# coverType

January 8, 2018

**Reconnaissance du type de recouvrement de la végétation forestière**

Autheurs : Jean-Baptiste AUJOGUE jean-baptiste.aujogue@etu.univ-lyon1.fr, Xian YANG xian.yang@etu.univ-lyon1.fr

20 déc. 2017

## 0.1 0. Introduction

Le but de cet exercice et de faire une analyse sur le jeu de données `covtype.data`, téléchargeable sous ce lien. Une description détaillée se trouve ici. Dans ce jeu de données, il s'agit de 581.012 terres forestières (observations) et, pour chaque observation, sont relevées 10 mesures numériques et 3 mesures catégoriques, dont le type de recouvrement de la forêt (`Cover_Type`), variable essentielle dans notre analyse. L'analyse se déroule comme suit :

1. On importe d'abord le jeu de données, étudie ses variables, vérifie sa conformité et en fait une statistique descriptive.
2. Une ACP avec toutes les variables numériques et une ACM avec les deux variables catégoriques explicatives nous paraîssent également nécessaires afin d'avoir un vague aperçu visuel de la séparabilité du jeu de données par les 7 différents types de recouvrement de la forêt.
3. Ensuite on va essayer de prédire la variable `Cover_Type` en utilisant les autres. A cette fin, légèrement différent de ce qui est demandé dans l'énoncé, on découpe l'entier jeu de données en 3 sous-groupes. Un jeu d'entraînement, sur lequel on ajustera nos modèles, contenant les premiers 11.340 observations. Un jeu de validation, pour sélectionner des hyperparamètres le cas échéant, contenant les 3.780 observations suivantes. Et un jeu de test, qui nous donnera une estimation de l'exactitude des modèles, contenant les dernières 565.892 observation qui restent.

Dans l'étape de la prédiction, les méthodes utiisées sont les suivantes : analyse discriminante linéaire ordinaire, analyse discriminante linéaire avec régularisation, analyse discriminante linéaire à rang réduit, analyse discriminante quadratique ordinaire, analyse discriminante régularisée, k plus proches voisins, k plus proches voisins précédée d'une réduction de dimensions par l'ACP.

## 0.2 I. Importation des données, vérifications de la conformité (pré-traitement) et statistique descriptive

On charge les librairies de Python nécessaires dans l'ensemble de notre notebook.

```
In [35]: # If we are not using Jupyter Notebook or IPython, don't take into account the two foll
         from IPython.core.interactiveshell import InteractiveShell
         InteractiveShell.ast_node_interactivity = "all"

         import pandas as pd
         import numpy as np
         import seaborn as sns
         import matplotlib.pyplot as plt
         %matplotlib inline

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

On lit le jeu de données puis donne à chaque variable un nom propre manuellement, car il n'est pas précisé dans le fichier original.

```
In [3]: df = pd.read_table("covtype.data", header = None, sep = ",",
                          names = ["Elevation", "Aspect", "Slope", "Horizontal_Distance_To_Hydro
                                   "Horizontal_Distance_To_Roadways", "Hillshade_9am", "Hillshad
                                   "Horizontal_Distance_To_Fire_Points"] +
                           ["Wilderness_Area_" + str(i) for i in range(1, 4+1)] +
                           ["Soil_Type_" + str(i) for i in range(1, 40+1)] +
                           ["Cover_Type"])
```

Une fois le jeu de données importé, on s'intéresse d'habitude tout de suite à quatres questions.

1. Mon jeu de donnée il est comment ? (Visualisation d'une petite partie)
2. Quelle est sa taille ? **Réponse : 581.012 lignes Œ 55 colonnes (13 variables).**
3. Les variables quantitatives et les variables qualitatives sous quelles formes sont-elle resp. (surtout les variables qualitatives) ? **Réponse : la variable** `Wilderness_Area` **occupe 4 colonnes en binaire, dont la somme à chaque ligne, d'après notre vérification, vaut strictement 1. Elle est donc bien transformée en modalités. Pareil pour la variable** `Soil_Type` **qui occupe 40 colonnes en binaire. En revanche, la variable** `Cover_Type` **occupe une seule colonne avec sa valeur allant de 1 à 7.**
4. Y a-t-il des valeurs manquantes ? **Réponse : non.**

```
In [6]: df.head()
        df.tail()

