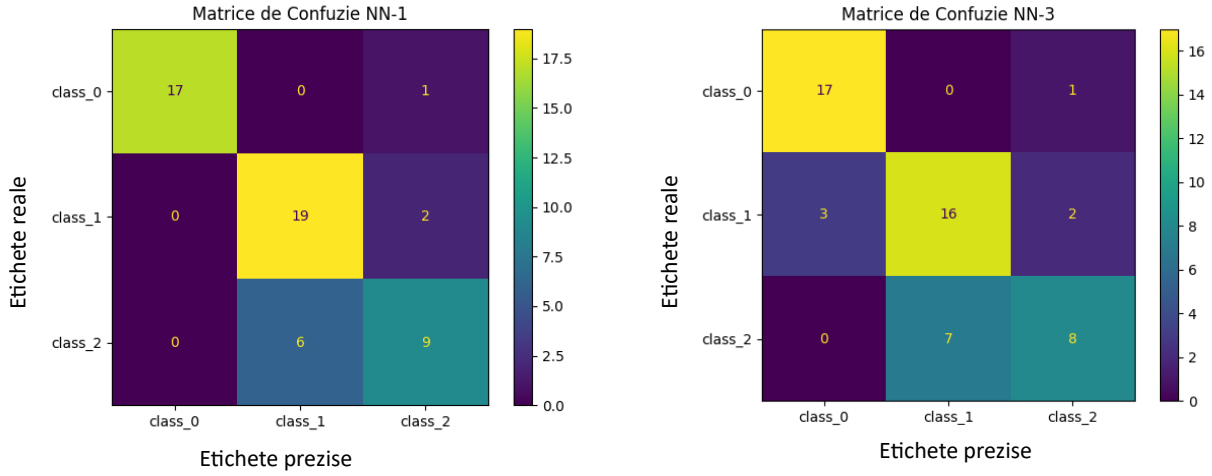
După aplicarea algoritmului PCA asupra întregului set de date, noile date sunt împărțite în două seturi: set de date de antrenare și set de date de test. Datele sunt împărțite în așa fel încât atât în setul de date de antrenare unde etichetele datelor sunt cunoscute cât și în setul de test unde etichetele trebuie aflate, numărul datelor din fiecare clasă se află în proporție egală. Astfel, se evită orice fel de afinitate a algoritmilor utilizați în determinarea cu o precizie mai mare a unei anumite clase față de o oricare alta. În continuare, sunt prezentate scorurile de clasificare pe lotul de test în funcție de procentul de date oferite spre antrenare.

Procent date antrenare/testare	Scor de clasificare Nearest Neighbor	Scor de clasificare 3-Nearest Neighbor	Scor de clasificare K-Means Supervizat
50/50	0.730	0.685	0.752
60/40	0.805	0.777	0.75
70/30	0.833	0.759	0.722
80/20	0.833	0.75	0.722
90/10	0.888	0.833	0.722

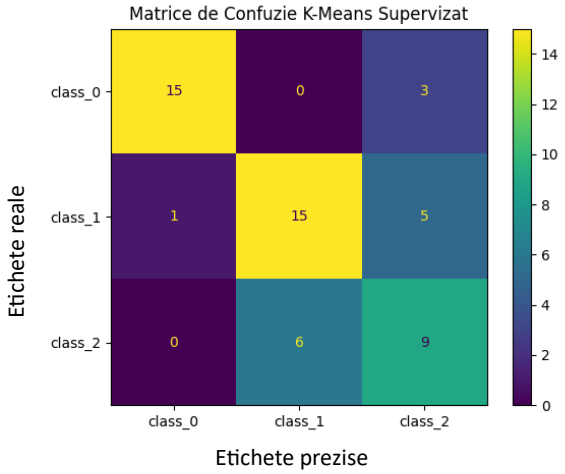
Tabel 2 Evoluția scorului de clasificare în funcție de împărțirea setului de date

Cu cât procentul de date de antrenare este mai mare, cu atât algoritmii realizează o clasificare mai bună, obținând un scor mai bun. Cu toate acestea, alegerea unui procent prea mare de date de antrenare va reduce semnificativ numărul de date disponibile pentru testarea capacităților de clasificare a algoritmilor, rezultatul final obținut fiind insuficient de general valabil. Prin urmare, se procedează în continuare la alegerea unei împărțiri a datelor setului de date în 70% date de antrenare, 30% date pentru testare, o practică destul de comună în domeniul algoritmilor cu învățare supervizată.

