IFT 3700 - Travail 2

Gabriel Ménard, Kevin Lessard

December 2021

Question 1

A - Trouver les 1000 paires d'étoiles les plus proches en termes de distance euclidienne

Premièrement, il faut remarquer qu'il pourrait y avoir une certaine perte d'information. En effet, on n'est pas certain de pouvoir calculer la distance entre chacune des étoiles, parce qu'il faudrait pouvoir mettre toutes les données sur un même serveur. Il se peut donc que certaines paires qui seraient dans les 1000 plus proches paires d'étoiles ne soient pas dans le résultat final, mais nous allons avoir une bonne approximation des 1000 plus proches paires d'étoiles. Il faudra donc commencer par trouver les 1000 paires sur chacun des serveurs. Nous allons ensuite prendre toutes ces listes de 1000 paires, qui contiendront aussi la distance entre ces deux points, et les mettre dans la même liste. Ensuite, on fait un Quicksort en les mettant de la plus petite distance à la plus grande et on garde les 1000 premières valeurs de ce vecteur trié. Pour trouver la paires des points les plus proches, on va utiliser l'algorithme de la page Wkipedia donnée dans l'énoncer qui se fait en $\mathcal{O}(n(\log n)^2)$. Appelons cet algorithme PlusProchePaire. Disons que la variables "serveurs" est une liste contenant les serveurs ayant un attribut "data" qui contiennent tous les individus et où chaque individus a des attributs.

Pseudo code:

```
plus_proches_paires_tous_serveurs_confondus = [ ]
for serveur in serveurs:

    plus_proches_paires_tous_serveurs_confondus.append(PlusProchePaire(serveur.data))

plus_proches_paires_tous_serveurs_confondus.sort()
return(plus_proches_paires_tous_serveurs_confondus[0:1000])
```

B - Compter le nombre d'étoile dans chaque catégorie

Ceci ressemble beaucoup à compter le nombre de mot dans un document. Nous allons donc utiliser la même logique. On traverse tous les 1200 jeu de données de façon parallèle et on compte le nombre d'occurrences de chacune des classes d'étoiles. On va mettre ces occurrences dans des paires (CLASSE, OCCURRENCE). Nous aurons donc 1200 listes de paires de taille 10 après cette étape. Ensuite, il faut regroupe tous ces listes sur un même serveur et additionner les occurrences. Cette étape ne sera pas très longue, puisqu'elle ne contient vraiment pas beaucoup de données. Nous aurons finalement une liste de paires de taille 10 qui contient les occurrences de chaque classe d'étoiles. Pour ce qui est du pseudo code, on peut se fier au note de cours donnees massives. Nous allons utiliser les méthode map(key, value) et reduce(key, values). "serveur.classes" est l'attribut contenant un liste des classes du serveur en question

Pseudo code:

```
vecteur_de_tous_les_serveurs = [ ]
vecteur_final= [ ]
for serveur in serveurs:

    vecteur_de_tous_les_serveurs.append(map(serveur, serveur.classes))

for i in range(10):

    vecteur_final.append(reduce(i,vecteur_de_tous_les_serveurs[1]))

return(vecteur_final)
```

C - Classifieur qui étant donnés les autres caractéristiques prédit la catégorie de l'étoile

K-mean. On calcule la moyenne sur chaque serveur et apres on calcule la moyenne des moyenne. KNN serait bcp trop lent ici.

On va utiliser Le classifieur de la plus proche moyenne. Il va être relativement long a entrainer, mais les predictions se feront relativement rapidement. Nous avions pensé à utiliser K-NN au début, mais les prédictions seraient beaucoup trop lentes et chaque prédictions demanderaient beaucoup trop d'énergie. Le classifieur de la plus proche moyenne ne fera que comparer un point avec 10 autres points, soient les 10 moyennes des classes. Cependant, en utilisant la plus proche moyenne, on fait l'hypothèse que les classes sont assez distinctes. En d'autres mots, on fait l'hypothèse que les classes sont linéairement séparables. Préférablement, il faudrait vérifier cette hypothèse. Nous avons quand même une autre astuce utile : La validation croisée. En entrainant plusieurs modèle en séparent les données en données d'entrainements et de validations de plusieurs

manières différentes, nous allons pour voir si notre précision est relativement bonne. Si elle l'est on sait que notre modèle est bon sans avoir à représenter les 7300 milliard d'étoiles sur un graphique pour voir si les groupes sont hétérogènes. Pour utiliser cette méthode, nous allons trouver la moyenne de chaque classe sur chacun des serveur et ensuite calculer la moyenne de ces moyennes.

Pseudo code:

```
moyennes = {j:0 for j in range(10)}
for server in servers:{

    dictionnaire_classe = {i:0 for i in range(10) }:


    for x in server.data:{


    dictionnaire_classe[x.classe] = dictionnaire_classe[x.classe] + 1

}

    for i in range(10):{

    moyenne[i] = moyenne[i]+dictionnaire_classe[x.classe]

    }
} for i in range(10):{

    moyenne[i] = moyenne[i]/nombre_etoiles
    }
```

Question 2

1

c)

Après l'élimination des pays qui ont 12 ou plus valeurs manquantes dans leurs colonnes, on a obtenu 159 pays restant. On a ainsi insérer ces données dans le fichier $dataA.csv$.

d)

Après avoir calculer les 2 régressions, on a mit le résultat dans le fichier $dataB.csv$.