        # Is there any missing value in the data frame?
        # Answer: no. So we are fine.
        df.isnull().values.any()

        # What are the numbers of rows and of columns?
        # Answer: 581,012 rows Œ 55 columns.
        df.shape

        # Any line where the Wilderness_Area columns do not sum up to 1?
        # Answer: no. So we are fine.
```

```
(df.loc[:, "Wilderness_Area_1":"Wilderness_Area_4"].sum(axis = 1) == 1).all()
# Any line where the Soil_Type columns do not sum up to 1?
# Answer: no. So we are fine.
(df.loc[:, "Soil_Type_1":"Soil_Type_40"].sum(axis = 1) == 1).all()
```

Out[6]:

| | Elevation | Aspect | Slope | Horizontal_Distance_To_Hydrology | \ |
|---|---|---|---|---|---|
| 0 | 2596 | 51 | 3 | 258 | |
| 1 | 2590 | 56 | 2 | 212 | |
| 2 | 2804 | 139 | 9 | 268 | |
| 3 | 2785 | 155 | 18 | 242 | |
| 4 | 2595 | 45 | 2 | 153 | |

| | Vertical_Distance_To_Hydrology | Horizontal_Distance_To_Roadways | \ |
|---|---|---|---|
| 0 | 0 | 510 | |
| 1 | -6 | 390 | |
| 2 | 65 | 3180 | |
| 3 | 118 | 3090 | |
| 4 | -1 | 391 | |

| | Hillshade_9am | Hillshade_Noon | Hillshade_3pm | \ |
|---|---|---|---|---|
| 0 | 221 | 232 | 148 | |
| 1 | 220 | 235 | 151 | |
| 2 | 234 | 238 | 135 | |
| 3 | 238 | 238 | 122 | |
| 4 | 220 | 234 | 150 | |

| | Horizontal_Distance_To_Fire_Points | ... | Soil_Type_32 | Soil_Type_33 | \ |
|---|---|---|---|---|---|
| 0 | 6279 | ... | 0 | 0 | |
| 1 | 6225 | ... | 0 | 0 | |
| 2 | 6121 | ... | 0 | 0 | |
| 3 | 6211 | ... | 0 | 0 | |
| 4 | 6172 | ... | 0 | 0 | |

| | Soil_Type_34 | Soil_Type_35 | Soil_Type_36 | Soil_Type_37 | Soil_Type_38 | \ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 0 | 0 | 0 | |
| 2 | 0 | 0 | 0 | 0 | 0 | |
| 3 | 0 | 0 | 0 | 0 | 0 | |
| 4 | 0 | 0 | 0 | 0 | 0 | |

| | Soil_Type_39 | Soil_Type_40 | Cover_Type |
|---|---|---|---|
| 0 | 0 | 0 | 5 |
| 1 | 0 | 0 | 5 |
| 2 | 0 | 0 | 2 |
| 3 | 0 | 0 | 2 |
| 4 | 0 | 0 | 5 |

[5 rows x 55 columns]

Out[6]:

| | Elevation | Aspect | Slope | Horizontal_Distance_To_Hydrology | \ |
|---|---|---|---|---|---|
| 581007 | 2396 | 153 | 20 | 85 | |
| 581008 | 2391 | 152 | 19 | 67 | |
| 581009 | 2386 | 159 | 17 | 60 | |
| 581010 | 2384 | 170 | 15 | 60 | |
| 581011 | 2383 | 165 | 13 | 60 | |

| | Vertical_Distance_To_Hydrology | Horizontal_Distance_To_Roadways | \ |
|---|---|---|---|
| 581007 | 17 | 108 | |
| 581008 | 12 | 95 | |
| 581009 | 7 | 90 | |
| 581010 | 5 | 90 | |
| 581011 | 4 | 67 | |

| | Hillshade_9am | Hillshade_Noon | Hillshade_3pm | \ |
|---|---|---|---|---|
| 581007 | 240 | 237 | 118 | |
| 581008 | 240 | 237 | 119 | |
| 581009 | 236 | 241 | 130 | |
| 581010 | 230 | 245 | 143 | |
| 581011 | 231 | 244 | 141 | |

| | Horizontal_Distance_To_Fire_Points | ... | Soil_Type_32 | \ |
|---|---|---|---|---|
| 581007 | 837 | ... | 0 | |
| 581008 | 845 | ... | 0 | |
| 581009 | 854 | ... | 0 | |
| 581010 | 864 | ... | 0 | |
| 581011 | 875 | ... | 0 | |

| | Soil_Type_33 | Soil_Type_34 | Soil_Type_35 | Soil_Type_36 | Soil_Type_37 | \ |
|---|---|---|---|---|---|---|
| 581007 | 0 | 0 | 0 | 0 | 0 | |
| 581008 | 0 | 0 | 0 | 0 | 0 | |
| 581009 | 0 | 0 | 0 | 0 | 0 | |
| 581010 | 0 | 0 | 0 | 0 | 0 | |
| 581011 | 0 | 0 | 0 | 0 | 0 | |

| | Soil_Type_38 | Soil_Type_39 | Soil_Type_40 | Cover_Type |
|---|---|---|---|---|
| 581007 | 0 | 0 | 0 | 3 |
| 581008 | 0 | 0 | 0 | 3 |
| 581009 | 0 | 0 | 0 | 3 |
| 581010 | 0 | 0 | 0 | 3 |
| 581011 | 0 | 0 | 0 | 3 |

[5 rows x 55 columns]

Out[6]: False

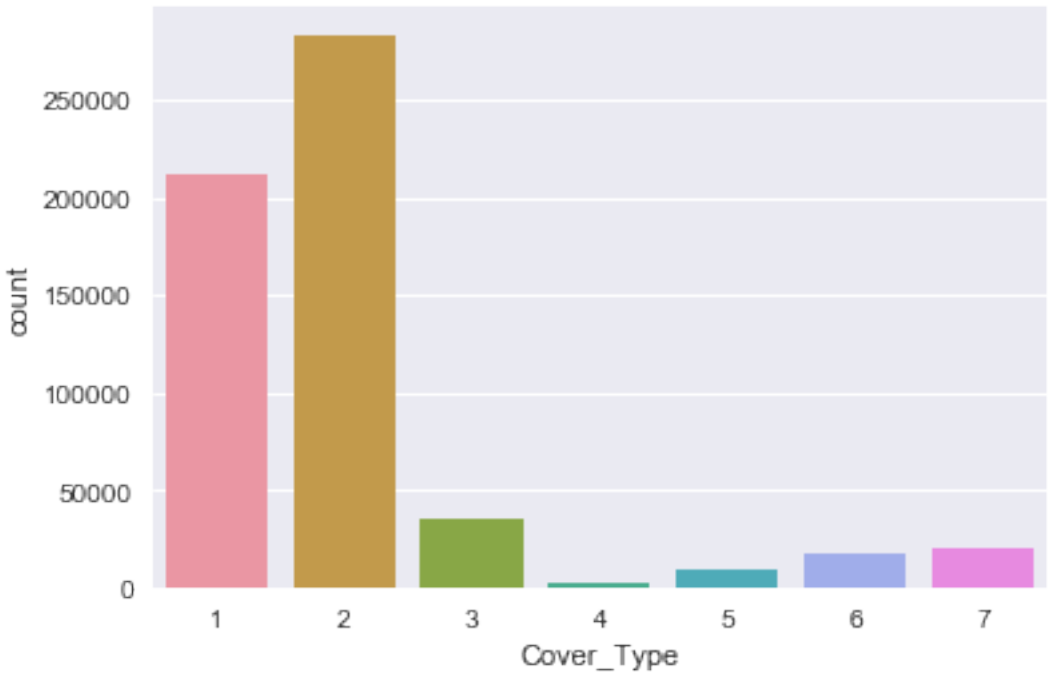
Out[6]: (581012, 55)

Out[6]: True

```
Out[6]: True
```

Ensuite on s'intéresse à la statistique descriptive. On commence d'abord par le type de recouvrement valant de 1 à 7. On voit que la répartition de ces 7 types n'est pas du tout équilibrée. Par contre on peut vérifier que les effectifs des 11.340 premières observations sont équilibrés, ce qui est important pour le modèle d'analyse discriminante linéaire dans la suite.

```
In [41]: df["Cover_Type"].value_counts()
         sns.countplot(x="Cover_Type", data=df)
         plt.show()

Out[41]: 2    283301
         1    211840
         3     35754
         7     20510
         6     17367
         5      9493
         4      2747
         Name: Cover_Type, dtype: int64

Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x2cf81c46f28>
```




Comment alors sont réparties les différentes modalités des variables qualitatives `Wilderness_Area` et `Soil_Type` ? Pas équilibrées du tout non plus.

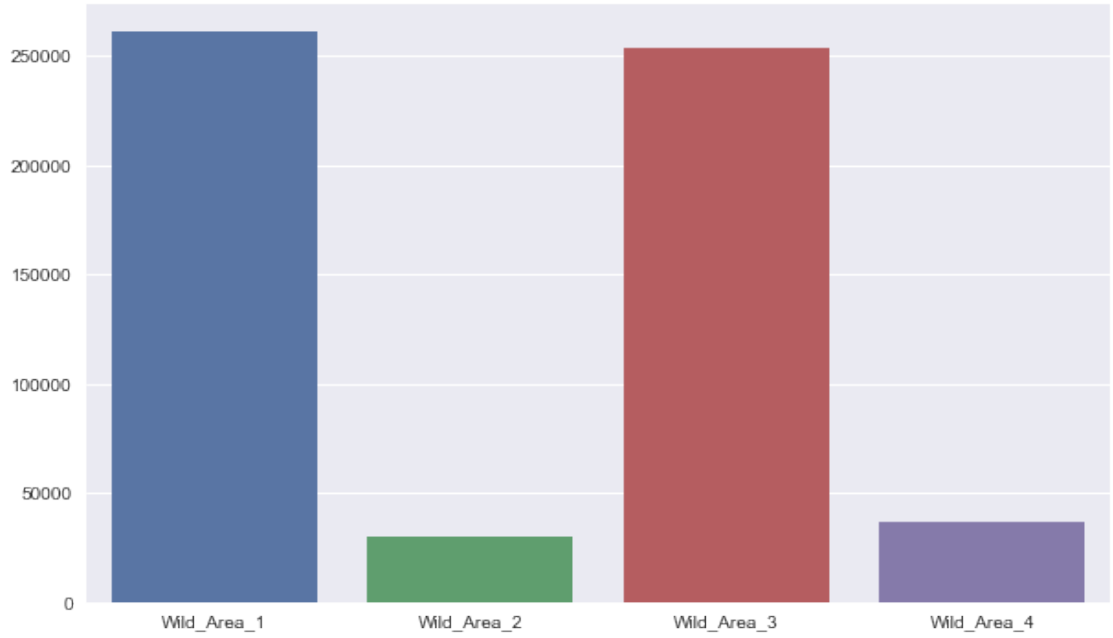
```
In [7]: df.loc[:, "Wilderness_Area_1":"Wilderness_Area_4"].sum(axis = 0)
        plt.figure(figsize=(10,6))
```

```
sns.barplot(x = ["Wild_Area_" + str(i) for i in range(1, 5)], y = df.loc[:, "Wilderness_
plt.show()

df.loc[:, "Soil_Type_1":"Soil_Type_40"].sum(axis = 0)
plt.figure(figsize=(20,6))
sns.barplot(x = list(range(1, 41)), y = df.loc[:, "Soil_Type_1":"Soil_Type_40"].sum(axis
plt.show()
```

```
Out[7]: Wilderness_Area_1    260796
        Wilderness_Area_2     29884
        Wilderness_Area_3    253364
        Wilderness_Area_4     36968
        dtype: int64
```