Pentru proporția 70% date de antrenare 30% date de testare au fost realizate matricele de confuzie ale fiecărui algoritm de clasificare non-neural studiat.



Figură 2 Matricea de confuzie pentru NN și 3-NN



Figură 3 Matrice de confuzie pentru algoritmul K-means Supervizat

Se poate observa tendința de a clasifica eronat vinurile ce provin din cea de a treia clasă, „class_2”, fiind într-un număr mai redus de exemple. Clasificatorul K-means Supervizat are matricea de confuzie ce mai echilibrată, clasificând clasele 0 și 1 în mod egal de corect. Nearest Neighbor clasifica cel mai bine clasa 1, având și cele mai multe eșantioane clasificate corect (conform rezultatelor anterioare), iar 3-Nearest Neighbor clasifică cel mai bine clasa 0, însă la fel de bine precum Nearest Neighbor (17 eșantioane identificate corect).

În final, voi măsura timpul necesar testării algoritmilor propuși, în funcție de numărul de valori proprii reținute. Timpul de antrenare este, în general, destul de mic fiind necesar calculul cel mult al fiecărui centroid corespondent fiecărei clase, în cazul K-Means. În cazul celorlalți algoritmi este suficientă cunoașterea etichetei fiecărui vector de intrare, fără vreo altă preprocesare anterioară.

Număr valori proprii reținute	Timp necesar testării Nearest Neighbor	Timp necesar testării 3-Nearest Neighbor	Timp necesar testării K-Means Supervizat
1	0.002 s	0.0019 s	0.116 s
2	0.002 s	0.0019 s	0.0015 s
3	0.0009 s	0.002 s	0.0025 s
4	0.002 s	0.002 s	0.0019 s
5	0.002 s	0.001 s	0.001 s
6	0.0009 s	0.001 s	0.002 s
7	0.0015 s	0.002 s	0.002 s
8	0.001 s	0.0026 s	0.0019 s
9	0.002 s	0.001 s	0.0023 s
10	0.0019 s	0.0035 s	0.002 s
11	0.0019 s	0.0015 s	0.002 s
12	0.001 s	0.0025 s	0.0019 s
13	0.002 s	0.002 s	0.0015 s

Tabel 3 Evoluția timpului necesar rulării algoritmilor pe setul de test în funcție de numărul de valori proprii reținute

Conform tabelului obținut, timpul de rulare al algoritmilor pe 30% din setul de date (reprezentând aproximativ 53 de eșantioane de test din totalul de 178 de eșantioane ale întregului set de date) rulează în aproximativ 0.001 secunde, indiferent de implementarea aleasă. Ținând cont de valorile optime de valori proprii reținute pentru fiecare algoritm, notate în Tabel 1, algoritmul Nearest Neighbor este cel mai eficient computațional, durând doar 0.0009 secunde pentru trei valori proprii reținute. 3-Nearest Neighbor oferă un timp aproximativ egal pentru șase valori proprii reținute, iar K-Means supervizat oferă cel mai slab timp la reținerea unei singure valori proprii. Cu toate acestea, majoritatea timpilor de execuție sunt de ordinul milisecundelor, prin urmare și diferențele acestora sunt insesizabile în cazul aplicației de față. Rezultatele au fost obținute pe un laptop cu procesor Intel Core i5-10500H și 16 GB de memorie internă RAM.

3 Concluzii

Utilizând algoritmul Nearest Neighbor se obține cel mai mare scor de clasificare păstrând și cele mai puține valori proprii. Astfel, procesele de antrenare și testare al algoritmului sunt mult mai rapide fiind necesare mult mai puține calcule.

Algoritmul K-means supervizat obține cea mai bună performanță la reținerea unei singure valori proprii, însă scorul de clasificare este mult mai slab decât cel obținut în cazul algoritmului Nearest Neighbor. Păstrarea a încă două valori proprii face astfel să merite efortul computațional suplimentar, îmbunătățind performanța de clasificare cu 0,093.

Algoritmul 3-Nearest Neighbor reușește să atingă aceleași performanțe precum algoritmul Nearest Neighbor însă pentru 6 valori proprii reținute, de două ori mai multe decât în cazul NN. Prin urmare, algoritmul nu justifică reținerea a mai multor valori proprii și deciderea clasei pe baza primilor 3 vecini ai fiecărei date de intrare a setului de date prezentat. Cu toate acestea, în cazul reținerii a una, respectiv două valori proprii, algoritmul obține un scor mult mai bun decât NN.