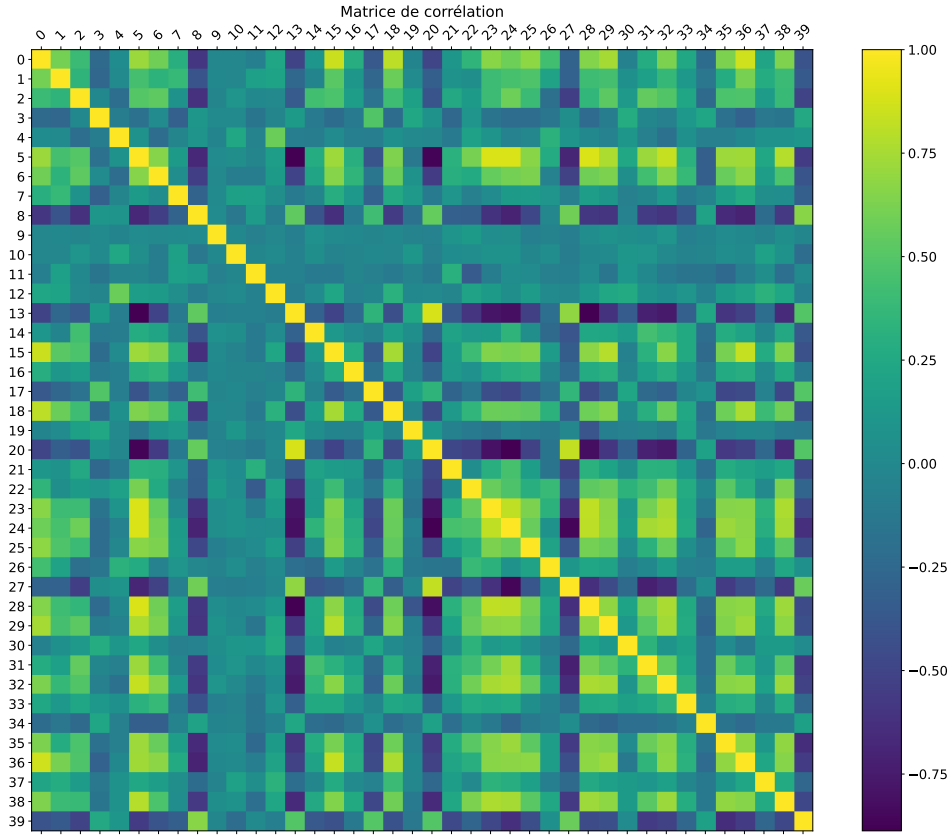
e)

Après avoir calculer les données binarisées, on a mit le résultat dans le fichier $dataC.csv$.

2

a)

Voici les corrélations calculés avec $pandad.DataFrame.corr()$:



Il y a plusieurs corrélations intéressantes qu'on peut s'amuser à regarder au travers de ce graphique disponible sous *correlations.pdf*.

b)

Les colonnes représentants la corrélation la plus forte pour chaque colonne en ordre est donné par:

```
{
    [36, 36, 8, 17, 12, 23, 36, 1, 24, 22, 4, 22, 4,
    28, 31, 0, 6, 39, 0, 28, 24, 27, 38, 5, 5, 36, 0,
    24, 5, 15, 13, 24, 5, 8, 6, 5, 0, 18, 5, 8]
}
```

disponible dans le fichier *max_corr.json*.

On peut confirmer ce résultat en se fiant à la matrice de corrélation au-dessus.

c)

L'ordre décroissante des corrélations moyennes pour chaque colonnes est:

```
{
    [5.0, 24.0, 28.0, 23.0, 32.0, 36.0, 13.0, 20.0,
    38.0, 29.0, 35.0, 15.0, 8.0, 0.0, 18.0, 27.0,
    6.0, 39.0, 25.0, 31.0, 2.0, 22.0, 1.0, 17.0,
    21.0, 33.0, 14.0, 16.0, 37.0, 34.0, 3.0, 26.0,
    7.0, 30.0, 12.0, 11.0, 4.0, 19.0, 10.0, 9.0]
}
```

disponible dans le fichier *ordre.json*.

