Dry Beans

Gabriele Galilei s256349

21 settembre 2021

Indice

1 Introduzione al Dataset

Lo scopo della seguente trattazione è approfondire l'applicazione di alcuni dei metodi studiati durante il corso di matematica per l'Intelligenza Artificiale ad un insieme di dati reale, per poter analizzare in modo diverso il comportamento delle variabili. La scelta è ricaduta su un dataset contenente dati riguardo ad alcune specie di fagiolo in forma secca. Su tale dataset si svolgeranno vari algoritmi di analisi e classificazione.

[1]:
```
import pandas as pd
import scipy as sp
import numpy as np

import sklearn
from sklearn import svm
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA,␣
 ↪QuadraticDiscriminantAnalysis as QDA
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.multiclass import OneVsRestClassifier
from mlxtend.plotting import plot_decision_regions

import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import itertools

from mpl_toolkits.mplot3d import Axes3D
from IPython.display import display
from linear_r2 import generate_square, HyperplaneR2
from FisherDA import MultipleFisherDiscriminantAnalysis as MDA
import cycler

import warnings
warnings.filterwarnings('ignore')
```

1.1 Importazione del dataset

Il dataset è stato creato da:

Murat KOKLU Faculty of Technology, Selcuk University, TURKEY. ORCID : 0000-0002-2737-2360 mkoklu@selcuk.edu.tr

e

Ilker Ali OZKAN Faculty of Technology, Selcuk University, TURKEY. ORCID : 0000-0002-5715-1040 ilkerozkan@selcuk.edu.tr

[2]:
```
beans = pd.read_csv('datasets/beans/Dry_Bean_Dataset.csv', header=None)
col_names = beans.loc[0,:]
beans.drop(0, inplace=True)
rename_dict = {k: col_names[k] for k in range(17)}
beans.rename(columns=rename_dict, inplace=True)
beans
```

[2]:

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation
1	28395	610.291	208.17811670853	173.88874704164	1.197191424116
2	28734	638.018	200.52479566365	182.73441935102	1.0973564606811
3	29380	624.11	212.82612986021	175.93114261271	1.2097126563244
4	30008	645.884	210.55799896093	182.5165156953	1.1536380593219
5	30140	620.134	201.84788216674	190.27927878665	1.0607980199098
...	...	...	...	...	...
13607	42097	759.696	288.72161204222	185.94470544011	1.552728330494
13608	42101	757.499	281.57639233416	190.71313645036	1.4764394187784
13609	42139	759.321	281.53992791426	191.18797890119	1.4725817466786
13610	42147	763.779	283.38263637995	190.27573077099	1.4893262279519
13611	42159	772.237	295.14274098853	182.20471589551	1.6198413939943

	Eccentricity	ConvexArea	EquivDiameter	Extent
1	0.54981218713835	28715	190.14109727451	0.76392251815981
2	0.41178525136724	29172	191.27275048584	0.78396813270763
3	0.56272731675041	29690	193.41090409881	0.7781132475237
4	0.49861597640272	30724	195.46706182478	0.78268127282212
5	0.33367965777318	30417	195.89650297623	0.77309803519212
...	...	...	...	...
13607	0.76500220104266	42508	231.51579884474	0.71457428028246
13608	0.73570221827149	42494	231.5267977425	0.79994299828995
13609	0.73406478120058	42569	231.63126122265	0.7299324441365
13610	0.74105478696066	42667	231.65324753163	0.70538912133891
13611	0.78669301640239	42600	231.68622308305	0.7889624971929

	Solidity	roundness	Compactness
1	0.988855998607	0.95802712625013	0.91335775479576
2	0.98498560263266	0.88703363655289	0.95386084226051
3	0.98955877399798	0.94784947301641	0.90877423851033
4	0.97669574274183	0.90393637431734	0.92832883475994
5	0.99089325048493	0.98487706935091	0.97051552324147
...	...	...	...
13607	0.99033123176814	0.91660312156215	0.8018651503334
13608	0.99075163552502	0.92201534243912	0.82225216334092
13609	0.9898987526134	0.91842409110677	0.82272970281214
13610	0.9878125952141	0.90790645653873	0.81745745078411
13611	0.98964788732394	0.88838036856664	0.78499719256879

	ShapeFactor1	ShapeFactor2	ShapeFactor3

```
1      0.0073315061351832  0.0031472891673357  0.83422238824556
2      0.0069786592769419   0.003563623712101  0.90985050639794
3      0.0072439118400343  0.0030477332172132  0.82587061658003
4      0.0070167288376741  0.0032145620793901  0.86179442544675
5      0.0066970100254392  0.0036649719644517  0.94190038085267
...                   ...                 ...               ...
13607  0.0068584842635394  0.0017490944660096  0.64298771931921
13608  0.0066881164897309  0.0018858349953099  0.67609862011883
13609  0.0066812199604703  0.0018882706374544  0.67688416388935
13610  0.0067236727733872  0.0018520254836679  0.66823668384245
13611  0.0070007054481493  0.0016398117055199  0.61622059234089

           ShapeFactor4     Class
1      0.99872388901317     SEKER
2      0.99843033144971     SEKER
3      0.99906613736327     SEKER
4      0.99419884850682     SEKER
5      0.99916605896078     SEKER
...                 ...       ...
13607  0.99838524794667  DERMASON
13608   0.9982186537251  DERMASON
13609  0.99676726435924  DERMASON
13610  0.99522241972615  DERMASON
13611  0.99817962330547  DERMASON

[13611 rows x 17 columns]
```

Come possiamo vedere, il dataset contiene immagini ad alta risoluzione di 13.611 chicchi di 7 diverse specie di fagiolo registrate. Ogni chicco è classificato attraverso 16 attributi, 12 riguardanti la dimensione e 4 riguardanti la forma.

Attributi:

1. **Area (A)**: L'area del fagiolo e numero di pixel interni al suo perimetro.
2. **Perimeter (P)**: Circonferenza del fagiolo definita come la lunghezza del suo bordo.
3. **Major axis length (L)**: Il segmento di lunghezza massima tra due punti del bordo del fagiolo
4. **Minor axis length (l)**: Il segmento di lunghezza massima tra due punti del bordo del fagiolo tra quelle perpendicolari a Major axis.
5. **Aspect ratio (K)**: Definisce il rapporto tra L e l.
6. **Eccentricity (Ec)**: Eccentricità dell'ellisse avente gli stessi momenti della regione.
7. **Convex area (C)**: Numero di pixel del più piccolo poligono convesso che circoscrive il fagiolo.
8. **Equivalent diameter (Ed)**: Il diametro del cerchio avente stessa area del fagiolo.
9. **Extent (Ex)**: Il rapporto tra i pixel dell'area del fagiolo e i pixel totali dell'inquadratura.

10. **Solidity (S)**: Anche detta convessità. Il rapporto tra i pixel nell'area del fagiolo e quelli nel guscio convesso.
11. **Roundness (R)**: Calcolato con la seguente formula: $\frac{4\pi A}{P^2}$.
12. **Compactness (CO)**: Misura della rotondità del fagiolo: $\frac{Ed}{L}$.
13. **ShapeFactor1 (SF1)**
14. **ShapeFactor2 (SF2)**
15. **ShapeFactor3 (SF3)**
16. **ShapeFactor4 (SF4)**
17. **Class**: le 7 diverse specie di fagiolo: Seker, Barbunya, Bombay, Cali, Dermosan, Horoz e Sira.

1.2 Analisi del dataset

Chiamiamo M e N le dimensioni del dataset.

[3]:
```
beans = beans.astype('category')
```

[4]:
```
M, N = beans.shape
```

Stampiamo a video alcune informazioni sul database.

[5]:
```
beans.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 13611 entries, 1 to 13611
Data columns (total 17 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Area             13611 non-null  category
 1   Perimeter        13611 non-null  category
 2   MajorAxisLength  13611 non-null  category
 3   MinorAxisLength  13611 non-null  category
 4   AspectRation     13611 non-null  category
 5   Eccentricity     13611 non-null  category
 6   ConvexArea       13611 non-null  category
 7   EquivDiameter    13611 non-null  category
 8   Extent           13611 non-null  category
 9   Solidity         13611 non-null  category
 10  roundness        13611 non-null  category
 11  Compactness      13611 non-null  category
 12  ShapeFactor1     13611 non-null  category
 13  ShapeFactor2     13611 non-null  category
 14  ShapeFactor3     13611 non-null  category
 15  ShapeFactor4     13611 non-null  category
 16  Class            13611 non-null  category
```

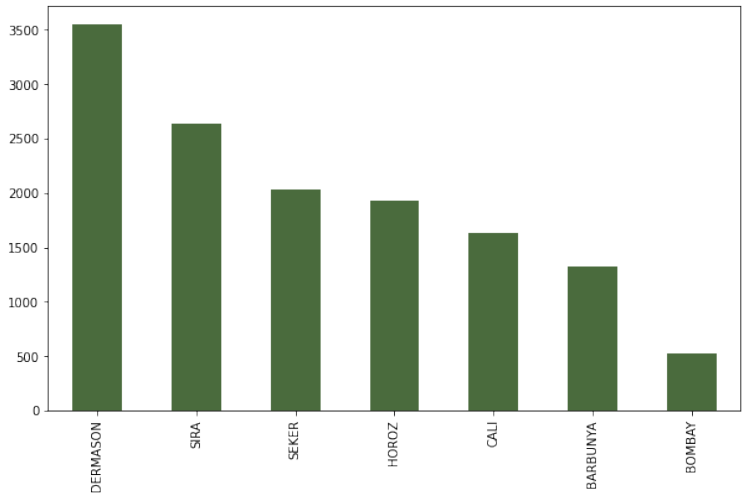
```
dtypes: category(17)
memory usage: 9.5 MB
```

Da queste possiamo vedere, come già dichiarato nel README, che il database non ha dati in Null e quindi è completo.

Ora andiamo a riportare il numero di istanze per classe e la frequenza di ogni classe sul totale. Successivamente stampiamo a video un istogramma che riporta le frequenze delle classi.

[6]:
```
class_cont_freq = pd.concat([beans['Class'].value_counts(), beans['Class'].
 ↪value_counts()/M], axis=1)
class_cont_freq.columns = ['counts', 'freq.']
class_cont_freq.index.name = 'Class'

display(class_cont_freq);
beans['Class'].value_counts().plot.bar(figsize=(10, 6), color=[(68/235,99/235,56/
 ↪235)]);
```

Class	counts	freq.
DERMASON	3546	0.260525
SIRA	2636	0.193667
SEKER	2027	0.148924
HOROZ	1928	0.141650
CALI	1630	0.119756
BARBUNYA	1322	0.097127
BOMBAY	522	0.038351



Possiamo quindi vedere come, almeno nella popolazione studiata dal dataset, la specie più comune è la Dermason e la meno comune è la Bombay.

1.3 Label Encoder

Per i prossimi algoritmi si crea un database di copia sul quale operare. Successivamente si cambia il datatype dell'attributo classe da char a int per poter eseguire algoritmi numerici sul dataset. Per far questo usiamo il Label Encoder. Nel nostro caso abbiamo 7 classi che saranno classificate dal LE come gli interi da 0 a 6.