Anexă

1. Implementarea algoritmului PCA:

```
import numpy as np

# calculul prototipului unei clase
def prototype(data):
    mean = np.zeros(data[0].shape)
    for vector in data:
        mean += vector

    return mean/len(data)

# calculul matricii de covariatie
def covmatrix(data, mean_vector):
    cov = np.zeros((13, 13))
    for x in data:
        cov += np.matmul(np.atleast_2d(x-mean_vector).T, np.atleast_2d(x-
mean_vector))
    return cov/len(data)

# calculul matricii K
def K_Matrix(cov, eigenNum): # (matrice covariatie, numar valori proprii
retinute)

    pcaMatrix = []
    eigenValues, eigenVectors = np.linalg.eig(cov)

    idx = eigenValues.argsort()[::-1]
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:, idx]
    # print(eigenValues)
    # print(eigenVectors[0])
    etha = 0

    if eigenNum is None: # daca nu se defineste numarul de valori proprii
retinute
        for i in range(len(eigenValues)):
            etha += ((float(eigenValues[i])) / (float(np.sum(eigenValues))))
* 100.0
            print(etha)

            pcaMatrix.append(eigenVectors[i].tolist())
            if etha > 90.0: # se alege o matrice K ce retine peste 90% din
informatia de baza
                return np.array(pcaMatrix)
    else: # altfel se retin atatea valori proprii cate au fost precizate
        for i in range(eigenNum):
            etha += ((float(eigenValues[i])) / (float(np.sum(eigenValues))))
* 100.0
            pcaMatrix.append(eigenVectors[i].tolist())
        print(etha)
```

```
    return np.array(pcaMatrix)

# aplicarea transformarii PCA
def PCA(data, K):
    # print(np.atleast_2d(data).shape)
    # print(K.shape)
    principalComponents = []
    for element in data:
        principalComponents.append(np.matmul(np.atleast_2d(element),
K.T).flatten().tolist())

    return np.array(principalComponents)
```