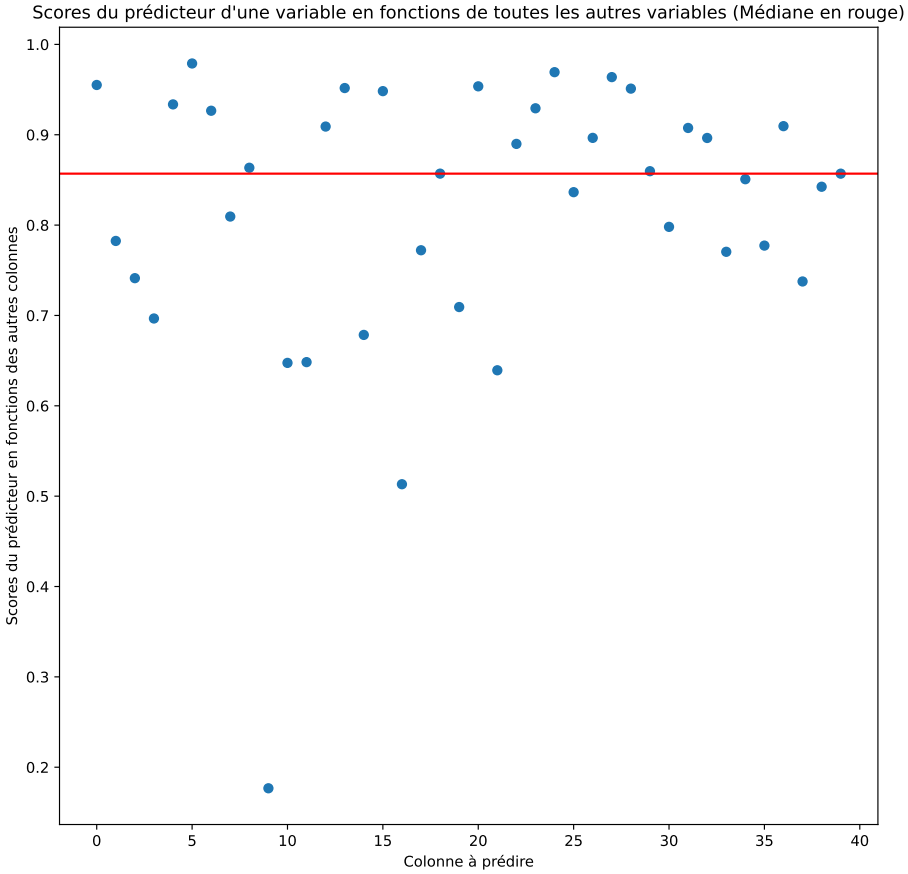
La colonne avec la meilleur corrélation moyenne est la colonne de HDI, soit l'index de développement humain, qui est déjà une donnée qui permet d'en résumer d'autres.

3

—Régression linéaire—

a)

Nous avons d'abord normalisé les colonnes et fait une régression pour chaque variable (colonne) individuelle en fonctions des autres variables. Nous avons ainsi obtenu un score de la régression pour chacune des colonnes.



On remarque que pour la variable à prédire #9, soit la variable relié au pourcentage de christianité, est très difficile à prédire (avec raison) à partir de toutes les autres variables avec un score de 0.1766.

Tandis que, pour la variable à prédire #5, soit la variable relié à l'index de développement humain, est la plus facile à prédire à partir de toutes les autres variables avec un score de 0.9788. En réalité, cet index est déjà relié à certaines autres variables par définition. Donc, cette réponse a du sens.

b)

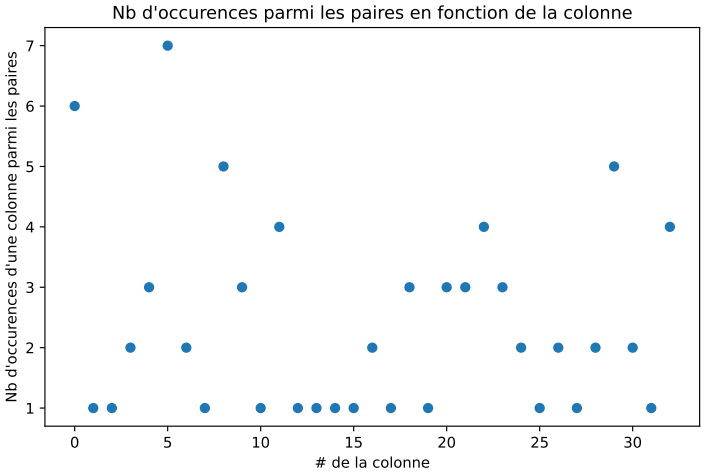
Les paires de colonnes qui nous permettent de prédire avec la meilleure précision en fonction de chaque colonne sont:

```
{
    [[26, 36], [11, 36], [8, 31], [17, 34], [12, 26],
    [13, 23], [25, 32], [1, 3], [36, 39], [22, 25],
    [4, 39], [20, 22], [4, 37], [20, 28], [26, 31],
```

```
    [0, 29], [6, 25], [3, 38], [0, 37], [8, 28],
    [13, 27], [11, 39], [11, 13], [5, 34], [5, 27],
    [0, 6], [0, 36], [0, 24], [5, 19], [5, 15],
    [5, 36], [2, 13], [4, 5], [7, 8], [24, 27],
    [5, 39], [0, 8], [18, 26], [22, 24], [8, 21]]
}
```

disponibles dans le fichier *lineaire_paires_colonnes.json*.

On peut remarquer que certains nombres reviennent plus souvent.



En effet, on remarque qu'il y a des variables qui sont beaucoup plus populaire que d'autres et certaines ne sont même pas présentes, d'où le fait que le tableau ne va jusqu'à 33. et celui-ci n'est pas représentatif de la colonne en question.