```
Out[7]: <matplotlib.figure.Figure at 0x172cc228fd0>
```

```
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x172cc2746a0>
```

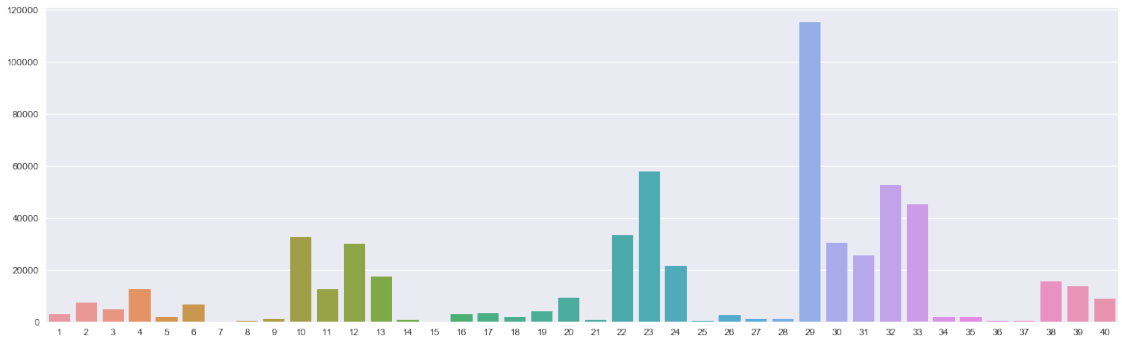



```
Out[7]: Soil_Type_1       3031
        Soil_Type_2       7525
        Soil_Type_3       4823
        Soil_Type_4      12396
        Soil_Type_5       1597
        Soil_Type_6       6575
        Soil_Type_7        105
        Soil_Type_8        179
        Soil_Type_9       1147
```

```
Soil_Type_10     32634
Soil_Type_11     12410
Soil_Type_12     29971
Soil_Type_13     17431
Soil_Type_14       599
Soil_Type_15         3
Soil_Type_16      2845
Soil_Type_17      3422
Soil_Type_18      1899
Soil_Type_19      4021
Soil_Type_20      9259
Soil_Type_21       838
Soil_Type_22     33373
Soil_Type_23     57752
Soil_Type_24     21278
Soil_Type_25       474
Soil_Type_26      2589
Soil_Type_27      1086
Soil_Type_28       946
Soil_Type_29    115247
Soil_Type_30     30170
Soil_Type_31     25666
Soil_Type_32     52519
Soil_Type_33     45154
Soil_Type_34      1611
Soil_Type_35      1891
Soil_Type_36       119
Soil_Type_37       298
Soil_Type_38     15573
Soil_Type_39     13806
Soil_Type_40      8750
dtype: int64
```

Out[7]: <matplotlib.figure.Figure at 0x172cc8390f0>

Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x172cc82f6a0>

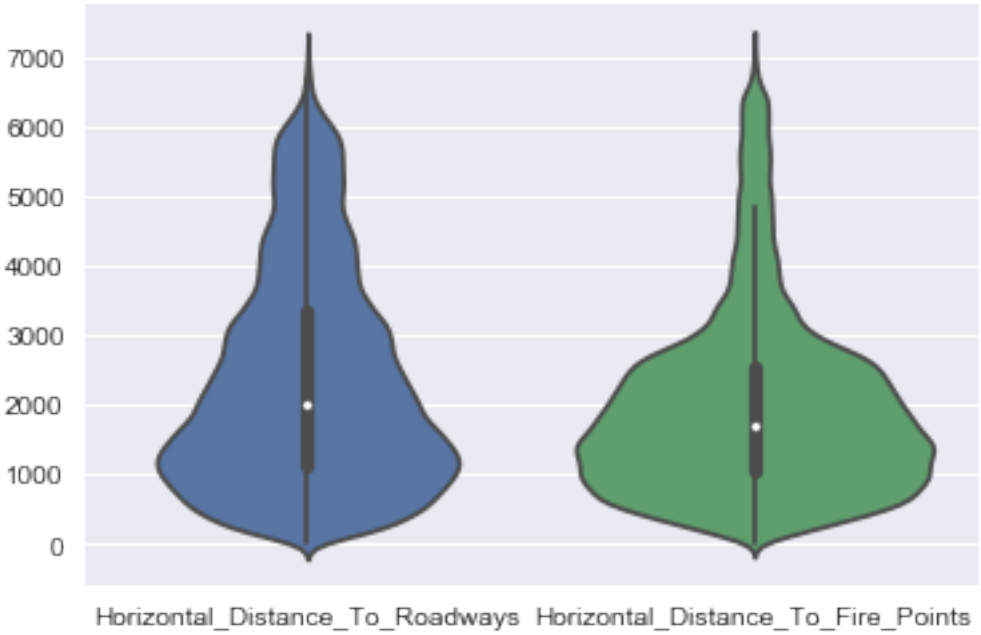


Pour finir cette partie, on fait de la statistique descriptive pour les variables quantitatives, regroupées par similarité de leurs significations.

```
In [136]: df.iloc[:, [5, 9]].describe()
          sns.violinplot(data=df.iloc[:, [5, 9]])
```

```
Out[136]:        Horizontal_Distance_To_Roadways  Horizontal_Distance_To_Fire_Points
          count                    581012.000000                       581012.000000
          mean                       2350.146611                         1980.291226
          std                        1559.254870                         1324.195210
          min                           0.000000                            0.000000
          25%                        1106.000000                         1024.000000
          50%                        1997.000000                         1710.000000
          75%                        3328.000000                         2550.000000
          max                        7117.000000                         7173.000000
```

```
Out[136]: <matplotlib.axes._subplots.AxesSubplot at 0x2cf98c4c8d0>
```

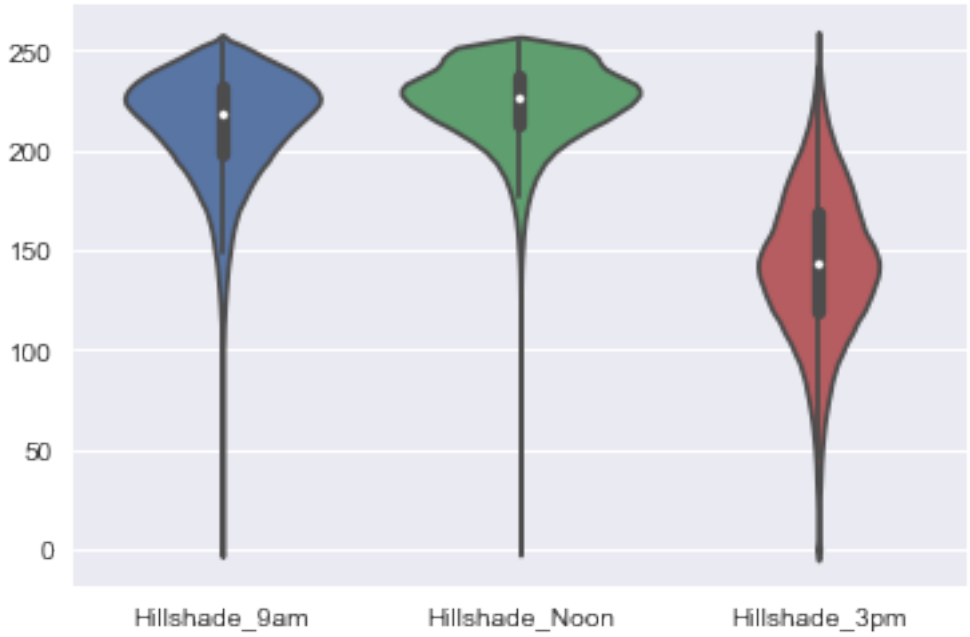



```
In [8]: df.iloc[:, [6, 7, 8]].describe()
        sns.violinplot(data=df.iloc[:, [6, 7, 8]])
```