[7]:
```
#beans_work = beans.sample(3000).copy()
beans_work = beans.copy()
beans_original = beans.copy()
beans = beans_work

le = LabelEncoder()
le.fit(beans['Class'])
beans['Class']=le.transform(beans['Class'])
Beans_codes = {'SEKER':0,'BARBUNYA':1, 'BOMBAY':2, 'CALI':3, 'HOROZ':4, 'SIRA':
 ↪5, 'DERMASON':6} #dizionario creato per visualizzare le legende
```

1.4 Controlli: Normality Test e Homoscedasticity Test

Controlliamo ora che al dataset in analisi siano applicabili i metodi di classificazione studiati. Analizziamo prima la normalità. Essendo il dataset abbastanza grande l'attesa è di p-value molto bassi, purtroppo. Infatti riteniamo accettabile l'ipotesi di normalità per p-value superiori a 0.05.

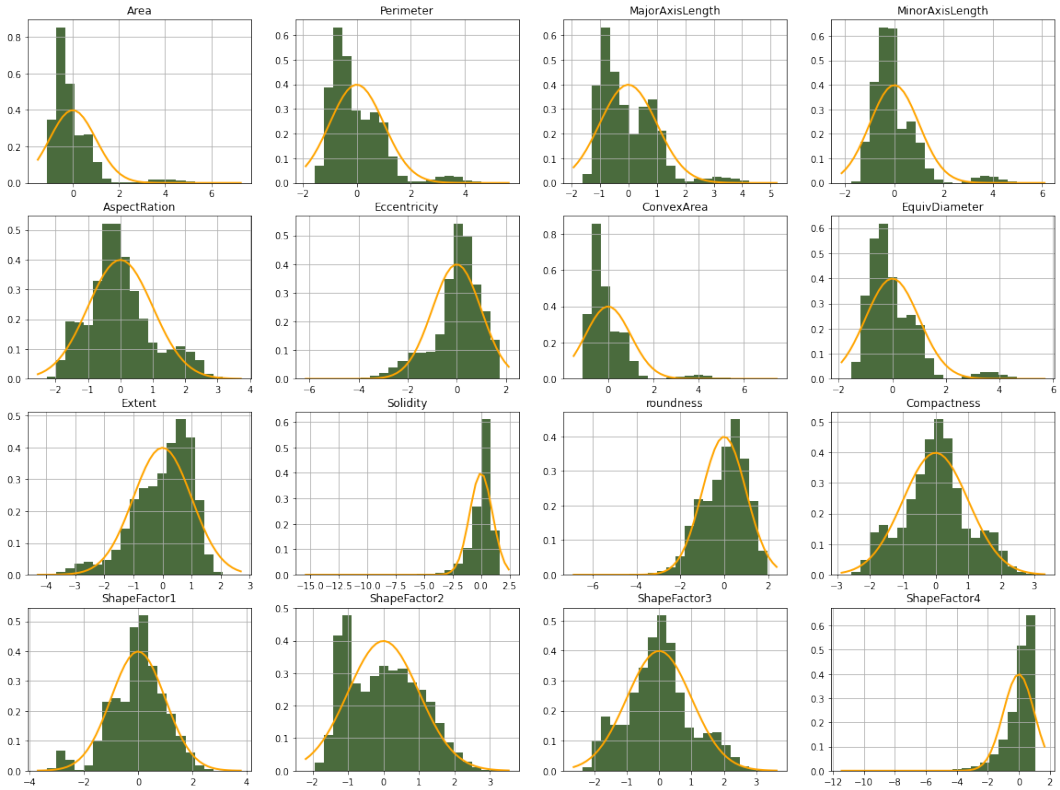
[8]:
```
X_beans = beans.iloc[:, :-1]
scaler_beans = StandardScaler()
scaler_beans.fit(X_beans.values)
X_beans_scaled = scaler_beans.transform(X_beans.values)
beans_scaled_use = pd.DataFrame(X_beans_scaled, columns=col_names[:-1])

normality = pd.DataFrame(index=['K-value','p-value'], columns=col_names[:-1])
for i in range(16):
    out = sp.stats.normaltest(beans_scaled_use.iloc[:,i])
    normality.iloc[0,i] = out[0]
    normality.iloc[1,i] = out[1]
normality
```

[8]:
```
0              Area    Perimeter MajorAxisLength MinorAxisLength  \
K-value  8520.242038  4276.327816      3302.36807     6392.690185
p-value           0.0          0.0             0.0             0.0

0       AspectRation Eccentricity ConvexArea EquivDiameter       Extent  \
K-value   673.613333  2173.078145  8491.0971    5440.13145  1499.118342
p-value          0.0          0.0        0.0           0.0          0.0

0           Solidity   roundness Compactness ShapeFactor1 ShapeFactor2  \
K-value  7930.056057  833.280887   38.993005   734.197341  1729.002733
p-value          0.0         0.0         0.0          0.0          0.0
```

```
0        ShapeFactor3 ShapeFactor4
K-value    143.399266  8388.387164
p-value           0.0          0.0
```

Come prospettato il normality test non ha dato i risultati sperati; quindi si passa ad una visualizzazione diretta.

[9]:
```
normaltest = plt.figure(figsize=(20, 15))
for j in range (len(col_names)-1):
    normaltest.add_subplot(4,4,j+1)
    mu, std = sp.stats.norm.fit(beans_scaled_use.iloc[:,j])

    plt.hist(beans_scaled_use.iloc[:,j], bins=20, density=True, alpha=1,␣
 ↪color=[(68/235,99/235,56/235)])

    xmin, xmax = plt.xlim()
    x = np.linspace(xmin, xmax)
    p = sp.stats.norm.pdf(x, mu, std)

    plt.plot(x, p, 'orange', linewidth=2)
    title = str(col_names[j]).format(mu, std)
    plt.grid()
    plt.title(title)
plt.show()
```



Possiamo quindi vedere come gli attributi siano distribuiti quasi normalmente e quindi possiamo ritenerli accettabili. Lo stesso discorso può essere fatto per l'omoschedasticità.

[10]:
```
outhom = sp.stats.levene(beans_scaled_use.iloc[:,0], beans_scaled_use.iloc[:,1],␣
↪beans_scaled_use.iloc[:,2],
                         beans_scaled_use.iloc[:,3], beans_scaled_use.iloc[:,4],␣
↪beans_scaled_use.iloc[:,5],
                         beans_scaled_use.iloc[:,6], beans_scaled_use.iloc[:,7],␣
↪beans_scaled_use.iloc[:,8],
                         beans_scaled_use.iloc[:,9], beans_scaled_use.iloc[:
↪,10], beans_scaled_use.iloc[:,11],
                         beans_scaled_use.iloc[:,12], beans_scaled_use.iloc[:
↪,13], beans_scaled_use.iloc[:,14],
                         beans_scaled_use.iloc[:,15], center='mean')

homoscedasticity = pd.DataFrame(index=['K-value','p-value'], columns=['Result'])
homoscedasticity.iloc[0,0] = outhom[0]
homoscedasticity.iloc[1,0] = outhom[1]
homoscedasticity
```

[10]:

	Result
K-value	115.7552
p-value	0.0

DISCLAIMER: ho provato ad usare Wine Quality (sia rosso che bianco), Page blocks, Magic, Hepatitis C Virus e Dry Beans, ma in tutti i casi i test davano esito negativo. Ho deciso quindi di usare questo dataset in quanto quello che "rispetta" maggiormente le condizioni.

2 PCA - Principal Component Analysis

La PCA è un algoritmo di data representation non supervisionato che ha come obiettivo l'individuazione, nell'iperspazio degli attributi, delle direzioni in cui i dati presentano la massima varianza per la riduzione della dimensionalità del dataset d-dimensionale proiettandolo in un sottospazio k-dimensionale (con $k < d$). È lecito chiedersi quale sia un valore di k tale che si ottenga una buona mole di informazioni. Analizziamo l'algoritmo per step:

- standardizzazione dei dati (std = 1, mean = 0);
- calcolo degli autovalori e dei relativi autovettori della matrice di covarianza;
- disposizione decrescente degli autovalori e selezione dei primi k autovettori (associati quindi ai k autovalori più grandi) dove k è la nuova dimensionalità;
- costruzione della projection matrix B;
- trasformazione del data set tramite la matrice B per ottenere il nuovo sottospazio.

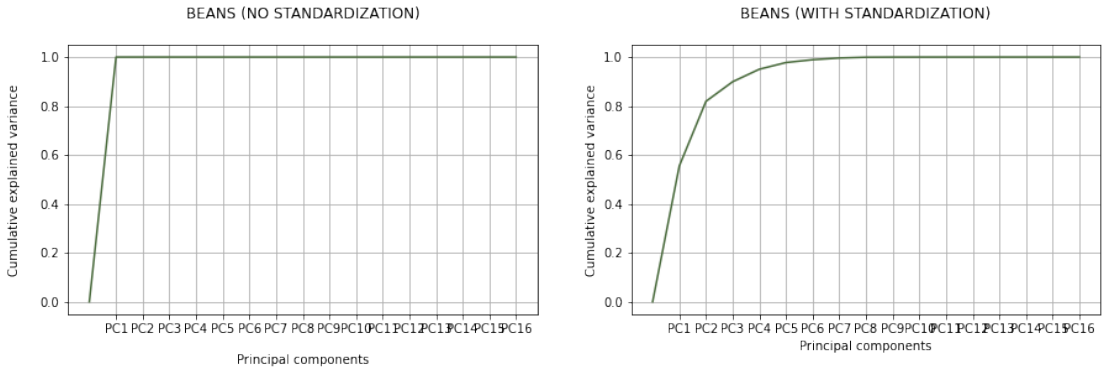
Quindi al dataset vengono rimossi i dati relativi alla classe, cioè la colonna dei target, e successivamente lo **Standard Scaler** normalizza le colonne in modo tale che queste abbiano media 0 e varianza unitaria. Questo passaggio è fondamentale perché nel dataset in analisi gli attributi hanno unità di misura non paragonabili e ampiezza del sottocampione molto diversa.

Si procede calcolando autovalori e autovettori della matrice di covarianza. Si ordinano in modo decrescente gli autovalori: questi rappresentano la quantità di varianza, detta **varianza spiegata**, nella direzione del corrispondente autovettore, detto **competente principale**.