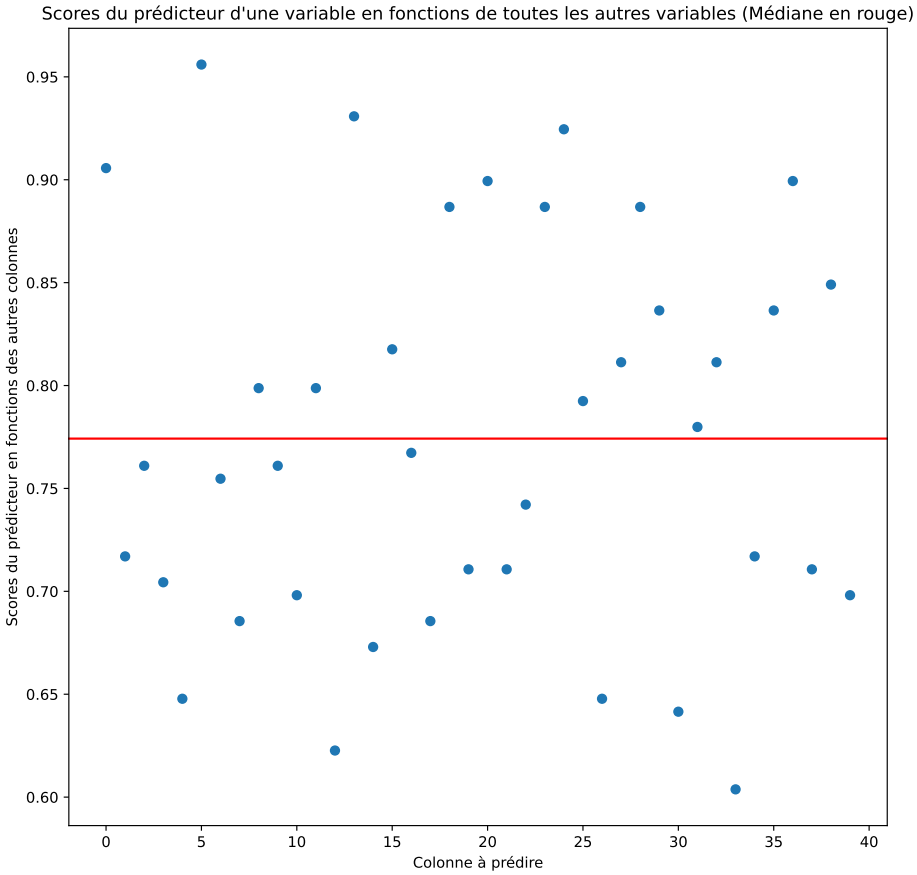
c)

La meilleure paire de colonne qui permet de prédire les autres colonnes est la colonne 0, soit 'GDP per capita' avec la colonne 24, soit 'Median age', avec un score moyen de prédictions pour chacune des colonnes de 0.3737. On remarque alors qu'il y a très probablement un problème puisque ces scores moyens ne sont vraiment pas élevés. Il est possible que soit ce n'est vraiment pas la bonne technique pour essayer de résoudre un tel problème, soit notre approche est la mauvaise, soit ce n'est pas un bon problème pour un prédicteur. Nous allons ainsi comparé ces résultats avec le prédicteur bayésien qui va suivre.

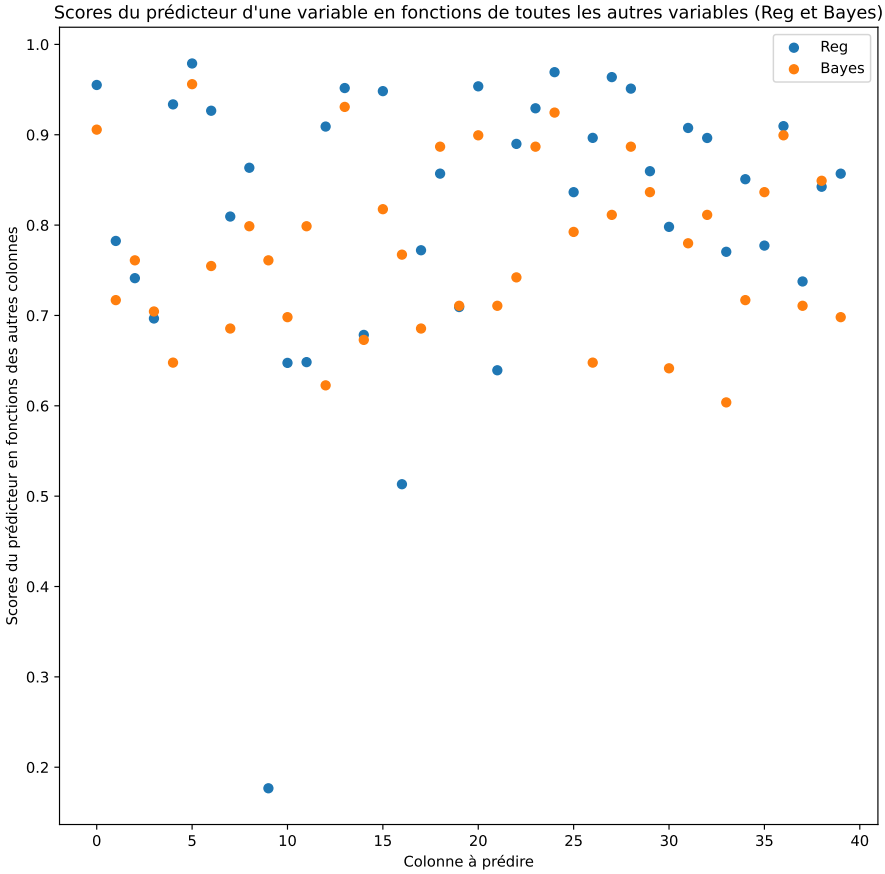
—Classifieur bayésien—

a)

Puisque les données sont binarisées, nous avons ainsi obtenu un score de la régression pour chacune des colonnes.