```
Out[8]:        Hillshade_9am  Hillshade_Noon  Hillshade_3pm
        count  581012.000000   581012.000000  581012.000000
        mean      212.146049      223.318716     142.528263
        std        26.769889       19.768697      38.274529
        min         0.000000        0.000000       0.000000
        25%       198.000000      213.000000     119.000000
```

| | | | |
|---|---|---|---|
| 50% | 218.000000 | 226.000000 | 143.000000 |
| 75% | 231.000000 | 237.000000 | 168.000000 |
| max | 254.000000 | 254.000000 | 254.000000 |

Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x172cc228ef0>

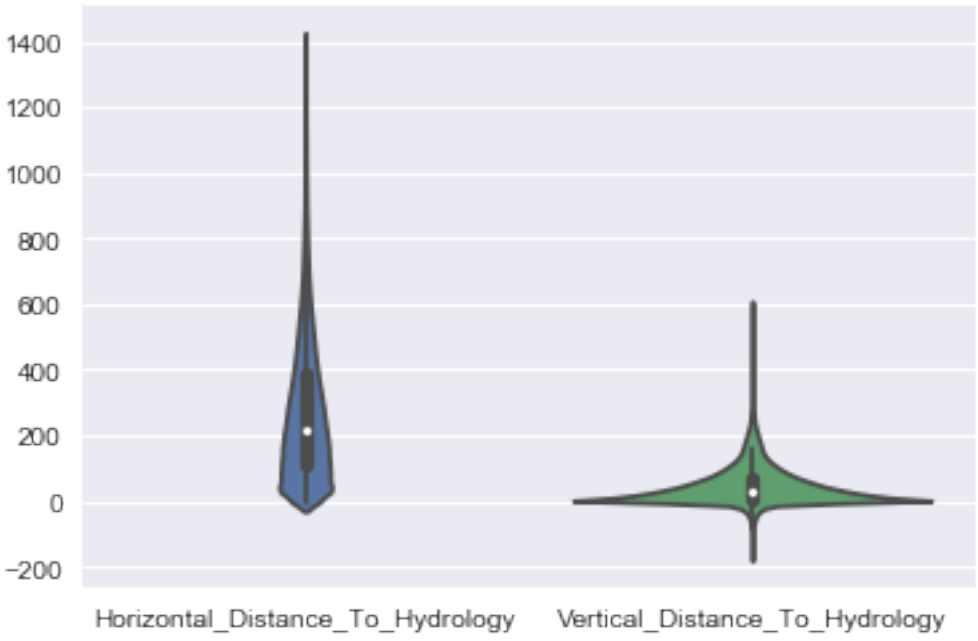



In [138]:
```
df.iloc[:, [3, 4]].describe()
sns.violinplot(data=df.iloc[:, [3, 4]])
```

Out[138]:

| | Horizontal_Distance_To_Hydrology | Vertical_Distance_To_Hydrology |
|---|---|---|
| count | 581012.000000 | 581012.000000 |
| mean | 269.428217 | 46.418855 |
| std | 212.549356 | 58.295232 |
| min | 0.000000 | -173.000000 |
| 25% | 108.000000 | 7.000000 |
| 50% | 218.000000 | 30.000000 |
| 75% | 384.000000 | 69.000000 |
| max | 1397.000000 | 601.000000 |

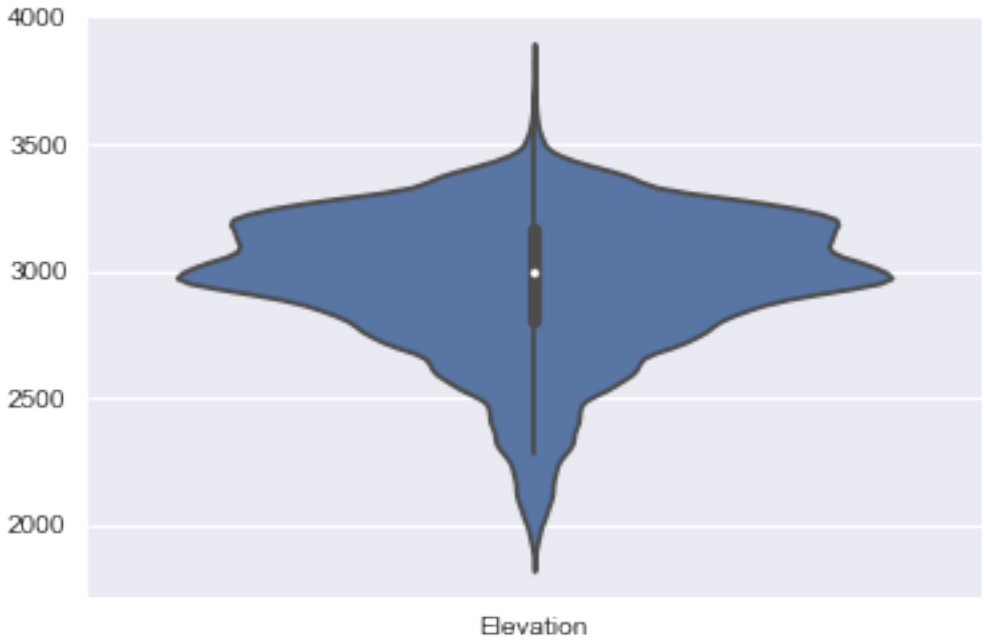
Out[138]: <matplotlib.axes._subplots.AxesSubplot at 0x2cf9afe0438>




```
In [142]: for i in range(3):
              df.iloc[:, [i]].describe()
              sns.violinplot(data=df.iloc[:, [i]])
              plt.show()
```