2.1 PCA: Standard Scaler

[11]:
```
# si ricorda che lo scaler è già stato usato nei controlli
pca_beans = PCA()
pca_beans_nostd = PCA()

pca_beans.fit(X_beans_scaled)
pca_beans_nostd.fit(X_beans.values)

fig_stand = plt.figure(figsize=(15, 4));
nostand = fig_stand.add_subplot(1, 2, 1);
plt.plot(np.insert(np.cumsum(pca_beans_nostd.explained_variance_ratio_), 0, 0),␣
 ↪color=(68/235,99/235,56/235))
nostand.set_title('\nBEANS (NO STANDARDIZATION)\n')
plt.xticks(ticks=np.arange(1, pca_beans_nostd.n_features_ + 1),
           labels=[f'PC{i}' for i in range(1, pca_beans_nostd.n_features_ + 1)])
plt.xlabel('\nPrincipal components\n')
plt.ylabel('\nCumulative explained variance\n')
plt.grid()

stand = fig_stand.add_subplot(1, 2, 2);
plt.plot(np.insert(np.cumsum(pca_beans.explained_variance_ratio_), 0, 0),␣
 ↪color=(68/235,99/235,56/235))
```

```
stand.set_title('\nBEANS (WITH STANDARDIZATION)\n')
plt.xticks(ticks=np.arange(1, pca_beans.n_features_ + 1),
           labels=[f'PC{i}' for i in range(1, pca_beans.n_features_ + 1)])
plt.xlabel('Principal components')
plt.ylabel('Cumulative explained variance')
plt.grid()
plt.show()
```



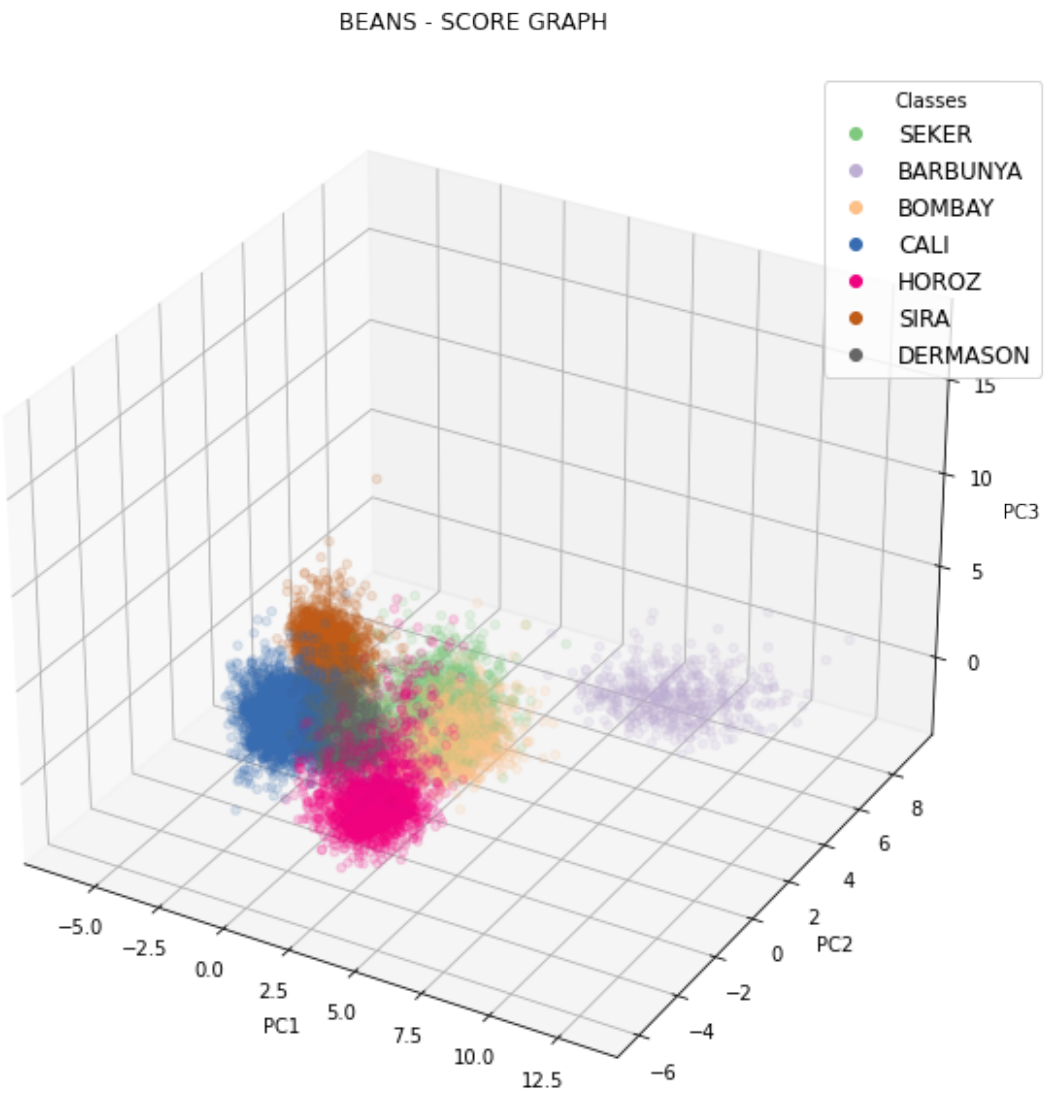
Da questo grafico si vede chiaramente come bastino sole due componenti principali per spiegare più dell'80% della varianza totale; per questo uno spazio bidimensionale è più che sufficiente per visualizzare la separazione delle classi. In 3 dimensioni la varianza spiegata cumulata arriva a circa il 90% ed è questo l'esempio riportato.

2.2 PCA: Score Graph

[12]:
```
pca_beans_m = PCA(n_components=3)
pca_beans_m.fit(X_beans_scaled)
Y_beans_m = pca_beans_m.transform(X_beans_scaled)
```

[13]:
```
fig_beansscore = plt.figure(figsize=[10,10])
ax = fig_beansscore.add_subplot(111, projection='3d')
plt.set_cmap('Accent')
scatter = ax.scatter(Y_beans_m[:, 0], Y_beans_m[:, 1], Y_beans_m[:, 2],␣
 ↪c=beans['Class'].values, alpha=1.00)
ax.legend(handles=scatter.legend_elements()[0], labels=Beans_codes.keys(),␣
 ↪bbox_to_anchor=(1.05, 1), fontsize='large', title="Classes")
plt.title('\nBEANS - SCORE GRAPH\n')
ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_zlabel('PC3')
scatter.set_alpha(0.15);
plt.grid()
```

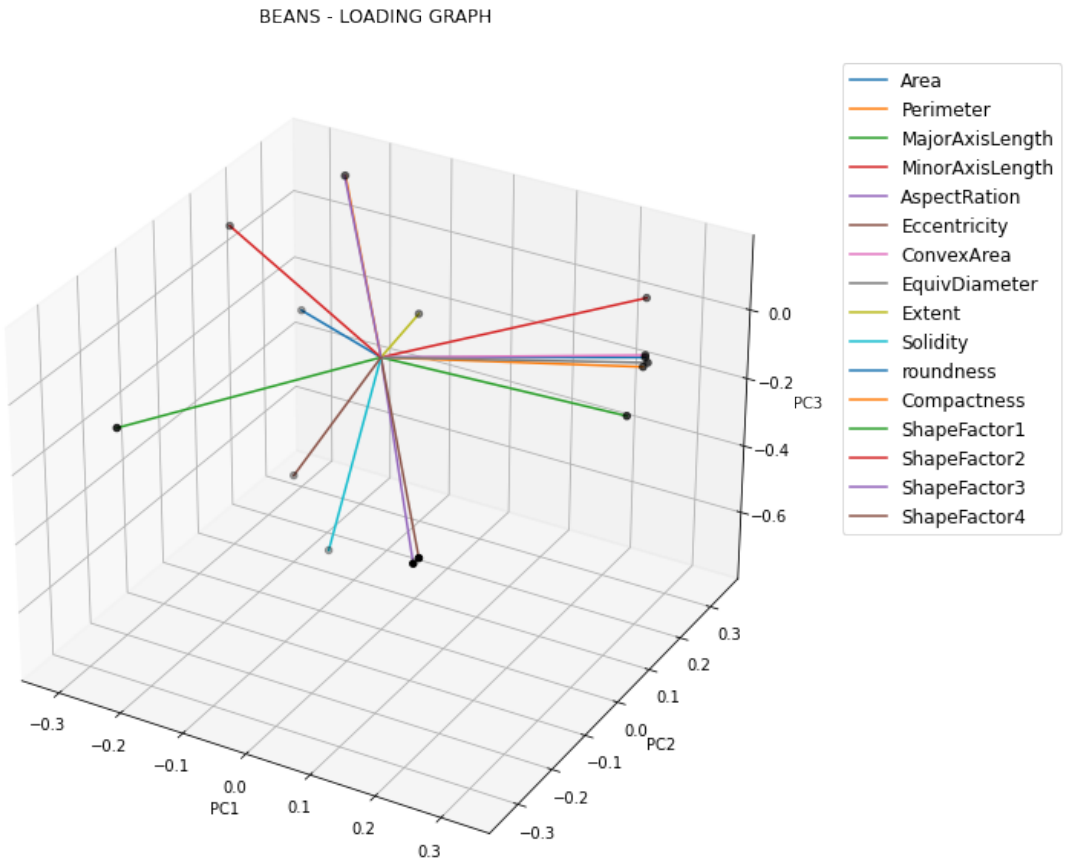
```
plt.show()
```



2.3 PCA: Loading Graph

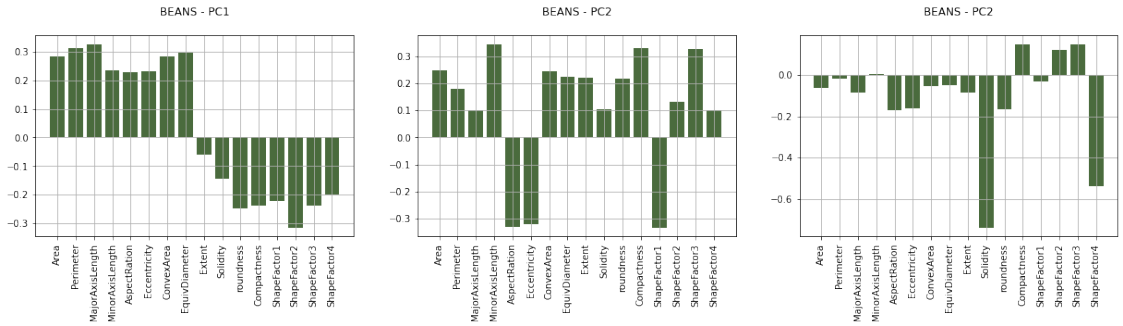
Resta da vedere come gli attributi siano legati alle varie componenti principali e quindi capire quali attributi siano i più caratterizzanti o meglio "classificanti". Per questo si usa il **Loading Graph**. Più un vettore relativo ad un attributo è parallelo alla PC1 più questo pesa sulla classificazione: in questo caso lo ShapeFactor1 è quello più caratterizzante. Inoltre, più l'angolo tra due vettori è piccolo (ca. 0) più gli attributi relativi sono correlati positivamente (come ConvexArea e EquivDiameter), più è grande (ca. π) più gli attributi relativi sono correlati negativamente (come per AspectRation e ShapeFactor3), più sono vicini all'ortogonalità (ca. $\frac{\pi}{2}$) più sono vicini alla scorrelazione (come per Perimeter e Compactness).

[14]:
```
plt.rcParams['figure.figsize'] = [10, 10];
fig_beansscore = plt.figure();
ax = fig_beansscore.add_subplot(111, projection='3d');
for i in range(pca_beans_m.n_features_):
    ax.plot([0, pca_beans_m.components_[0, i]], [0, pca_beans_m.components_[1,␣
 ↪i]], [0, pca_beans_m.components_[2, i]], label=X_beans.columns[i]);
ax.scatter(pca_beans_m.components_[0, :], pca_beans_m.components_[1, :],␣
 ↪pca_beans_m.components_[2, :], c='k');
plt.legend(bbox_to_anchor=(1.05, 1), fontsize='large');
plt.title('\nBEANS - LOADING GRAPH\n');
ax.set_xlabel('PC1');
ax.set_ylabel('PC2');
ax.set_zlabel('PC3');
plt.grid();
plt.show();
```



Ora mostriamo un'interpretazione più chiara del loading graph che ci permette di vedere le componenti di ogni attributo per ogni componente principale.