On remarque que les précisions sont distribués très aléatoirement dans le graphe, mais on remarque que la variable 5 est encore très élevé. On peut alors comparer les 2 graphes:



On voit alors que les scores ont beaucoup plus de variances pour la régression linéaire que pour le prédicteur bayésien. On a aussi que les 2 plus bas scores pour la régression linéaire vont avoir un impact immense sur la moyenne de ces scores. Bref, les scores du prédicteur bayésien sont un peut mieux répartis que les scores pour la régression. Une bonne majorité des scores pour la régression semblent plus élevés, mais les autres scores ne sont tout simplement pas acceptables (ex: 9 et 17).

b)

Les paires de colonnes qui nous permettent de prédire avec la meilleure précision en fonction de chaque colonne sont:

```
{
    [[26, 36], [11, 36], [8, 31], [17, 34], [12, 26],
    [13, 23], [25, 32], [1, 3], [36, 39], [22, 25],
    [4, 39], [20, 22], [4, 37], [20, 28], [26, 31],
    [0, 29], [6, 25], [3, 38], [0, 37], [8, 28],
```

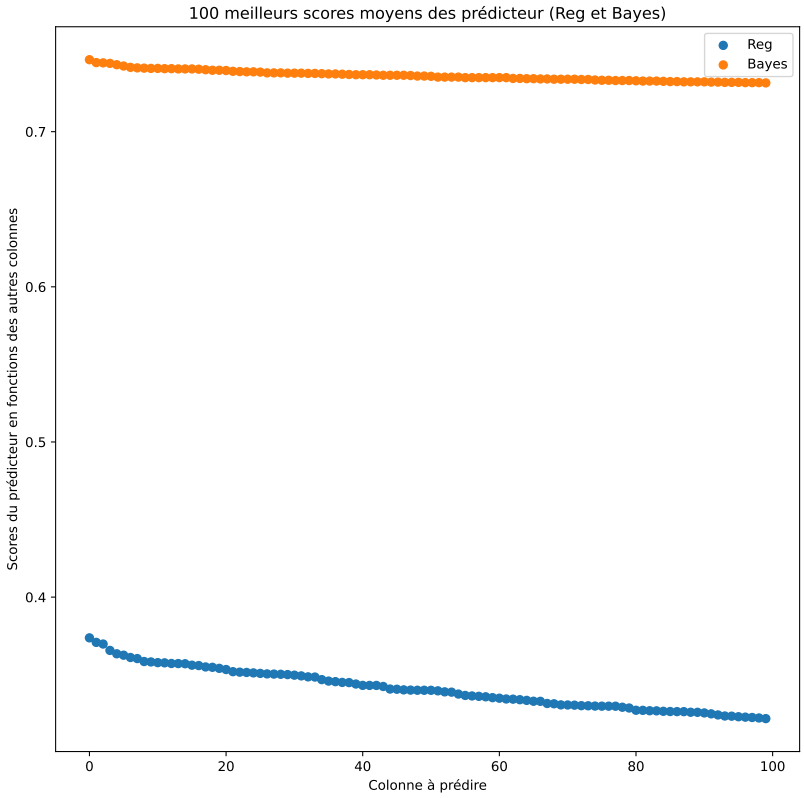
```
6       [13, 27], [11, 39], [11, 13], [5, 34], [5, 27],
7       [0, 6], [0, 36], [0, 24], [5, 19], [5, 15], [5, 36],
8       [2, 13], [4, 5], [7, 8], [24, 27], [5, 39], [0, 8],
9       [18, 26], [22, 24], [8, 21]]
10  }
```

disponibles dans le fichier *bayesien_paires_colonnes.json*.

c)

La meilleure paire de colonne qui permet de prédire les autres colonnes est la colonne 5, soit 'HDI', ainsi que la colonne 17, soit 'Income equality', avec un score moyen de prédictions pour chacune des colonnes de 0.7463.

On remarque que les scores moyens pour le prédicteur bayésien sont nettement supérieurs au prédicteur provenant de la régression linéaires pour la prédiction des valeurs d'une colonne.