2. Codul principal pentru antrenarea și testarea algoritmilor:

```
# from ucimlrepo import fetch_ucirepo
import numpy as np
import matplotlib.pyplot as plt
import PCA
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier, NearestCentroid
from sklearn.datasets import load_wine
import time


# preluarea setului de date de la uci - trecut la sklearn
# wine = fetch_ucirepo(id=109)
data = load_wine()

# data (as pandas dataframes)
# X = wine.data.features
X = data.data
# y = wine.data.targets
y = data.target

# metadatele bazei de date uci
# print(wine.metadata)
counts, bins = np.histogram(y)

plt.figure('Histograma Claselor'), plt.hist(bins[:-1], bins, weights=counts),
plt.title("Histograma Claselor")
# informatiile variabilelor
# print(wine.variables)
# 178 de vectori - vinuri
# print(X.shape) # 13 caracteristici ale vinului
# print(y.shape) # 1 eticheta pentru fiecare vin - 3 clase (tipuri) de vinuri

# datele de intrare convertite ca numpy array
Xnp = np.array(X)

# calculul matricii de covariatie
# print(Xnp)
```

```
# print(prototype(Xnp))

# implementare proprie
# cov1 = covmatrix(Xnp, prototype(Xnp))
# implementare numpy
cov2 = np.cov(Xnp, rowvar=False, bias=True)

# print(cov1.shape)
# print(cov2.shape)
# print(covmatrix(Xnp, prototype(Xnp)))
# print(np.cov(X, rowvar=False))
# print(abs(cov1 - cov2))

# evaluarea algoritmilor in functie de numarul de valori proprii retinute

for numValues in range(1, 14):
    K = PCA.K_Matrix(cov2, numValues)  # calcul matrice K
    # print(K)
    # print(K.shape)
    # print(Xnp.shape)
    X_comp = PCA.PCA(Xnp, K)
    Y_comp = np.array(y)
    # print(X_comp.shape)

    X_train, X_test, Y_train, Y_test = train_test_split(X_comp, Y_comp,
test_size=0.3, random_state=19, stratify=Y_comp)

    NN1 = KNeighborsClassifier(n_neighbors=1, metric='euclidean')
    NN3 = KNeighborsClassifier(n_neighbors=3, metric='euclidean')
    K_Means_supervizat = NearestCentroid()

    t1 = time.time()
    NN1.fit(X_train, np.ravel(Y_train))
    t2 = time.time()
    print(f"Time Train NN: {t2 - t1}")

    t1 = time.time()
    NN3.fit(X_train, np.ravel(Y_train))
    t2 = time.time()
    print(f"Time Train 3-NN: {t2 - t1}")

    t1 = time.time()
    K_Means_supervizat.fit(X_train, np.ravel(Y_train))
    t2 = time.time()
    print(f"Time Train K-Means: {t2 - t1}")

    print(f"Pentru {numValues} valori proprii")
    t1 = time.time()
    print(f"Scor clasificare NN:{NN1.score(X_test, np.ravel(Y_test))}")
    t2 = time.time()
    print(f"Time Test NN: {t2-t1}")
    t1 = time.time()
    print(f"Scor clasificare 3-NN:{NN3.score(X_test, np.ravel(Y_test))}")
    t2 = time.time()
    print(f"Time Test 3-NN: {t2 - t1}")
    t1 = time.time()
    print(f"Scor clasificare K-Means Supervizat:
```

```
{K_Means_supervizat.score(X_test, np.ravel(Y_test))}")
    t2 = time.time()
    print(f"Time Test K-Means: {t2 - t1}")

# calculul matricii PCA si aplicarea transformarii
K = PCA.K_Matrix(cov2, 3)  # calcul matrice K
# print(K)
# print(K.shape)
# print(Xnp.shape)
X_comp = PCA.PCA(Xnp, K)
Y_comp = np.array(y)
# print(X_comp.shape)

# impartirea setului de date
X_train, X_test, Y_train, Y_test = train_test_split(X_comp, Y_comp,
test_size=0.3, random_state=19, stratify=Y_comp)

# definirea clasificatorilor
NN1 = KNeighborsClassifier(n_neighbors=1, metric='euclidean')
NN3 = KNeighborsClassifier(n_neighbors=3, metric='euclidean')
K_Means_supervizat = NearestCentroid()

# antrenarea si testarea algoritmilor
NN1.fit(X_train, np.ravel(Y_train))
NN3.fit(X_train, np.ravel(Y_train))
K_Means_supervizat.fit(X_train, np.ravel(Y_train))

print(f"Pentru {3} valori proprii")
print(f"Scor clasificare NN:{NN1.score(X_test, np.ravel(Y_test))}")
print(f"Scor clasificare 3-NN:{NN3.score(X_test, np.ravel(Y_test))}")
print(f"Scor clasificare K-Means Supervizat:
{K_Means_supervizat.score(X_test, np.ravel(Y_test))}")

# testarea algoritmilor si afisarea predictiilor facute
Y_1 = NN1.predict(X_test)
Y_3 = NN3.predict(X_test)
Y_K = K_Means_supervizat.predict(X_test)

plt.figure('NN'), plt.plot(X_test, Y_1, 'or'), plt.plot(X_test, Y_test,
'^g'), plt.ylabel("Clasa"),
plt.xlabel("Valoare intrare"), plt.legend(["NN Predictii", "Real"]),
plt.title("Esantioane NN-1")
plt.figure('3-NN'), plt.plot(X_test, Y_3, 'or'), plt.plot(X_test, Y_test,
'^g'), plt.ylabel("Clasa"),
plt.xlabel("Valoare intrare"), plt.legend(["3-NN Predictii", "Real"]),
plt.title("Esantioane NN-3")
plt.figure('K-Means Supervizat'), plt.plot(X_test, Y_K, 'or'),
plt.plot(X_test, Y_test, '^g'), plt.ylabel("Clasa"),
plt.xlabel("Valoare intrare"), plt.legend(["K-Means Supervizat - Predictii",
"Real"]),
plt.title("Esantioane K-Means Supervizat")

# calculul si afisarea matricii de confuzie pentru fiecare algoritm
conf_NN1 = confusion_matrix(Y_test, Y_1)
conf_NN3 = confusion_matrix(Y_test, Y_3)
conf_K = confusion_matrix(Y_test, Y_K)
```

```
disp_1 = ConfusionMatrixDisplay(confusion_matrix=conf_NN1,
display_labels=data.target_names)
disp_1.plot()
plt.title("Matrice de Confuzie NN-1")
disp_3 = ConfusionMatrixDisplay(confusion_matrix=conf_NN3,
display_labels=data.target_names)
disp_3.plot()
plt.title("Matrice de Confuzie NN-3")
disp_K = ConfusionMatrixDisplay(confusion_matrix=conf_K,
display_labels=data.target_names)
disp_K.plot()
plt.title("Matrice de Confuzie K-Means Supervizat")
plt.show()
```