[15]:
```
components = plt.figure(figsize=(21, 4));
pc1_comp = components.add_subplot(1, 3, 1);
plt.bar(np.arange(pca_beans_m.n_features_), pca_beans_m.components_[0, :],␣
 ↪color=[(68/235,99/235,56/235)])
plt.xticks(ticks=np.arange(pca_beans_m.n_features_),
           labels=X_beans.columns.to_list(),
           rotation=90)
plt.title('\nBEANS - PC1\n')
plt.grid()

pc1_comp = components.add_subplot(1, 3, 2);
plt.bar(np.arange(pca_beans_m.n_features_), pca_beans_m.components_[1, :],␣
 ↪color=[(68/235,99/235,56/235)])
plt.xticks(ticks=np.arange(pca_beans_m.n_features_),
           labels=X_beans.columns.to_list(),
           rotation=90)
plt.title('\nBEANS - PC2\n')
plt.grid()

pc3_comp = components.add_subplot(1, 3, 3);
plt.bar(np.arange(pca_beans_m.n_features_), pca_beans_m.components_[2, :],␣
 ↪color=[(68/235,99/235,56/235)])
plt.xticks(ticks=np.arange(pca_beans_m.n_features_),
           labels=X_beans.columns.to_list(),
           rotation=90)
plt.title('\nBEANS - PC2\n')
plt.grid()
plt.show()
```



Da qui possiamo vedere che mentre la PC1 sembra essere legata alla dimensione del fagiolo, la PC2 e la PC3 sembrano correlate alla sua forma, in particolare alla sua rotondità. Quindi possiamo dare i seguenti nomi:

- PC1 : Dimension;
- PC2 : Roundness1;
- PC3 : Roundness2.

3 MDA - Multiple Fischer Discriminant Analysis

Si parla di MDA come della versione multiclasse della FDA (Fischer Discriminant Analysis): sono algoritmi supervisionati di data classification. Nel caso in analisi si hanno 7 classi quindi la MDA può ridurre la dimensionalità fino a 6. Scopo della MDA è appunto quello di proiettare i dati in uno spazio di dimensione minore massimizzando lo spazio tra una classe e l'altra (tramite l'uso della beetween scatter matrix) e minimizzando lo spazio tra elementi della stessa classe (within scatter matrix).

3.1 MDA: basi teoriche

Sia $\{x_i\}_{i=1}^n$ l'insieme dei sample divisi in c classi con cardinalità n_i. Sia V la matrice di proiezione. Allora i punti proiettati sono $y_i = V^T x_i$.

Media totale: $\mu = \frac{1}{n} \sum_{i=1}^n x_i$

Media della classe j: $\mu_j = \frac{1}{n_j} \sum_{x_i \in classj} x_i$

Media totale delle proiezioni: $\tilde{\mu} = \frac{1}{n} \sum_{i=1}^n y_i$

Media delle proiezioni della classe j: $\tilde{\mu}_j = \frac{1}{n_j} \sum_{x_i \in classj} y_i$

Definiamo la **Within Scatter Matrix** come:

$$S_W = \sum_{j=1}^{c} [\sum_{x_k \in \ class\ j} (x_k - \mu_i)(x_k - \mu_i)^T]$$

Definiamo la **Between Scatter Matrix** come:

$$S_B = \sum_{j=1}^{c} n_j (\mu_i - \mu)(\mu_i - \mu)^T$$

Si dimostra che la proiezione voluta V è tale che massimizza la funzione obiettivo:

$$J(V) = \frac{det(V^T S_B V)}{det(V^T S_W V)}.$$

Da qui si dimostra che basta risolvere il problema agli autovalori generalizzati:

$$S_B v = \lambda S_W v$$

dal quale ricaviamo al massimo c-1 autovalori e corrispondenti autovettori. Ordiniamo gli autovalori in ordine descrescente e prendendo i k autovalori più grandi, i relativi autovettori formeranno la matrice di proiezione cercata su uno spazio k-dimensionale.

3.2 MDA: Visualizzazione e confronto con PCA

Si visualizza dunque l'MDA a confronto con la PCA in 1, 2, 3 dimensioni.

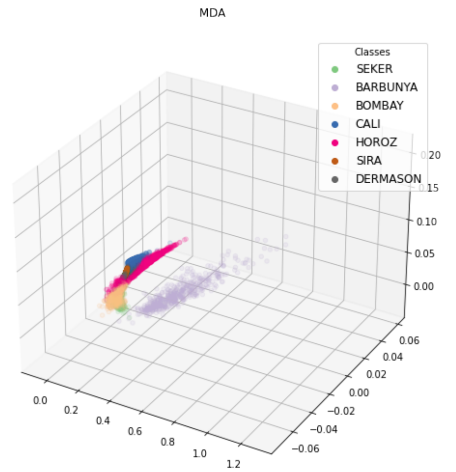
[16]:
```
# Inizializzazione degli oggetti MultipleFisherDiscriminantAnalysis
mda_3dim = MDA(n_dimensions=3)  # Per la proiezione su 3 dimensioni
mda_2dim = MDA(n_dimensions=2)  # Per la proiezione su 2 dimensioni
mda_1dim = MDA(n_dimensions=1)  # Per la proiezione su una dimensione

# Inizializzazione degli oggetti PrincipalComponentAnalysis
pca_3dim = PCA(n_components=3)  # Per la proiezione su 3 dimensioni
pca_2dim = PCA(n_components=2)  # Per la proiezione su 2 dimensioni
pca_1dim = PCA(n_components=1)  # Per la proiezione su una dimensione
```

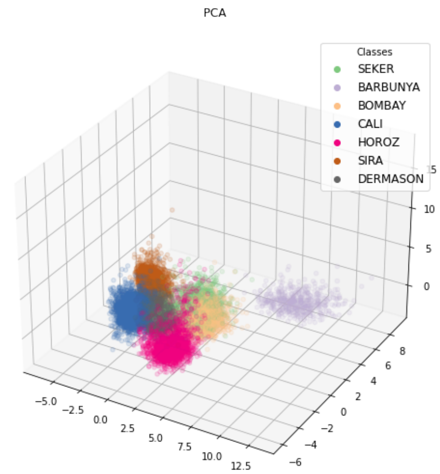
[17]:
```
# Preparazione dataset per i metodi "fit" di mda_1dim, mda_2dim, pca_1dim,␣
 ↪pca_2dim.
X = X_beans_scaled;
y = beans['Class'].values;
#beans['Class'] = le.inverse_transform(beans['Class'])  fmt=['SEKER', 'BARBUNYA'␣
 ↪'BOMBAY', 'CALI', 'HOROZ', 'SIRA', 'DERMASON']

mda_3dim.fit(X_beans_scaled, y);
mda_2dim.fit(X_beans_scaled, y);
mda_1dim.fit(X_beans_scaled, y);

pca_3dim.fit(X_beans_scaled);
pca_2dim.fit(X_beans_scaled);
pca_1dim.fit(X_beans_scaled);
```

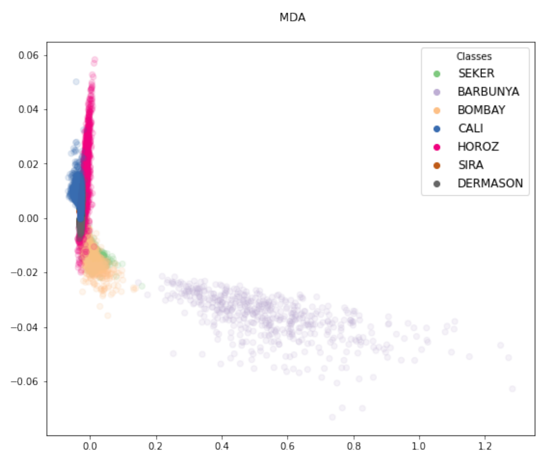
[18]:
```
# Trasformazione del dataset X rispetto alle proiezioni eseguite da mda_1dim,␣
 ↪mda_2dim, pca_1dim, pca_2dim.
Z3m = mda_3dim.transform(X_beans_scaled)  # Trasformazione rispetto mda_3dim
Z2m = mda_2dim.transform(X_beans_scaled)  # Trasformazione rispetto mda_2dim
Z1m = mda_1dim.transform(X_beans_scaled)  # Trasformazione rispetto mda_1dim
Z3p = pca_3dim.transform(X_beans_scaled)  # Trasformazione rispetto pca_3dim
Z2p = pca_2dim.transform(X_beans_scaled)  # Trasformazione rispetto pca_2dim
Z1p = pca_1dim.transform(X_beans_scaled)  # Trasformazione rispetto pca_1dim
```

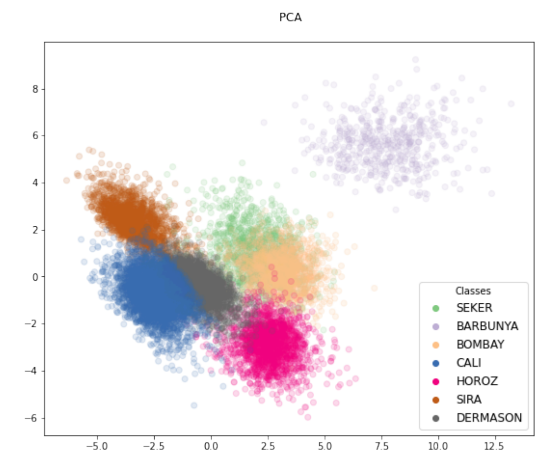
[19]:
```
# Plot per proiezione in R^3
fig3 = plt.figure(figsize=(22.6, 8.475));
ax3m = fig3.add_subplot(1, 2, 1, projection='3d');
scatter = ax3m.scatter(Z3m[:, 0], Z3m[:, 1], Z3m[:, 2], c=y, alpha=1.00);
ax3m.set_title('\n MDA\n');
ax3m.legend(handles=scatter.legend_elements()[0], labels=Beans_codes.keys(),␣
 ↪fontsize='large', title="Classes");
scatter.set_alpha(0.15);
ax3p = fig3.add_subplot(1, 2, 2, projection='3d');
scatter = ax3p.scatter(Z3p[:, 0], Z3p[:, 1], Z3p[:, 2], c=y, alpha=1.00);
ax3p.set_title('\n PCA\n');
ax3p.legend(handles=scatter.legend_elements()[0], labels=Beans_codes.keys(),␣
 ↪fontsize='large', title="Classes");
```