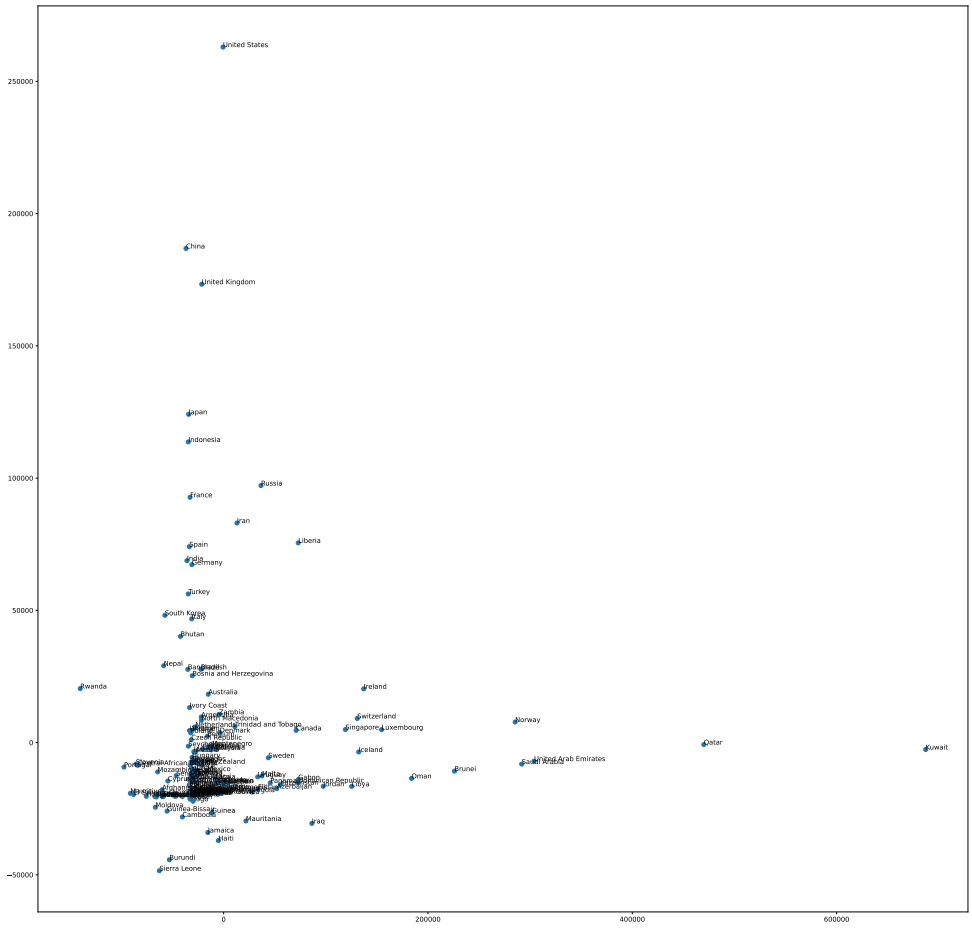


Ainsi, on peut conclure qu'il est beaucoup plus simple pour le prédicteur de prédire une classification par rapport à la médiane que les données elle-mêmes.

4

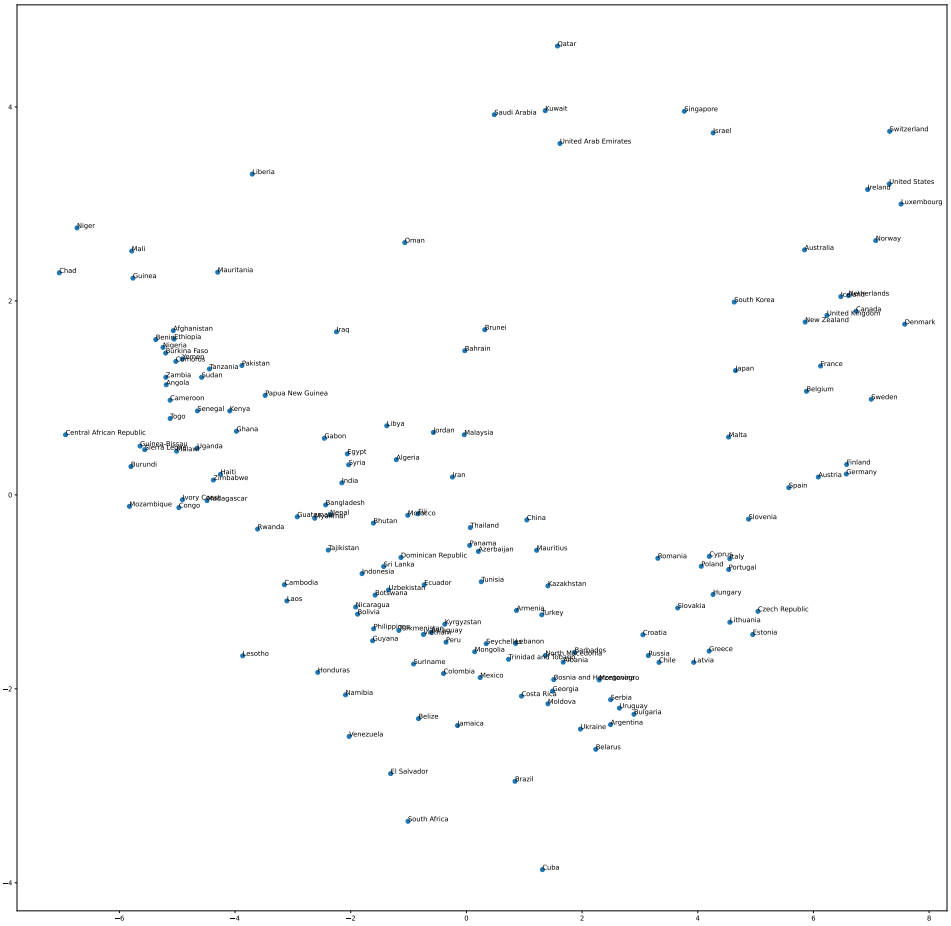
a)

D'abord, on a essayer un PCA a deux composantes avec dataB (les données remplis après les 2 régressions). Ainsi, nous avons obtenu ceci:



Le fichier est disponible dans la remise en plus grand format avec plus de séparation entre les points. Voir *pca_50x50.pdf*