```
Out[142]:            Elevation
          count  581012.000000
          mean     2959.365301
          std       279.984734
          min      1859.000000
          25%      2809.000000
          50%      2996.000000
          75%      3163.000000
          max      3858.000000
```

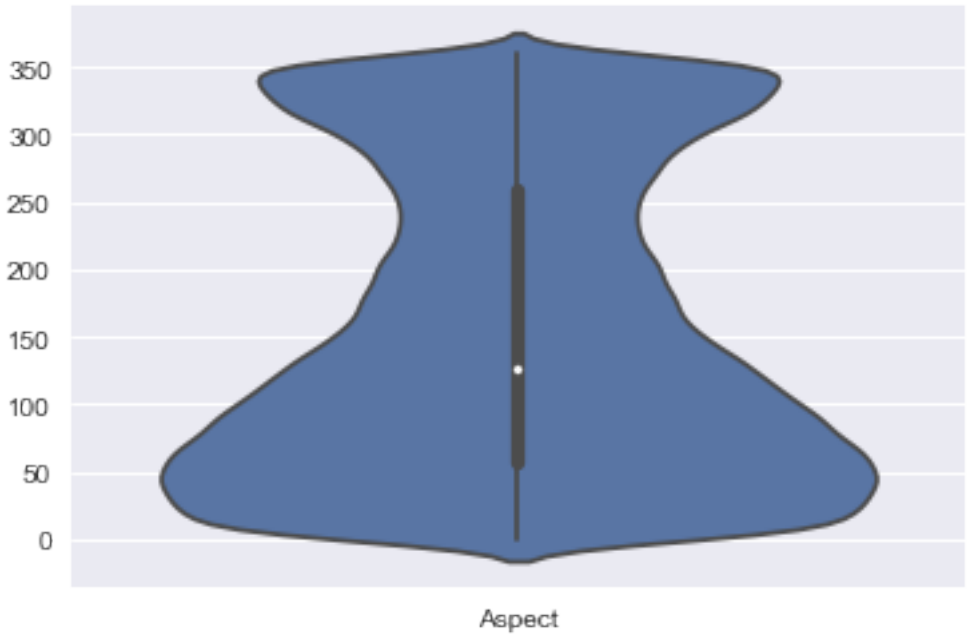
```
Out[142]: <matplotlib.axes._subplots.AxesSubplot at 0x2cf9bc5e390>
```




Out[142]:

| | Aspect |
|---|---|
| count | 581012.000000 |
| mean | 155.656807 |
| std | 111.913721 |
| min | 0.000000 |
| 25% | 58.000000 |
| 50% | 127.000000 |
| 75% | 260.000000 |
| max | 360.000000 |

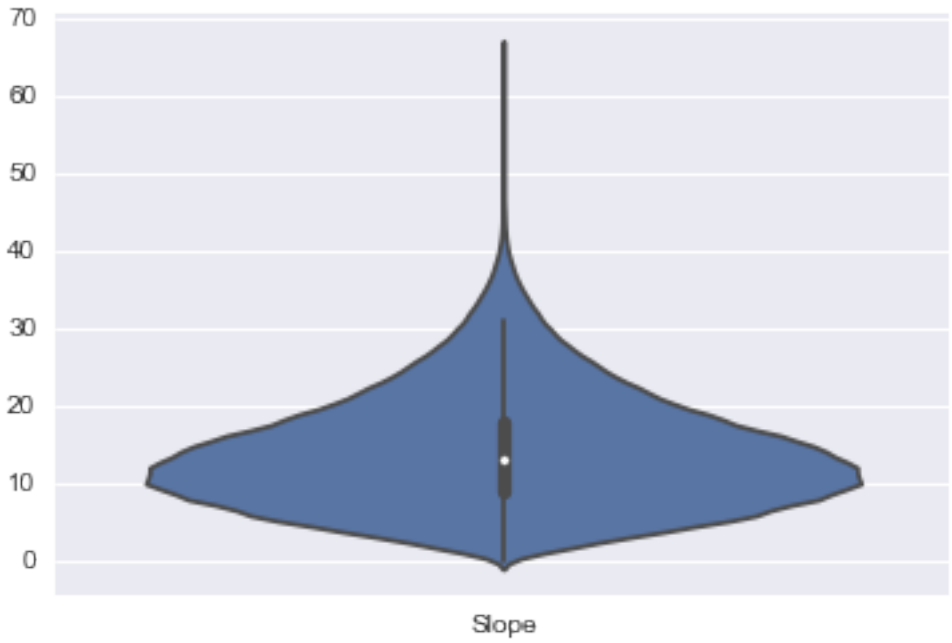
Out[142]: <matplotlib.axes._subplots.AxesSubplot at 0x2cf9c249dd8>




Out[142]:

| | Slope |
|---|---|
| count | 581012.000000 |
| mean | 14.103704 |
| std | 7.488242 |
| min | 0.000000 |
| 25% | 9.000000 |
| 50% | 13.000000 |
| 75% | 18.000000 |
| max | 66.000000 |

Out[142]: <matplotlib.axes._subplots.AxesSubplot at 0x2cf9c6f5828>




Now we cut the whole dataset into training, validation and test.

## 0.3 II. Découpage du jeu de données et ACP

Maintenant en tant que préparation à la modélisation prédictive, on découpe notre jeu de données en jeu d'entraînement, jeu de validation et jeu de test, également en partie explicative (représentée par $X$) et réponse (représentée par $y$). Attention, on va virer les modalités `Soil_Type_7` et `Soil_Type_15`, puisque dans le jeu d'entraînement aucune observation ne possède l'un de ces deux types de terre. Par conséquent, elles ne nous serviraient à rien pour la prédiction et ne poseraient que des problèmes de rang.

```
In [25]: train = range(11340)
         validation = range(11340, 11340+3780)
         test = range(11340+3780, 11340+3780+565892)

         df.drop(["Soil_Type_7", "Soil_Type_15"], axis = 1, inplace = True) # Only run this once

         X_train = df.iloc[train].drop("Cover_Type", axis = 1)
         X_train.shape
         y_train = df.iloc[train]["Cover_Type"]
         y_train.shape

         X_validation = df.iloc[validation].drop("Cover_Type", axis = 1)
         X_validation.shape
         y_validation = df.iloc[validation]["Cover_Type"]
         y_validation.shape
```

```
X_test = df.iloc[test].drop("Cover_Type", axis = 1)
X_test.shape
y_test = df.iloc[test]["Cover_Type"]
y_test.shape
```

```
Out[25]: (11340, 52)
```

```
Out[25]: (11340,)
```

```
Out[25]: (3780, 52)
```

```
Out[25]: (3780,)
```

```
Out[25]: (565892, 52)
```

```
Out[25]: (565892,)
```

Avant de faire de la prédiction, on aimerait bien effectuer une ACP avec les variables quantitatives, afin de révéler quelques pistes dans les données. Pour faire l'ACP, il nous convient de d'abord normaliser les variables quantitatives.

```
In [23]: from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train.iloc[:, 0:10])

X_train_numeric_scaled = scaler.transform(X_train.iloc[:, 0:10])
```

```
Out[23]: StandardScaler(copy=True, with_mean=True, with_std=True)
```

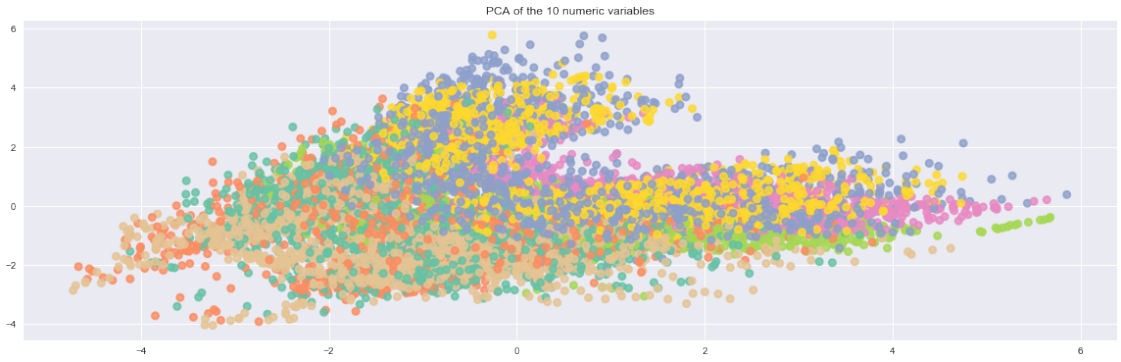
```
In [63]: from sklearn.decomposition import PCA

# PCA with 2 principal components using the 10 quantitative variables.
pca_numeric_only = PCA(n_components = 2)
X_train_numeric_only_scores = pca_numeric_only.fit_transform(X_train_numeric_scaled)
# What's the cumulative ratio of explained variance?
pca_numeric_only.explained_variance_ratio_.cumsum()

plt.figure(figsize=(20,6))
colors = sns.color_palette("Set2", 7)
lw = 2
y_train_to_colors = y_train.map(lambda x: colors[x-1])

plt.scatter(X_train_numeric_only_scores[:, 0], X_train_numeric_only_scores[:, 1], color
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.title('PCA of the 10 numeric variables')
plt.show()
```

```
Out[63]: array([ 0.28635891,  0.5145866 ])
```

```
Out[63]: <matplotlib.figure.Figure at 0x17282db3e10>
```

Out[63]: <matplotlib.collections.PathCollection at 0x172859735f8>

Out[63]: <matplotlib.text.Text at 0x17285948668>




In [55]:
```
import mca

mca_catagorical_only = mca.MCA(X_train.iloc[:, 10:52], ncols=2)
mca_catagorical_only.fs_r(1).shape
```

Out[55]: (11340, 3)

In [153]:
```
mca_catagorical_only.L
X_train_catagorical_only_scores = mca_catagorical_only.cont_r()
X_train.iloc[:, 10:52].head()

plt.figure(figsize=(20,6))

plt.scatter(X_train_catagorical_only_scores[:, 0], X_train_catagorical_only_scores[:,
plt.xlim(-0.00001, 0.00026)
plt.ylim(-0.00001, 0.0002)
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.title('MCA of 2 qualitative variables, individual plot')
plt.show()
```

Out[153]: array([ 0.78866226,  0.52369596,  0.10688624])

Out[153]:

| | Wilderness_Area_1 | Wilderness_Area_2 | Wilderness_Area_3 | Wilderness_Area_4 | \ |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | |
| 1 | 1 | 0 | 0 | 0 | |
| 2 | 1 | 0 | 0 | 0 | |
| 3 | 1 | 0 | 0 | 0 | |
| 4 | 1 | 0 | 0 | 0 | |

| | Soil_Type_1 | Soil_Type_2 | Soil_Type_3 | Soil_Type_4 | Soil_Type_5 | \ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | |

```
1              0              0              0              0              0
2              0              0              0              0              0
3              0              0              0              0              0
4              0              0              0              0              0

   Soil_Type_6       ...       Soil_Type_31  Soil_Type_32  Soil_Type_33  \
0             0      ...                  0             0             0
1             0      ...                  0             0             0
2             0      ...                  0             0             0
3             0      ...                  0             0             0
4             0      ...                  0             0             0

   Soil_Type_34  Soil_Type_35  Soil_Type_36  Soil_Type_37  Soil_Type_38  \
0             0             0             0             0             0
1             0             0             0             0             0
2             0             0             0             0             0
3             0             0             0             0             0
4             0             0             0             0             0

   Soil_Type_39  Soil_Type_40
0             0             0
1             0             0
2             0             0
3             0             0
4             0             0

[5 rows x 42 columns]
```

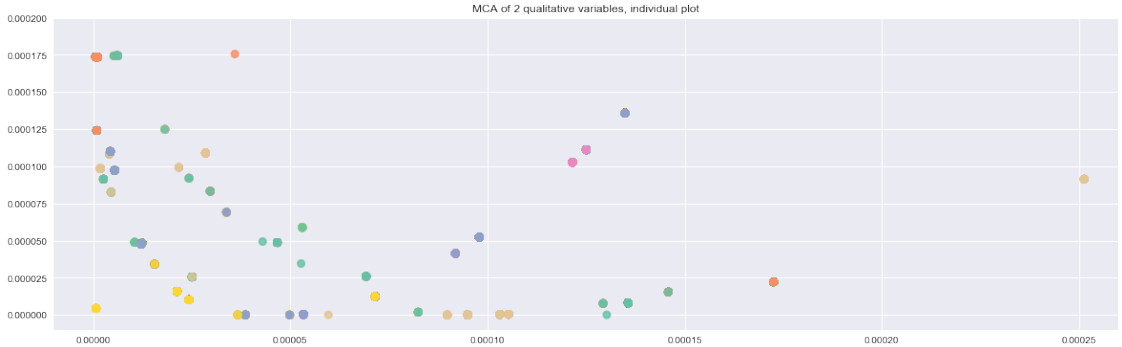
Out[153]: <matplotlib.figure.Figure at 0x17283dff6a0>

Out[153]: <matplotlib.collections.PathCollection at 0x17287af15f8>

Out[153]: (-1e-05, 0.00026)

Out[153]: (-1e-05, 0.0002)

Out[153]: <matplotlib.text.Text at 0x17287ac63c8>



## 0.4 III. Prédiction du type de recouvrement de la forêt par modèles d'analyse discriminante

Dans cette section, on essaie de prédire la variable `Cover_Type` avec toutes les autres. On met un accent sur les méthodes d'analyse discriminante linéaire et quadratique les plus utilisées, auxquelles s'ajoute le modèle des k plus proches voisins. Plus concrètement, on aura

1. Un modèle d'analyse discriminante linéaire ordinaire (ordinary LDA) (exactitude de la prédiction : 58%),
2. Un modèle d'analyse discriminante linéaire avec régularisation automatique (LDA with auto-shrinkage) (57%),
3. Un modèle d'analyse discriminante linéaire avec un terme de régularisation sélectionné par le jeu de validaton (LDA with fixed shrinkage) (47%),
4. Une visualisation en 2D à l'aide de l'analyse discriminante linéaire à rang réduit (reduced rank LDA),
5. Un modèle d'analyse discriminante quadratique ordinaire (ordinary QDA) (8%),
6. Un modèle d'analyse discriminante régularisée (regularized discriminant analysis) (59%),
7. Un modèle des k plus proches voisins (kNN) (67% **meilleure méthode**),
8. Un modèle des k plus proches voisins (kNN) précédé d'une réduction de dimensions par l'ACP (53%).

En général, la famille des méthodes d'analyse discriminante suppose que le vecteur aléatoire, étant donné la classe à laquelle il appartient $X \mid y = i$ , suive une loi gaussienne $N(\mu_i, \Sigma_i), i = 1, ..., 7$. Le classifieur à chercher sous cette hypothèse est alors le classifieur bayésien optimal dont les frontières de décision sont normalement quadratiques. L'analyse discriminante linéaire suppose en plus que les matrices des classes sont commune, en d'autre termes $\Sigma_i = \Sigma, i = 1, ..., 7$, ce qui fait que le classifieur bayésien attribuera à l'observation $X = x$ la classe $k$ pour laquelle

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

avec le terme quadratique simplifié grâce à la variance commune, soit maximisé. Ici $\pi_k$ est la probabilité a priori. Au cas d'effectifs équilibrés comme dans notre exemple, ce terme peut aussi être ignoré. Maintenant les frontières de décision deviennent une séries d'hyperlans linéaires ce qui est en cohérence avec l'explication de Fisher à l'époque, qui cherchait à maximiser le ratio de la variance empirique inter-groupe à la somme des variances empiriques intra-groupe avec des séparateurs linéaires, sans néanmoins introduire d'hypothèse sur la loi.

On commence par la méthode de LDA la plus simple, qui résulte à une qualité de prédiction de 58% sur le jeu de test. Il est important de remarquer que dans cette méthode, différente de beaucoup d'autres, *une normalisation des variables explicatives au préalable n'est pas nécessaire.*

In [107]:
```
### Ordinary LDA

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
# We don't have to precise priors because the frequencies are balanced.
lda_ordinaire = LinearDiscriminantAnalysis()
lda_ordinaire.fit(X_train, y_train)
print("Accuracy : {:.3f}".format(lda_ordinaire.score(X_test, y_test)))
```

Out[107]:
```
LinearDiscriminantAnalysis(n_components=None, priors=None, shrinkage=None,
                           solver='svd', store_covariance=False, tol=0.0001)
```

```
Accuracy : 0.584
```

Ensuite on veut ajouter au modèle un terme de shrinkage $\alpha$, soit en formule $\Sigma = \alpha\Lambda + (1 - \alpha)\Sigma_{emp}$, où $\Lambda$ est une matrice diagonale dont le but est de simplifier la matrice de variance commune à estimer et donc d'éviter le sur-apprentissage. Lorsque $\alpha = 0$ il n'y a aucun shrinkage et quand $\alpha = 1$, la matrice de variance se réduit à une matrice diagonale. On peut soit automatiser l'ajustement de cemaramètre en faisant une validation croisée directement sur le jeu de données, soit prendre une grille de valeur entre 0 et 1 et en choisir la meilleur sur le jeu de validation.

```
In [108]: ### Auto-schrinked LDA

          lda_shrink_auto = LinearDiscriminantAnalysis(solver = "lsqr", shrinkage = "auto")
          lda_shrink_auto.fit(X_train, Y_train)
          print("Accuracy : {:.3f}".format(lda_shrink_auto.score(X_test, Y_test)))

Out[108]: LinearDiscriminantAnalysis(n_components=None, priors=None, shrinkage='auto',
                        solver='lsqr', store_covariance=False, tol=0.0001)

Accuracy : 0.575


In [112]: ### Shrinkage with grid at a pace = 0.05, method least square
          lda_shrink_manuel = pd.Series(np.arange(0, 1, 0.05)).map(lambda i: LinearDiscriminantA
          lda_shrink_manuel.map(lambda model: model.fit(X_train, y_train))

          print("Results on the validation dataset with the optimal alpha = 0 (no shrinkage at a
          lda_shrink_manuel.map(lambda model: model.score(X_validation, y_validation))

          print("Results on the test dataset with the optimal alpha = 0 (no shrinkage at all): "
          lda_shrink_manuel.map(lambda model: model.score(X_test, y_test))

Out[112]: LinearDiscriminantAnalysis(n_components=None, priors=None, shrinkage=0.05,
                        solver='lsqr', store_covariance=False, tol=0.0001)