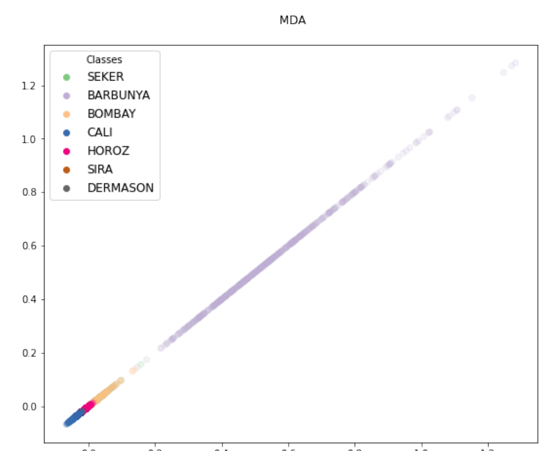
```
scatter.set_alpha(0.15);

# Plot per proiezione in R^2
fig2, axs2 = plt.subplots(1, 2, figsize=(20, 7.5));
scatter = axs2[0].scatter(Z2m[:, 0], Z2m[:, 1], c=y, alpha=1.00);
axs2[0].set_title('\n MDA\n');
axs2[0].legend(handles=scatter.legend_elements()[0], labels=Beans_codes.keys(),␣
 ↪fontsize='large', title="Classes");
scatter.set_alpha(0.15);
scatter = axs2[1].scatter(Z2p[:, 0], Z2p[:, 1], c=y, alpha=1.00);
axs2[1].set_title('\n PCA\n');
axs2[1].legend(handles=scatter.legend_elements()[0], labels=Beans_codes.keys(),␣
 ↪fontsize='large', title="Classes");
scatter.set_alpha(0.15);

# Plot per proiezione in R^1
fig1, axs1 = plt.subplots(1, 2, figsize=(20, 7.5));
scatter = axs1[0].scatter(Z1m, Z1m, c=y, alpha=1.00);
axs1[0].set_title('\n MDA\n');
axs1[0].legend(handles=scatter.legend_elements()[0], labels=Beans_codes.keys(),␣
 ↪fontsize='large', title="Classes");
scatter.set_alpha(0.15);

scatter = axs1[1].scatter(Z1p, Z1p, c=y, alpha=1.00);
axs1[1].set_title('\n PCA\n');
axs1[1].legend(handles=scatter.legend_elements()[0], labels=Beans_codes.keys(),␣
 ↪fontsize='large', title="Classes");
scatter.set_alpha(0.15);
```

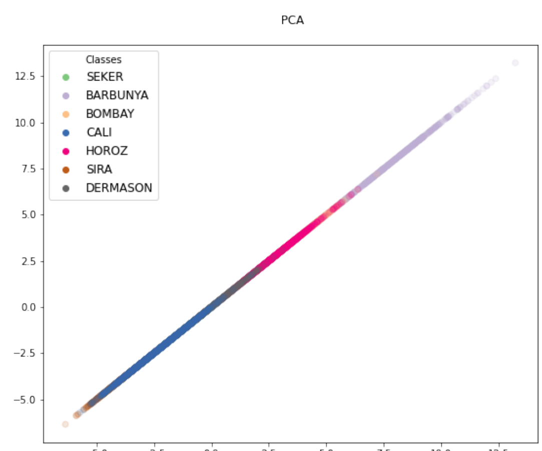

MDA
Classes
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON


PCA
Classes
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON


MDA
Classes
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON


PCA
Classes
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON


MDA
Classes
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON


PCA
Classes
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

4 LDA - Linear Discriminant Analysis

La LDA è un algoritmo supervisionato di classificazione di nuovi dati che quindi richiede l'uso di un training set e di un test set.

4.1 LDA: basi teoriche

Questo si basa sull'analisi distinta di ciascun predittore X su ciascuna delle classi target k tramite la formula di Bayes:

$$\mathbb{P}(Y = k|X = x) = \frac{\mathbb{P}(X = x|Y = k)\mathbb{P}(Y = k)}{\mathbb{P}(X = x)}.$$

Possiamo riscriverlo come:

$$\mathbb{P}(Y = k|X = x) = \frac{\pi_k f_k(x)}{\sum_{p=1}^{k} \pi_p f_p(x)}$$

dove:

- $f_k(x) = \mathbb{P}(X = x|Y = k)$ è la densità di X nella classe k;
- $\pi_k = \mathbb{P}(Y = k)$ è la probabilità marginale (o a priori) della classe k.

In generale si userà il concetto della **highest density**, cioè ogni istanza verrà assegnata alla classe alle quale corrisponde la più alta densità di probabilità.

In tal senso ha senso definire dei **decision boundaries**, cioè degli iperpiani, relativi ad una singola classe (One-vs-Rest), che divino lo spazio in due iperspazi, uno di appartenenza alla classe e uno di non appartenenza.

Anche in questo caso è richiesta omoschedasticità e distribuzione "quasi" normale. In tali condizioni risulta, utilizzando la notazione introdotta in precedenza:

$$f_k(x) = \frac{1}{(2\pi)^{k/2} det(\Sigma)^{1/2}} \exp\Big(-\frac{1}{2}(x-\mu_k)^T \Sigma^{-1}(x-\mu_k)\Big)$$

dove con Σ si intende la matrice di covarianza del dataset. Possiamo sostituire quest'ultima nella formula di Bayes; passando poi al logaritmo (funzione monotona strettamente crescente –> non cambia massimi e minimi) otteniamo il **Discriminant Score**:

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k + \log \pi_k,$$

che come possiamo notare è una funzione lineare di x. Massimizzare la densità di probabilità cercata significa quindi massimizzare il discriminant score. Si definiscono quindi i **Bayes Desicion Boundaries** come le curve lungo le quali i discriminant score di due classi confinanti si eguagliano.

In primo luogo, dividiamo il dataset in training set e test set attraverso la funzione **train_test_split**.

[20]:
```
random_seed = 20210422  # Random seed caratterizzante la suddivisione in␣
 ↪training e test set
test_p = 0.45  # Percentuale di dati da utilizzare come test set

X = X_beans_scaled
y = beans['Class'].values
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_p,␣
 ↪random_state=random_seed, shuffle=True)
```

Quindi, attraverso il training set, stimiamo π_k, μ_k, Σ (metodo: fit) per ottenere una stima del discriminant score e quindi della denisità di probabilità:

$$\hat{\mathbb{P}}(Y = k|X = x) = \frac{\exp(\hat{\delta}_k(x))}{\sum_{i=1}^{K} \exp(\hat{\delta}_i(x))}.$$

Successivamente applichiamo il modello cercato al test set (metodo: predict).

4.2 LDA: Accuracy e Confusion Matrix

Visualizziamo i risultati ottenuti e la loro precisione tramite accuracy score e confusion matrix.

[21]:
```
lda = LDA()

lda.fit(X_train, y_train)
y_pred = lda.predict(X_test)
y_pred_proba = lda.predict_proba(X_test)

y_pred_df = pd.DataFrame({'Pred. Class': y_pred,
                          'P(Class 0) - %': np.round(y_pred_proba[:, 0] * 100,␣
 ↪decimals=2),
                          'P(Class 1) - %': np.round(y_pred_proba[:, 1] * 100,␣
 ↪decimals=2),
                          'P(Class 2) - %': np.round(y_pred_proba[:, 2] * 100,␣
 ↪decimals=2),
                          'P(Class 3) - %': np.round(y_pred_proba[:, 3] * 100,␣
 ↪decimals=2),
                          'P(Class 4) - %': np.round(y_pred_proba[:, 4] * 100,␣
 ↪decimals=2),
                          'P(Class 5) - %': np.round(y_pred_proba[:, 5] * 100,␣
 ↪decimals=2),
                          'P(Class 6) - %': np.round(y_pred_proba[:, 6] * 100,␣
 ↪decimals=2)}) # a scopo estetico

scores_dict = {'Training Set': lda.score(X_train, y_train), 'Test Set': lda.
 ↪score(X_test, y_test)}
scores = pd.DataFrame(scores_dict, index=['Accuracy'])

display(scores)
display(y_pred_df)

cm = confusion_matrix(y_test, y_pred)
cm=pd.DataFrame(cm)
```

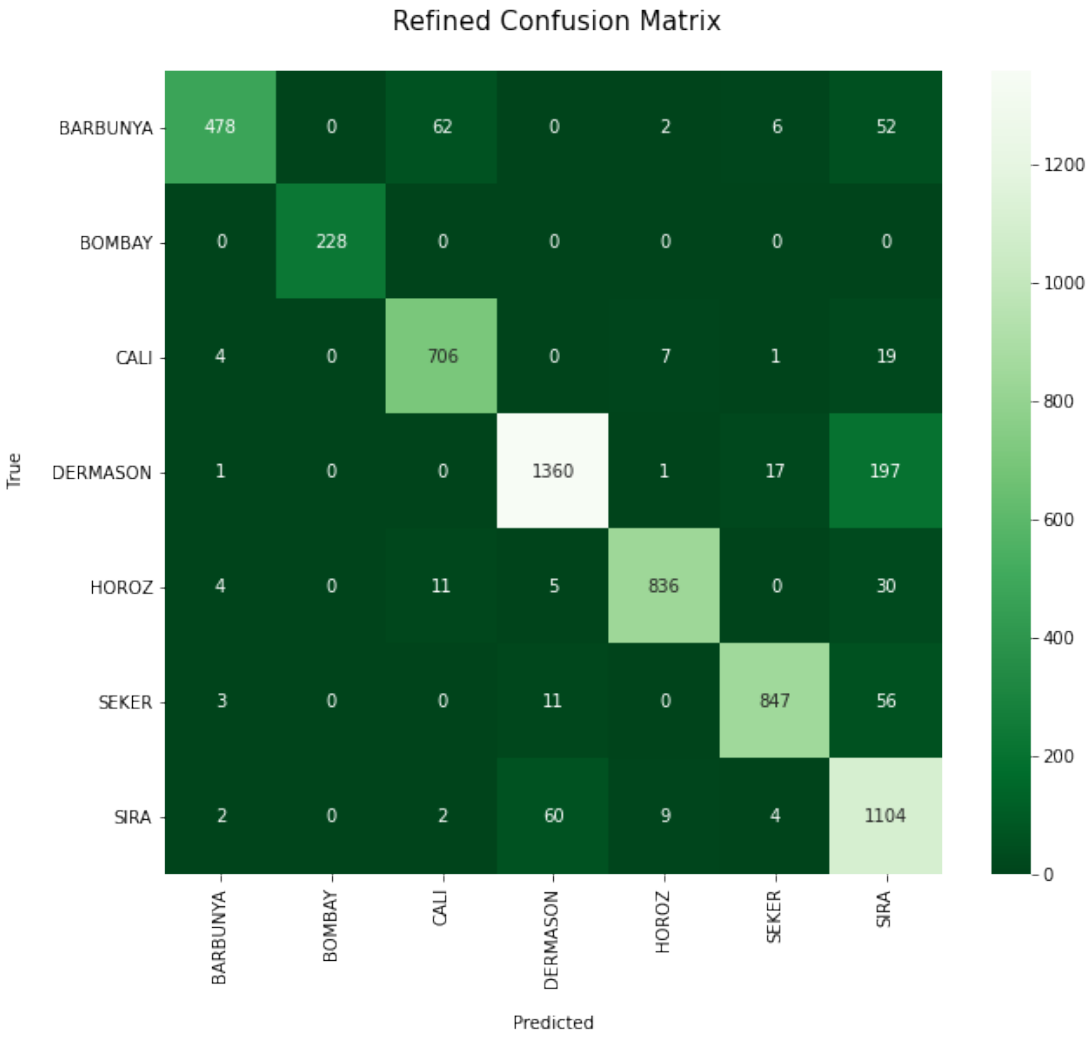
```
labels = le.classes_
class_names = labels

fig = plt.figure(figsize=(10, 8.5))
ax = plt.subplot()
sns.heatmap(cm, annot=True, ax = ax, fmt = 'g', cmap='Greens_r'); #annot=True to␣
 ↪annotate cells
ax.set_xlabel('\nPredicted\n', fontsize=10)
ax.xaxis.set_label_position('bottom')
plt.xticks(rotation=90)
ax.xaxis.set_ticklabels(class_names, fontsize = 10)
ax.xaxis.tick_bottom()

ax.set_ylabel('\n True\n', fontsize=10)
ax.yaxis.set_ticklabels(class_names, fontsize = 10)
plt.yticks(rotation=0)

plt.title('\n Refined Confusion Matrix\n', fontsize=15)
plt.show()
```

```
          Training Set  Test Set
Accuracy      0.902752  0.907592
```