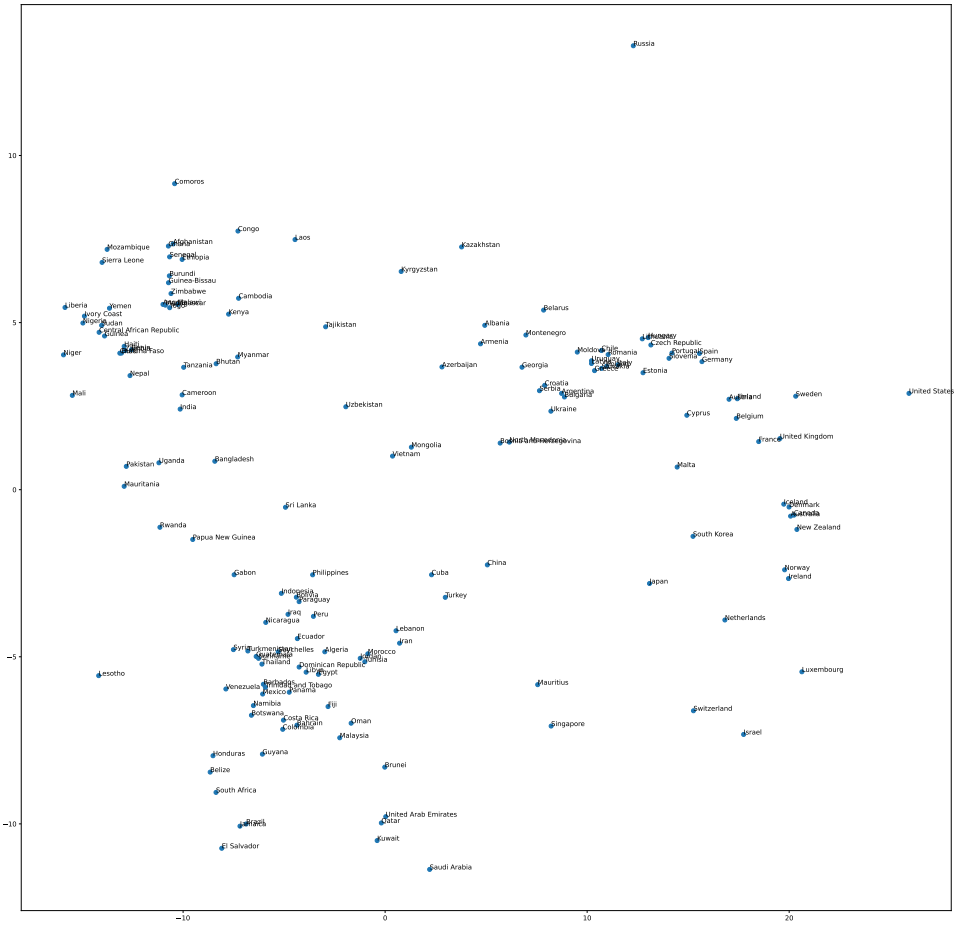
On a alors remarqué qu'on avait un gros problème d'échelle. On peut quand même voir une certaine relation avec les pays riche, mais il n'y a pas beaucoup de répartition entre les pays moins riches. Ce graphe est trop difficile a analyser. Ainsi, nous avons normalisé les données et nous avons refait un PCA à deux composantes. Par normalisation, on veut dire ici standardisation puisque l'on soustrait la moyenne et divise par l'écart-type.



Le fichier est disponible dans la remise en plus grand format avec plus de séparation entre les points. Voir *pca_norma_*50x50*.pdf*

On remarque bel-et-bien une meilleur répartition des pays. On remarque que les pays plus développés (enrichies) se retrouvent à la droite. Sinon, peut-être que des notions de géopolitiques nous aideraient mieux à y voir une relation.

Pour le plaisir, nous avons aussi essayer avec Isomap à 2 composantes et 5 voisins pour voir si on pouvait obtenir quelque chose de mieux à l'aide des relations non linéaires.



Le fichier est disponible dans la remise en plus grand format avec plus de séparation entre les points. Voir *isomap_norma_*50*x*50*.pdf*