Out[112]: 0     LinearDiscriminantAnalysis(n_components=None, ...
          1     LinearDiscriminantAnalysis(n_components=None, ...
          2     LinearDiscriminantAnalysis(n_components=None, ...
          3     LinearDiscriminantAnalysis(n_components=None, ...
          4     LinearDiscriminantAnalysis(n_components=None, ...
          5     LinearDiscriminantAnalysis(n_components=None, ...
          6     LinearDiscriminantAnalysis(n_components=None, ...
          7     LinearDiscriminantAnalysis(n_components=None, ...
          8     LinearDiscriminantAnalysis(n_components=None, ...
          9     LinearDiscriminantAnalysis(n_components=None, ...
          10    LinearDiscriminantAnalysis(n_components=None, ...
          11    LinearDiscriminantAnalysis(n_components=None, ...
          12    LinearDiscriminantAnalysis(n_components=None, ...
          13    LinearDiscriminantAnalysis(n_components=None, ...
```

```
14    LinearDiscriminantAnalysis(n_components=None, ...
15    LinearDiscriminantAnalysis(n_components=None, ...
16    LinearDiscriminantAnalysis(n_components=None, ...
17    LinearDiscriminantAnalysis(n_components=None, ...
18    LinearDiscriminantAnalysis(n_components=None, ...
19    LinearDiscriminantAnalysis(n_components=None, ...
dtype: object
```

Results on the validation dataset with the optimal alpha = 0 (no shrinkage at all):

```
Out[112]: 0     0.653704
          1     0.601852
          2     0.594709
          3     0.590476
          4     0.585714
          5     0.581746
          6     0.579365
          7     0.573280
          8     0.571429
          9     0.569577
          10    0.563228
          11    0.552910
          12    0.543651
          13    0.536508
          14    0.521164
          15    0.506349
          16    0.484127
          17    0.463492
          18    0.430688
          19    0.391270
          dtype: float64
```

Results on the test dataset with the optimal alpha = 0 (no shrinkage at all):

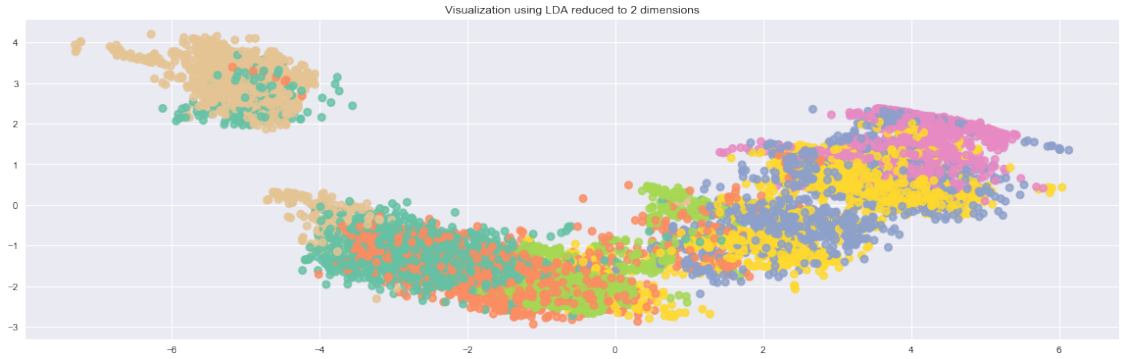
```
Out[112]: 0     0.583815
          1     0.472846
          2     0.467033
          3     0.462898
          4     0.458459
          5     0.453551
          6     0.447831
          7     0.440277
          8     0.431711
          9     0.422089
          10    0.411736
          11    0.398892
          12    0.383886
```

```
13    0.368747
14    0.354893
15    0.338985
16    0.320363
17    0.296597
18    0.265787
19    0.232735
dtype: float64
```

Comme on a vu, dans l'auto-shrinkage le résultat est légèrement pire que dans LDA ordinaire. Dans la méthode de shrinkage manuel, plus grand le shrinkage, pire la qualité de prédiction, tant sur le jeu de validation que sur le jeu de test et l'optimale valeur de $\alpha$ reste à 0, ce qui revient au modèle de LDA ordinaire. En conclusion, la LDA ordinaire sans aucune technique de régularisation marche le mieux parmi les trois variantes. Ceci semble de contradire à notre intuition au sujet de sur-apprentissage au premier coup d'oeil. Mais en fait non. La raison est que le nombre de paramètres à estimer dans les centres des classes et la matrice de variance commune, entre environ 1.000 et 2.000 au total, est assez petit par rapport au nombre d'observations (11.340). Du coup les techniques de régularisation, ayant pour but d'éviter l'e sur-apprentissage, sont redondantes.

On voudrait maintenant visualiser les classes en 2D en maximisant l'effet de séparation. La LDA à rang réduit nous permet de le faire. En effet, quand on fait de la prédiction avec LDA, on cherche pour une nouvelle observation la classe laquelle lui donne la plus grande probabilité conditionnelle. Au cas où les effectifs sont équilibrés, cela revient à calculer les distances entre ce point et les centres des classes et à en prendre la plus petite. Il en résulte que pour la prédiction on n'a pas besoin de garder les données en haute-dimension. Il suffit de regarder les observations dans un sous-espace engendré par les centres de dimension égale au nombre de classes - 1. Plus loin, on peut chercher un sous-espace de dimension encore plus faible (2D par exemple) mais qui garde au mieux l'étendue des classes. Cette optimisation se fait à travers le critère de Fisher.

```
In [124]: ### Reduced rank LDA
          lda_reduced_rank = LinearDiscriminantAnalysis(n_components = 2)
          lda_reduced_rank.fit(X_train, y_train)
          X_train_2dims = lda_reduced_rank.transform(X_train)

          plt.figure(figsize=(20,6))
          plt.scatter(X_train_2dims[:, 0], X_train_2dims[:, 1], color=y_train_to_colors, alpha=.
          plt.legend(loc='best', shadow=False, scatterpoints=1)
          plt.title('Visualization using LDA reduced to 2 dimensions')
          plt.show()

Out[124]: LinearDiscriminantAnalysis(n_components=2, priors=None, shrinkage=None,
                        solver='svd', store_covariance=False, tol=0.0001)

Out[124]: <matplotlib.figure.Figure at 0x172861b1940>

Out[124]: <matplotlib.collections.PathCollection at 0x1728711d898>

Out[124]: <matplotlib.text.Text at 0x1728705b8d0>
```




Le graphe de LDA à rang réduit a revelé quelques pistes dans les données, par exemple un sous-groupe isolé composé notamment des terres kakis et vertes, ce qu'on ne voiyait pas avant dans le graphe d'ACP de 2 dimensions. Cette visualisation explique d'ailleurs bien pourquoi kNN marche mieux que les LDA et QDA. En effet, les classes sont loin d'être des lois gaussiennes ; certaines d'entre elles ont plusieurs clusters. Si on voulait rester avec LDA, on pourrait modéliser avec plus de sous-groupes.

Les derniers membres de cette famille de méthodes qu'on va regarder sont les méthodes d'analyse discriminante quadratique. N'ayant aucune contraînte dans les matrices de variance des classes, elle marche assez bien quand le nombre d'observations est suffisant et assez mal dans le cas contraire. Dans notre exemple, il y aura presque 10.000 paramètres à estimer quand les variances des classes peuvent être différentes, ce qui entraîne que la QDA sans régularisation va donner un mauvais résultat, comme on le verra au premier modèle suivant. Par contre, si on consière un terme de régularisation, même s'il est aussi simple qu'une matrice d'identité, soit en formule : $\Sigma_i = \alpha I + (1-\alpha)\Sigma_{i,emp}$, la qualité de prédiction peut même dépasser celle de la LDA. Comme toujours, on choisit ce paramètre $\alpha$ sur le jeu de validation. Malheureusement la librairie `scikit-learn` de `Python` ne propose que cette simple régularisation dans QDA. `R`, en revanche, sait faire mieux : le package `rda` implémente la double régularisation proposé dans [1], qui délivrerait probablemant un résultat encore légèrement meilleur.

Dans tous les cas, sous contexte de cette simple régularisation, on peut arriver à une exactitude de 59%.