	Pred. Class	P(Class 0) - %	P(Class 1) - %	P(Class 2) - %	\
0	6	0.16	0.0	0.00	
1	6	0.00	0.0	0.00	
2	6	0.00	0.0	0.00	
3	6	0.00	0.0	0.00	
4	2	15.56	0.0	84.44	
...	...	...	...	...	
6120	2	0.00	0.0	100.00	
6121	3	0.00	0.0	0.00	
6122	0	100.00	0.0	0.00	
6123	3	0.00	0.0	0.00	
6124	4	0.00	0.0	0.00	

	P(Class 3) - %	P(Class 4) - %	P(Class 5) - %	P(Class 6) - %
0	5.28	0.0	0.0	94.56
1	0.12	0.0	0.0	99.88
2	1.71	0.0	0.0	98.29
3	0.33	0.0	0.0	99.67
4	0.00	0.0	0.0	0.00
...	...	...	...	...
6120	0.00	0.0	0.0	0.00
6121	72.89	0.0	0.0	27.11
6122	0.00	0.0	0.0	0.00
6123	97.88	0.0	0.0	2.12

```
6124            0.00            100.0            0.0            0.00

[6125 rows x 8 columns]
```

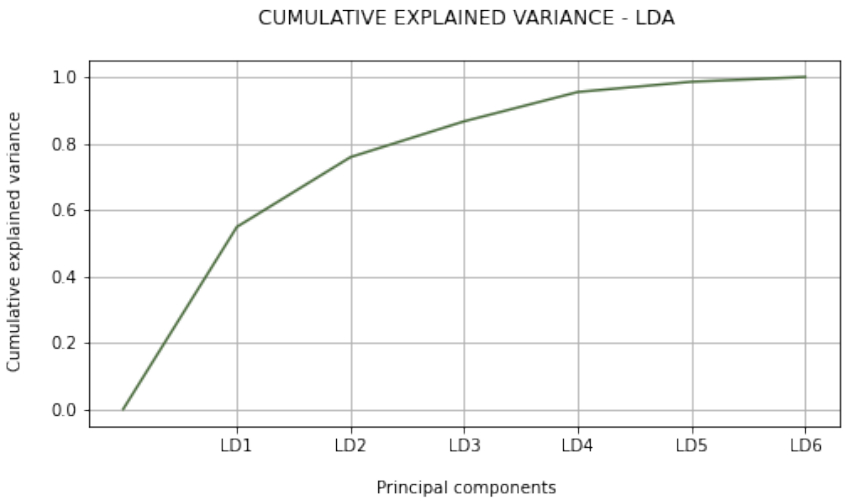


Possiamo vedere come la precisione del test set sia abbastanza alta, di circa il 91%, valore più che accettabile.

4.3 LDA: Visualizzazione e confronto con MDA

Passiamo ora alla rappresentazione. Secondo lo stesso metodo già applicato vediamo in quante dimensioni si ha un livello di varianza cumulativa spiegata sufficiente ad una corretta visualizzazione.

[22]:
```
X_beans = beans.iloc[:, :-1]  # Escludo l'ultima colonna dei target

fig = plt.figure(figsize=(8, 4));
plt.plot(np.insert(np.cumsum(lda.explained_variance_ratio_), 0, 0), color=(68/
 ↪235,99/235,56/235))
plt.xticks(ticks=np.arange(1, 7),
           labels=[f'LD{i}' for i in range(1, 7)])
plt.xlabel('\n Principal components\n')
plt.ylabel('\n Cumulative explained variance\n')
plt.title('\n CUMULATIVE EXPLAINED VARIANCE - LDA\n')
plt.grid()
plt.show()
```

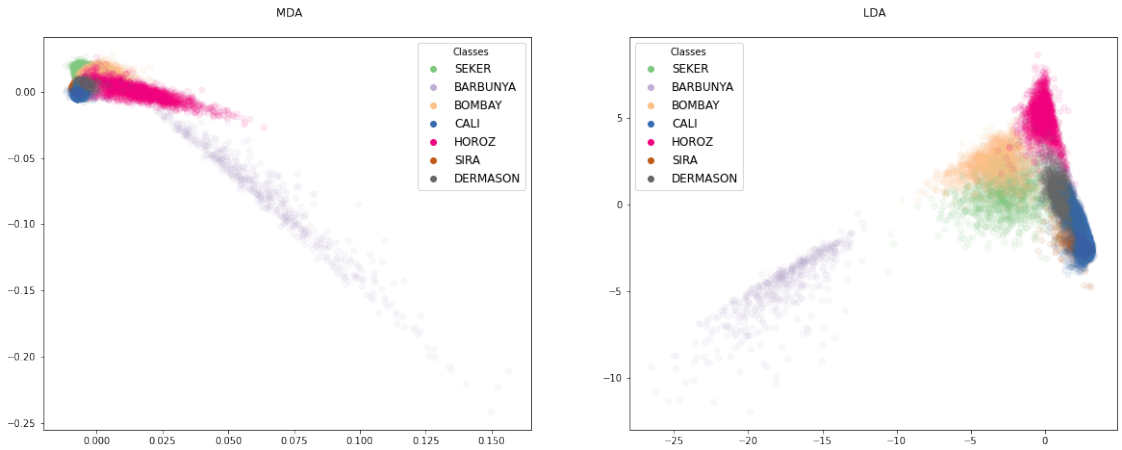


Possiamo vedere come in 2 dimensioni abbiamo più del 70% di varianza totale, quindi accettabile.

Proseguiamo quindi visualizzando i risultati della LDA e confrontandoli con quelli della MDA.

[23]:
```
mda = MDA()
mda.fit(X_train, y_train) # l'ho fatta solo sul training set, per eseere␣
 ↪comparabile con la LDA.

Zmda = mda.transform(X)
Zlda = lda.transform(X)

fig, axs = plt.subplots(1, 2, figsize=(20, 7.5));
scatter = axs[0].scatter(Zmda[:, 0], Zmda[:, 1], c=y, alpha=1.00);
axs[0].legend(handles=scatter.legend_elements()[0], labels=Beans_codes.keys(),␣
 ↪fontsize='large', title="Classes");
axs[0].set_title('\n MDA\n');
scatter.set_alpha(0.075)
plt.grid()
```

```
scatter = axs[1].scatter(Zlda[:, 0], Zlda[:, 1], c=y, alpha=1.00);
axs[1].legend(handles=scatter.legend_elements()[0], labels=Beans_codes.keys(),␣
 ↪fontsize='large', title="Classes");
axs[1].set_title('\n LDA\n');
scatter.set_alpha(0.075)
plt.grid()
```

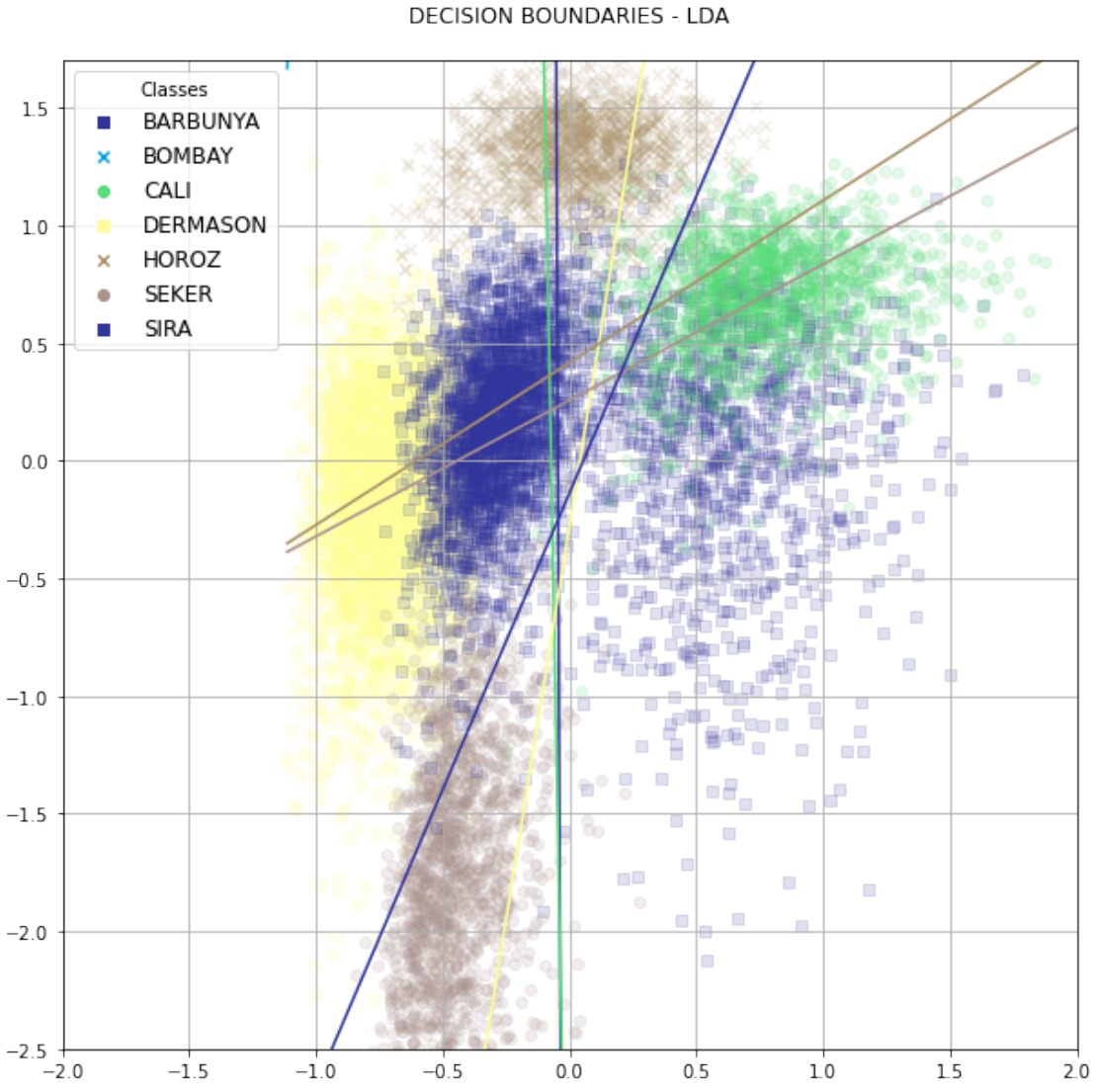


4.4 LDA: Decision Boundaries OvR

Non resta che visualizzare i Bayes Desicion Boundaries. Per tale visualizzazione si è scelta una classifcazione One-vs-Rest, cioè il decision boundary di una classe è costruito eguagliando il determinant score della classe in esame e quello di tutti gli altri dati trattati come un'unica classe. Questo è utile per visualizzazioni multiclasse (più di 2).