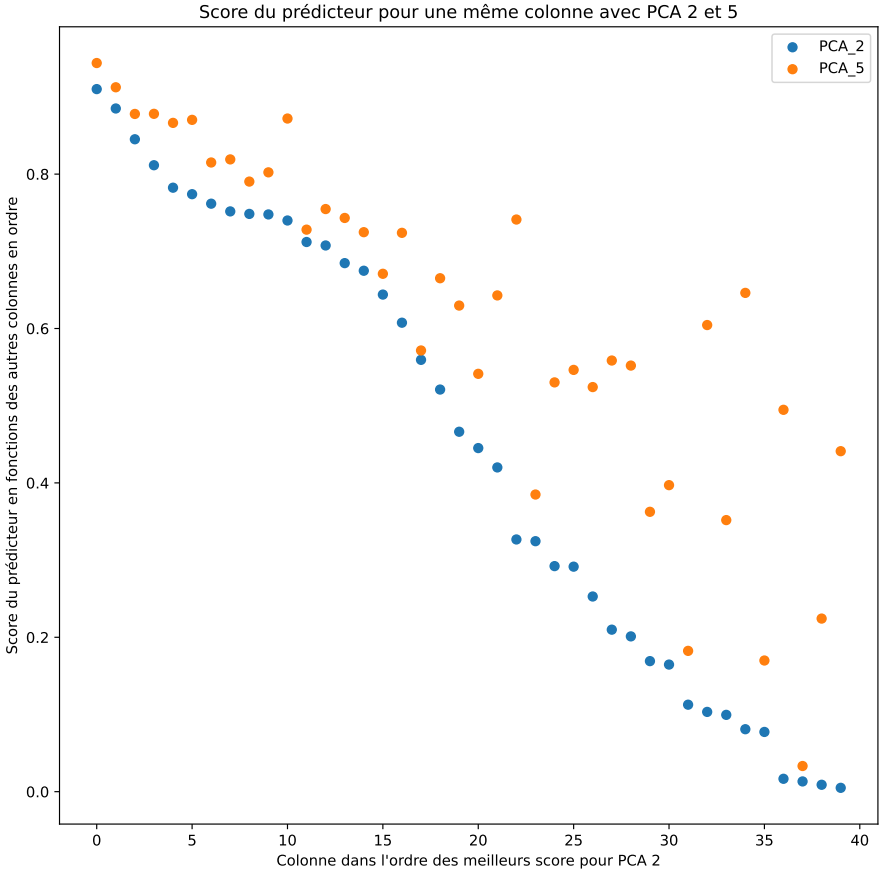
On remarque une formule généralement semblable avec les pays qui tombent a peu près à la même place qu'ils étaient en PCA normalisé. Bref, il est certain que l'impact économique d'un pays, ou plutôt son niveau de développement) soit une partie importante de l'une des composantes principales trouvée.

b)

Nous avons d'abord analysé les résultats sans normalisation en ayant nos PCA à 2 et à 5 composantes pour faire des régression sur chaque colonne en fonction de toutes les autres colonnes comme fait précédemment. Nous avons alors obtenus un score moyen pour PCA_2 de 0.13989 alors que PCA_5 a obtenu 0.3877.

On voit bien qu'il y a un certain problème avec nos résultats. Nous avons alors

utilisé les données normalisées et nous avons eu pour PCA_2 un score moyen de 0.4488 et pour PCA_5 un sore moyen de 0.6147. C'est résulat sont nettement supérieur, montrant que SKlearn ne standardise pas les données qu'on lui donne. Ainsi, en revenant vers le but de l'analyse, on remarque que 2 composantes n'est nettement pas assez pour bien prédire les données des colonnes et qu'il serait préférable d'utiliser 5 composantes principales plutôt que 2. Voici un graphique montrant dans l'ordre des colonnes ayant les meilleurs score pour PCA 2 la différence avec PCA 5 pour la même colonne:



On remarque que pour une grande majorité des colonnes, il y a une nette amélioration des scores avec une PCA à 5 composantes. Ainsi, c'est bien amusant de pouvoir observer les données en 2 dimensions, mais ce n'est pas pratique pour prédire dans le cas où on possède 40 variables. On aurait tendance à vouloir soit augmenter le nombre de composantes principales, soit essayer quelque chose d'autre. On avait vu que Isomap ressemblait beaucoup à PCA2 normalisé. Ainsi, je ne crois pas que les scores auraient suffisamment varié pour l'utilisation d'Isomap.

5

b)

Nous avons bel-et-bien réussi à faire le modèle. On pensait alors utiliser *.score()* avec chaque combinaison possible qu'on a calculé de cette manière:

In [139]:
```
comb = list(itertools.combinations(range(40), 2))
```

In [140]:
```
dict_comb_baye = {}
```

In [141]:
```
comb_baye = []
```

In [142]:
```
for x, y in comb:
    arr1 = []
    arr2 = []
    arr3 = []
    arr4 = []
    for i in range(0,x):
        arr1.append(None)
        arr2.append(None)
        arr3.append(None)
        arr4.append(None)
    arr1.append(0)
    arr2.append(0)
    arr3.append(1)
    arr4.append(1)
    for i in range(x+1,y):
        arr1.append(None)
        arr2.append(None)
        arr3.append(None)
        arr4.append(None)
    arr1.append(0)
    arr2.append(1)
    arr3.append(0)
    arr4.append(1)
    for i in range(y,39):
        arr1.append(None)
        arr2.append(None)
        arr3.append(None)
        arr4.append(None)
    comb_baye.append([arr1,arr2,arr3,arr4])
```

In [149]:
```
print(comb_baye[779])
```

```
[[None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, 0, 0], [None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, N
one, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, 0, 1], [None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, N
one, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, 1, 0], [None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, N
one, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, 1, 1]]
```

In [150]:
```
print(comb_baye[0])
```

```
[[0, 0, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None], [0, 1, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, N
one, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None], [1, 0, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, N
one, None, None, None, None, None, None, None, None, None, None, None, None, None], [1, 1, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, N
one, None, None, None, None, None, None, None, None]]
```

On allait par la suite utiliser notre réseau disponible dans *reseau.json* pour chacune des 780 combinaisons possibles et on aurait ensuite trouvé le score moyen en comparant chaque prédicteur sur toutes les colonnes avec les autres prédicteurs. Ainsi, on aurait ensuite pu faire comme au 3.c et commenter ces résultats.