```
In [129]: from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

          qda_shrink_manuel = pd.Series(np.arange(0, 1, 0.05)).map(lambda i: QuadraticDiscrimina
          qda_shrink_manuel.map(lambda model: model.fit(X_train, y_train))
          print("Results on the validation dataset with the optimal alpha = 0.05 (few shrinkage)
          qda_shrink_manuel.map(lambda model: model.score(X_validation, y_validation))
          print("Results on the test dataset: ")
          qda_shrink_manuel.map(lambda model: model.score(X_test, y_test))
          print("Accuracy with the chosen alpha = 0.05 is 59%. But with alpha = 0.95 (almost ide

Out[129]: 0     QuadraticDiscriminantAnalysis(priors=None, reg...
          1     QuadraticDiscriminantAnalysis(priors=None, reg...
          2     QuadraticDiscriminantAnalysis(priors=None, reg...
          3     QuadraticDiscriminantAnalysis(priors=None, reg...
```

```
4     QuadraticDiscriminantAnalysis(priors=None, reg...
5     QuadraticDiscriminantAnalysis(priors=None, reg...
6     QuadraticDiscriminantAnalysis(priors=None, reg...
7     QuadraticDiscriminantAnalysis(priors=None, reg...
8     QuadraticDiscriminantAnalysis(priors=None, reg...
9     QuadraticDiscriminantAnalysis(priors=None, reg...
10    QuadraticDiscriminantAnalysis(priors=None, reg...
11    QuadraticDiscriminantAnalysis(priors=None, reg...
12    QuadraticDiscriminantAnalysis(priors=None, reg...
13    QuadraticDiscriminantAnalysis(priors=None, reg...
14    QuadraticDiscriminantAnalysis(priors=None, reg...
15    QuadraticDiscriminantAnalysis(priors=None, reg...
16    QuadraticDiscriminantAnalysis(priors=None, reg...
17    QuadraticDiscriminantAnalysis(priors=None, reg...
18    QuadraticDiscriminantAnalysis(priors=None, reg...
19    QuadraticDiscriminantAnalysis(priors=None, reg...
dtype: object
```

Results on the validation dataset with the optimal alpha = 0.05 (few shrinkage):

```
Out[129]: 0     0.422222
          1     0.729630
          2     0.728836
          3     0.720899
          4     0.712169
          5     0.703175
          6     0.696296
          7     0.690212
          8     0.685185
          9     0.678571
          10    0.674074
          11    0.669048
          12    0.667460
          13    0.660582
          14    0.659788
          15    0.656085
          16    0.647354
          17    0.639683
          18    0.632275
          19    0.628307
          dtype: float64
```

Results on the test dataset:

```
Out[129]: 0     0.079770
          1     0.593665
          2     0.593306
```

```
3     0.590671
4     0.587345
5     0.583933
6     0.581537
7     0.580809
8     0.581374
9     0.582643
10    0.584995
11    0.587607
12    0.590772
13    0.594142
14    0.597697
15    0.601604
16    0.605868
17    0.610657
18    0.616171
19    0.622993
dtype: float64
```

```
Accuracy with the chosen alpha = 0.05 is 59%. But with alpha = 0.95 (almost identity matrix), it
```

Finalement, on touche la méthode des k plus proches voisins. Comme d'habitude, on choisit le nombre de voisins optimal sur le jeu de données de validation. Ici, on étudie deux variantes : dans le premier cas on ajuste directement le modèle sans réduction de dimensions des données. Mais sachant que la méthode kNN est basée sur la distance euclidienne et que cette méthode va donc subir du *fléau de la dimension*, on essaie dans le deuxième cas de faire d'abord une ACP pour réduire le nombre de dimensions, avant de faire kNN. Vu que le calcul avec cette méthode est très coûteux en termes de temps, on va pas prendre encore une fois une grille pour trouver le meilleur nombre de composantes dans l'ACP. Par contre on va prendre arbitrairement un nombre de composantes(ici : 5).

Il s'avère que le kNN direct marche mieux que celui précédé d'une réduction de dimensions par l'ACP. Avec le nombre de voisins égal à 1, cette prémière atteint visuellement son exactitude optimale 67%. La raison pour laquelle l'ACP n'a pas porté d'amélioration est que sa variance éxpliquée est répartie de façon très équilibrée dans un grand nombre de composantes.

```
In [146]: ### Direct k nearest neighbours
          scaler_all_variables = StandardScaler()
          X_train_scaled = scaler_all_variables.fit_transform(X_train)

          from sklearn.neighbors import KNeighborsClassifier
          knn_nb_neighbours = pd.Series(np.arange(1, 10, 2)).map(lambda i : KNeighborsClassifier
          knn_nb_neighbours.map(lambda model: model.fit(X_train_scaled, y_train))
          print("Results on validation set with number of neighbours = 1, 3, 5, 7, 9")
          knn_nb_neighbours.map(lambda model: model.score(scaler_all_variables.transform(X_valid
```

```
Out[146]: 0    KNeighborsClassifier(algorithm='auto', leaf_si...
          1    KNeighborsClassifier(algorithm='auto', leaf_si...
          2    KNeighborsClassifier(algorithm='auto', leaf_si...
```

```
3    KNeighborsClassifier(algorithm='auto', leaf_si...
4    KNeighborsClassifier(algorithm='auto', leaf_si...
dtype: object
```

Out[146]:
```
0    0.793122
1    0.780952
2    0.762963
3    0.756614
4    0.749471
dtype: float64
```

In [147]:
```
print("Result of 1 nearest neighbour on the test set: ")
knn_nb_neighbours[0].score(scaler_all_variables.transform(X_test), y_test)
print("Result of 3 nearest neighbours on the test set: ")
knn_nb_neighbours[1].score(scaler_all_variables.transform(X_test), y_test)

# np.arange(0.1, 1, 0.1)
# sum(i for i in list(range(5)))
# dd = list(map(lambda x: x+1, list(range(5))))
# dd
# pd.Series(range(5)).map(lambda x: x+1)
# de = [range(5)].map(lambda x: x+1)
# de
# [range(5)] # yi
```

```
Result of 1 nearest neighbour:
```

Out[147]:
```
0.6703151838159932
```

```
Result of 3 nearest neighbours:
```

Out[147]:
```
0.63559654492376638
```

In [149]:
```
### k nearest neighbours preceeded by a PCA of 5 components

pca_all_variables = PCA(n_components = 5)
X_train_PCAed = pca_all_variables.fit_transform(X_train_scaled)
pca_all_variables.explained_variance_ratio_.cumsum()

knn_PCAed = KNeighborsClassifier(n_neighbors=1)
knn_PCAed.fit(X_train_PCAed, y_train)
print("Results on respectively the validation and test set with nubmer of neighbours =
knn_PCAed.score(pca_all_variables.transform(scaler_all_variables.transform(X_validatio
knn_PCAed.score(pca_all_variables.transform(scaler_all_variables.transform(X_test)), y

knn_PCAed = KNeighborsClassifier(n_neighbors=3)
knn_PCAed.fit(X_train_PCAed, y_train)
```

```
print("Results on respectively the validation and test set with nubmer of neighbours =
knn_PCAed.score(pca_all_variables.transform(scaler_all_variables.transform(X_validatio
knn_PCAed.score(pca_all_variables.transform(scaler_all_variables.transform(X_test)), y
```

```
Out[149]: array([ 0.0818578 ,  0.13956369,  0.18547225,  0.22465215,  0.25496726])
```

```
Out[149]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=1, p=2,
                    weights='uniform')
```

```
Out[149]: 0.69417989417989423
```

```
Out[149]: 0.53159790207318713
```

```
Out[149]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=3, p=2,
                    weights='uniform')
```

```
Out[149]: 0.68968253968253967
```

```
Out[149]: 0.52386321064796815
```

Pour conclure, la meilleure méthode qu'on a trouvée est celle du seul plus proche voisin ($k = 1$). Son exactitude sur le jeu de test est 67%.