[24]:
```
cmap = matplotlib.cm.get_cmap('terrain')

for l,c,m in zip(np.unique(y),[np.array([cmap(0)]),np.array([cmap(1/6)]),np.
 ↪array([cmap(1/3)]),np.array([cmap(1/2)]),np.array([cmap(2/3)]),np.
 ↪array([cmap(5/6)]),np.array([cmap(1)])],['s','x','o','s','x','o','s']):
    scatter = plt.scatter(X[y==l,0], X[y==l,5], c=c, marker=m, label=le.
 ↪classes_[l], alpha=1.00)
    scatter.set_alpha(0.15);
    plt.xlim(-2, 2);
    plt.ylim(-2.5, 2);
    plt.grid();

x1 = np.array([np.min(X[:,0], axis=0), np.max(X[:,0], axis=0)])

for i, c in enumerate([cmap(0), cmap(1/6),cmap(1/3),cmap(1/2),cmap(2/3),cmap(5/
 ↪6),cmap(1)]):
    b, w1, w2 = lda.intercept_[i], lda.coef_[i][0], lda.coef_[i][5];
```

```
    y1 = -(b+x1*w1)/w2;
    lines = plt.plot(x1,y1,c=c,alpha=1);
    leg = plt.legend(handles=scatter.legend_elements()[0], labels=le.
 ↪classes_[i], fontsize='large', title="Classes");
    plt.xlim(-2, 2);
    plt.ylim(-2.5, 1.7);
    plt.title('\nDECISION BOUNDARIES - LDA\n');

for lh in leg.legendHandles:
    lh.set_alpha(1)
```



5 SVM - Support Vector Machine

La SVM è un algoritmo supervisionato di classificazione di nuovi dati che quindi richiede l'uso di un training set e di un test set.

L'idea è quella di separare dei punti in uno spazio d-dimensionale con un **iperpiano**, cioè un sotto spazio piano affine (d-1)-dimensionale, a seconda della classe di appartenenza. Diviso lo spazio in tanti spazi quanti le classi, all'inserimento di un nuovo dato, per classificarlo basterà vedere la regione di appartenenza.

L'algoritmo si occupa anche di cercare il miglior iperpiano che massimizza il **margine**, cioè il doppio della distanza minima dell'iperpiano dai punti di ogni classe. Inoltre, se non esiste tale iperpiano, si può settare una tolleranza. Nel caso il data set non sia neanche "quasi" linearmente separabile, si passa all'utilizzo dei **kernel**, cioè delle proiezioni che generano delle regioni di appartenenza alle classi non poligonali.

5.1 SVM: basi teoriche

Consideriamo il caso di classificazione binaria (2 classi). Qualche notazione:

1. $\mathcal{T} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_T, y_T)\} \subset \mathbb{R}^n \times \{\pm 1\}$ è il training set costituito da T coppie $(\boldsymbol{x}_i, y_i)$, con $y_i = \pm 1$ rappresentante la classe del vettore $\boldsymbol{x_i} \in \mathbb{R}^n$.
2. Indichiamo rispettivamente l'insieme dei vettori e l'insieme delle classi in $\mathcal{T}$ con
$$X_{\mathcal{T}} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T\},$$
$$Y_{\mathcal{T}} = \{y_1, \ldots, y_T\}$$
gli insiemi dei vettori
3. Indichiamo con $\Pi_{\boldsymbol{w},b}$ l'iperpiano di $\mathbb{R}^n$ definito dal vettore normale $\boldsymbol{w} \in \mathbb{R}^n$ e dal parametro $b \in \mathbb{R}$, cioè
$$\Pi_{\boldsymbol{w},b} := \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{w}^\top \boldsymbol{x} + b = 0\}.$$
4. Indichiamo con $\text{dist}(\Pi_{\boldsymbol{w},b}, \boldsymbol{x})$ la distanza euclidea tra un vettore $\boldsymbol{x} \in \mathbb{R}^n$ e l'iperpiano $\Pi_{\boldsymbol{w},b}$. In particolare, ricordiamo che
$$\text{dist}(\Pi_{\boldsymbol{w},b}, \boldsymbol{x}) = \frac{|\boldsymbol{w}^\top \boldsymbol{x} + b|}{||\boldsymbol{w}||}.$$
5. L'ampiezza del margine di un iperpiano è:
$$M_{\boldsymbol{w},b} = 2 \cdot \min_{\boldsymbol{x}} \text{dist}(\Pi_{\boldsymbol{w},b}, \boldsymbol{x}).$$

Poiché per ogni scalare $k \in \mathbb{R} \setminus \{0\}$ vale $\Pi_{k\boldsymbol{w},kb} \equiv \Pi_{\boldsymbol{w},b}$, possiamo restringere la ricerca dell'iperpiano separatore ottimale a quelli con parametri $\boldsymbol{w}$ e b tali che
$$|\boldsymbol{w}^\top \boldsymbol{x} + b| \geq 1\,,\ \forall\, \boldsymbol{x} \in X_{\mathcal{T}}\,.$$

Questi sono detti **iperpiani canonici** rispetto $X_{\mathcal{T}}$ e vettori $\boldsymbol{x} \in X_{\mathcal{T}}$ tali che $|\boldsymbol{w}^\top \boldsymbol{x} + b| = 1$ sono detti **support vectors**.

Notiamo come se $\Pi_{w,b}$ è un iperpiano canonico, allora $M_{w,b} = \frac{2}{||w||}$ e che restringendo il problema agli iperpiani canonici, massimizzare $M_{w,b} = \frac{2}{||w||}$ è equivalente a minimizzare $\frac{1}{2}||w||^2 = \frac{1}{2}w^\top w$, da cui:

$$\begin{cases} \min_w \frac{1}{2} w^\top w \\ y_i(w^\top x_i + b) \geq 1\,, \quad \forall\, i = 1, \ldots, T \end{cases} \tag{1}$$

Passiamo al problema duale:

$$\begin{cases} \min_\alpha \frac{1}{2} \alpha^\top Q \alpha - \sum_{i=1}^T \alpha_i \\ \sum_{i=1}^T \alpha_i y_i = 0 \\ \alpha_i \geq 0\,, \quad \forall\, i = 1, \ldots, T \end{cases}, \tag{2}$$

dove $Q = \left(q_{i,j}\right)_{i,j=1,\ldots,T} = \left(y_i y_j x_i^\top x_j\right)_{i,j=1,\ldots,T}$.

I dati in analisi sono (altamente) affetti da rumore quindi va adoperato una Soft Margin SVM. Riformuliamo di conseguenza il problema.

$$\begin{cases} \min_w \frac{1}{2} \left(w^\top w + C \sum_{i=1}^T \xi_i^2 \right) & \\ y_i(w^\top x_i + b) \geq 1 - \xi_i\,, & \forall\, i = 1, \ldots, T\ ; \\ \xi_i \geq 0\,, & \forall\, i = 1, \ldots, T \end{cases} \tag{3}$$

Il parametro $C \in \mathbb{R}^+$ è un **parametro di regolarizzazione** che caratterizza il rilassamento delle condizioni per il margine:

- $C \to 0$ aumenta la "*morbidezza*" del margine, permettendo ai vettori x_i di superarlo illimitatamente;
- $C \to +\infty$ aumenta la "*durezza*" del margine, permettendo ai vettori x_i di superarlo impercettibilmente;

Il data set in analisi, inoltre, ha classi fortemente non linearmente separabili, quindi è conveniente l'uso dei Kernel. L'idea è quella di proiettare le istanze in uno spazio di dimensione maggiore dove sono quasi linearmente separabili, classificarle e riproiettarle. Quindi scegliamo una mappa (tipicamente *non lineare*) arbitraria

$$\phi : \mathbb{R}^n \to \mathbb{R}^m\,, \quad \text{con } m > n\,,$$

e risolviamo il problema con una SVM rispetto al *nuovo training set*

$$\mathcal{T}^\phi = \{(\varphi_1, y_1)\,, \ldots, (\varphi_T, y_T)\} \in \mathbb{R}^m \times \{\pm 1\} \ \text{ con } \varphi_i = \phi(x_i),\ \forall i = 1, \ldots, T$$

sperando che ora le classi siano diventate *linearmente separabili*.

Chiaramente ciò comporta un considerevole aumento dei costi computazionali e un possibile fallimento nella classificazione lineare. Per ovviare a ciò si usa il **Kernel Trick**. Sia $\mathcal{X} \subseteq \mathbb{R}^n$ il dominio dei vettori x, quindi un kernel è nella forma $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$.

Per cui esiste un'unica mappa $\phi : \mathcal{X} \to \mathcal{H} \subseteq \mathbb{R}^m$, con $m > n$ ed $\mathcal{H}$ spazio di *Hilbert*, per cui il prodotto scalare in $\mathcal{H}$ di $\varphi_i = \phi(x_i)$ e $\varphi_j = \phi(x_j)$, per ogni $x_i, x_j \in \mathcal{X}$, è definito da k; in altre parole:

$$\langle \varphi_i, \varphi_j \rangle_{\mathcal{H}} = k(x_i, x_j).$$

I vantaggi di tale "trucco" sono due: parliamo di prodotti scalari in $\mathbb{R}^m$ che possono essere calcolati in $\mathbb{R}^n$, quindi si può scegliere un qualsiasi $m > n$ senza appesantire l'algoritmo; non è necessario conoscere ϕ, ma solo k.

5.2 SVM: Grid Search

Scegliamo i parametri C, γ e grado che danno la maggiore precisione nell'uso dei seguenti algoritmi: LinearSVC, SVC con kernel lineare, rbf, polinomiale. Per far questo usiamo la **Grid Search**. Significato dei parametri:

- **C**: aggiunge una penalità ogni volta si incorre in una classificazione errata. Perciò un basso valore di C ha più oggetti mal classificati, ma un alto valore di C potrebbe causare overfitting;
- γ: determina l'influenza dei punti di training sulla classificazione. Per un basso valore di γ, tutti i punti di training avrebbero influenza, un alto valore di γ metterebbe in luce solo i punti vicini al decision boundary, potenzialmente causando overfitting;
- **degree**: indica il grado del polinomio usato nel kernel polinomiale.

Per prima cosa dividiamo il dataset in training, test e validation set (rispettivamente: 30%, 50%, 20% del dataset).

[25]:
```
indices = np.arange(13611)
random_state = 20210520
test_p = 0.3 + 0.2;
val_p = 0.2 / 0.5;
ind_train, ind_test = train_test_split(indices, test_size=test_p,␣
 ↪random_state=random_state, shuffle=True)
ind_train, ind_val = train_test_split(ind_train, test_size=val_p,␣
 ↪random_state=random_state, shuffle=True)
```

Successivamente si crea una griglia tridimensionale dove ad ogni punto sono associati tre valori di C, γ e degree presi dalle liste in input. La funzione GridSearchCV, grazie ai test sul validation set, permette di sottoporre ogni set di parametri ad una lista di algoritmi kernel SVM in input e conseguentemente di stilare una classifica di questi in base all'accuracy.

[32]:
```
param_grid = {'C': [0.01, 0.1, 1, 10, 100, 1000],
              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'degree': [2.0, 3.0],
              'kernel': ['rbf','poly','linear']};

grid = GridSearchCV(estimator=svm.SVC(class_weight='balanced'),␣
 ↪param_grid=param_grid, verbose=0, scoring='f1_weighted',
                     return_train_score=True, cv=zip([ind_train], [ind_val]));
grid.fit(X_beans_scaled, beans['Class'].values);
```

[33]:
```
print(grid.best_estimator_)
grid_predictions = grid.predict(X_test);
df_results = pd.DataFrame(grid.cv_results_)
df_results = df_results.sort_values(['rank_test_score'], ascending=True)
df_results
```

[33]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C
126	0.156030	0.0	0.300859	0.0	100
141	0.160849	0.0	0.287574	0.0	100
93	0.180243	0.0	0.323982	0.0	10
108	0.171835	0.0	0.328953	0.0	10
171	0.251682	0.0	0.281186	0.0	1000
..	...	...	...	...	...
13	1.330852	0.0	0.408091	0.0	0.01
12	1.781157	0.0	1.521823	0.0	0.01
28	1.350783	0.0	0.449401	0.0	0.01
10	1.333883	0.0	0.437775	0.0	0.01
27	1.780757	0.0	1.519031	0.0	0.01

	param_degree	param_gamma	param_kernel
126	2.0	0.01	rbf
141	3.0	0.01	rbf
93	2.0	0.1	rbf
108	3.0	0.1	rbf
171	3.0	0.01	rbf
..	...	...	...
13	2.0	0.0001	poly
12	2.0	0.0001	rbf
28	3.0	0.0001	poly
10	2.0	0.001	poly
27	3.0	0.0001	rbf

	params	split0_test_score
126	{'C': 100, 'degree': 2.0, 'gamma': 0.01, 'kern...	0.926354
141	{'C': 100, 'degree': 3.0, 'gamma': 0.01, 'kern...	0.926354
93	{'C': 10, 'degree': 2.0, 'gamma': 0.1, 'kernel...	0.923691
108	{'C': 10, 'degree': 3.0, 'gamma': 0.1, 'kernel...	0.923691
171	{'C': 1000, 'degree': 3.0, 'gamma': 0.01, 'ker...	0.922920
..	...	...
13	{'C': 0.01, 'degree': 2.0, 'gamma': 0.0001, 'k...	0.033530
12	{'C': 0.01, 'degree': 2.0, 'gamma': 0.0001, 'k...	0.033530
28	{'C': 0.01, 'degree': 3.0, 'gamma': 0.0001, 'k...	0.033530
10	{'C': 0.01, 'degree': 2.0, 'gamma': 0.001, 'ke...	0.033530
27	{'C': 0.01, 'degree': 3.0, 'gamma': 0.0001, 'k...	0.033530

	mean_test_score	std_test_score	rank_test_score	split0_train_score
126	0.926354	0.0	1	0.930488
141	0.926354	0.0	1	0.930488
93	0.923691	0.0	3	0.938295
108	0.923691	0.0	3	0.938295
171	0.922920	0.0	5	0.937811
..	...	...	...	...
13	0.033530	0.0	175	0.033866

```
12          0.033530            0.0             175             0.033866
28          0.033530            0.0             175             0.033866
10          0.033530            0.0             175             0.033866
27          0.033530            0.0             175             0.033866

     mean_train_score  std_train_score
126          0.930488              0.0
141          0.930488              0.0
93           0.938295              0.0
108          0.938295              0.0
171          0.937811              0.0
..                ...              ...
13           0.033866              0.0
12           0.033866              0.0
28           0.033866              0.0
10           0.033866              0.0
27           0.033866              0.0
```

Da qui possiamo vedere come il migliore tra gli algoritmi analizzati è una SVM con un kernel rbf e parametri C=100 e γ=0.01.

[28]:
```
df_results.iloc[:,4:7] = df_results.iloc[:,4:7].astype(float);
to_plot_par = pd.DataFrame(columns=['linear','rbf','poly'],␣
 ↪index=['C','Gamma','degree']);

best = df_results[df_results.param_kernel =='linear'].head(1);
to_plot_par.iloc[0,0] = best.iloc[0,4];
to_plot_par.iloc[1,0] = best.iloc[0,6];

best = df_results[df_results.param_kernel =='poly'].head(1);
to_plot_par.iloc[0,2] = best.iloc[0,4];
to_plot_par.iloc[1,2] = best.iloc[0,4];
to_plot_par.iloc[2,2] = best.iloc[0,5];

best = df_results[df_results.param_kernel =='rbf'].head(1);
to_plot_par.iloc[0,1] = best.iloc[0,4];
to_plot_par.iloc[1,1] = best.iloc[0,6];

to_plot_par
```

[28]:
```
        linear    rbf  poly
C         10.0  100.0  10.0
Gamma     0.01   0.01  10.0
degree     NaN    NaN   2.0
```

Per scopi illustrativi, i migliori parametri anche per una SVM con kernel lineare e per una SVM con kernel polinomiale sono riportati qui sopra.

5.3 SVM: Visualizzazione

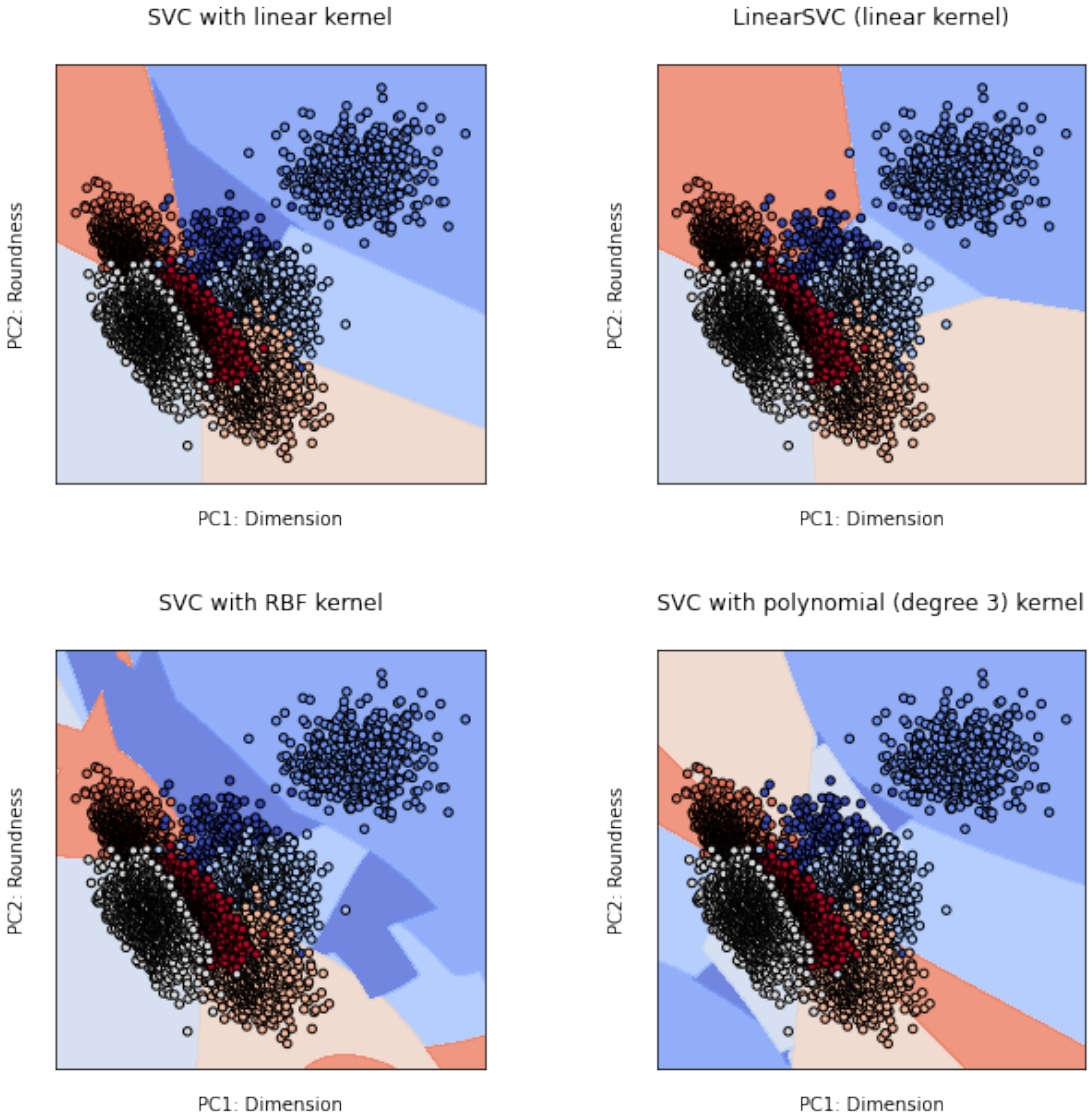
Andiamo quindi a visualizzare i plot dei modelli studiati con i migliori iperparametri.

[29]:
```
def make_meshgrid(x, y, h=.02):
    """Create a mesh of points to plot in

    Parameters
    ----------
    x: data to base x-axis meshgrid on
    y: data to base y-axis meshgrid on
    h: stepsize for meshgrid, optional

    Returns
    -------
    xx, yy : ndarray
    """
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    return xx, yy
```

[30]:
```
def plot_contours(ax, clf, xx, yy, **params):
    """Plot the decision boundaries for a classifier.

    Parameters
    ----------
    ax: matplotlib axes object
    clf: a classifier
    xx: meshgrid ndarray
    yy: meshgrid ndarray
    params: dictionary of params to pass to contourf, optional
    """
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out
```

[31]:
```
# Take the first two features. We could avoid this by using a two-dim dataset
X = np.zeros((Z2p[:,0].shape[0], 2))
for i in range (Z2p[:,0].shape[0]):
    X[i,0] = Z2p[i,0];
    X[i,1] = Z2p[i,1];
y = beans['Class'];

# we create an instance of SVM and fit out data. We do not scale our
# data since we want to plot the support vectors
```

```
models = (svm.SVC(kernel='linear', C=to_plot_par.iloc[0,0]),
          svm.LinearSVC(C=to_plot_par.iloc[0,0], max_iter=10000),
          svm.SVC(kernel='rbf', gamma=to_plot_par.iloc[1,1], C=to_plot_par.
 ↪iloc[0,1]),
          svm.SVC(kernel='poly', degree=to_plot_par.iloc[2,2], gamma=to_plot_par.
 ↪iloc[1,2], C=to_plot_par.iloc[0,2]))
models = (clf.fit(X, y) for clf in models)

# title for the plots
titles = ('\nSVC with linear kernel\n',
          '\nLinearSVC (linear kernel)\n',
          '\nSVC with RBF kernel\n',
          '\nSVC with polynomial (degree 3) kernel\n')

cmap = plt.cm.coolwarm

# Set-up 2x2 grid for plotting.
fig, sub = plt.subplots(2, 2)
plt.subplots_adjust(wspace=0.4, hspace=0.4)
X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)
for clf, title, ax in zip(models, titles, sub.flatten()):
    plot_contours(ax, clf, xx, yy, cmap=cmap, alpha=0.8)
    ax.scatter(X0, X1, c=y, cmap=cmap, s=20, edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xlabel('\nPC1: Dimension\n')
    ax.set_ylabel('\nPC2: Roundness\n')
    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_title(title)

plt.show()
```



6 Conclusioni

Possiamo vedere come l'algoritmo di classificazione più accurato è la Support Vector Machine con Kernel rbv e iperparametri C=100, γ=0.01 per una accuracy score finale del 93,0488%.