Rapport Projet Apprentissage Automatique Desire Brondon Serge

December 2, 2020

1 INTRODUCTION

La performance énergetique est un paramètre important à prendre en compte lors de la construction d'un bâtiment. Elle dépend des caractéristiques du bâtiment ainsi que des conditions climatiques du milieu. Afin d'optimiser sa consomation énergetique, il serait important de quantifier l'influence de ces données sur la performance énergetique. La prédiction de cette perfomance à partir des données mesurées sur un échantillon d'immeuble à l'aide des techniques statistiques et d'apprentissage automatique, pourrait permettre d'optimiser la consommation énergetique des immeubles.

Dans ce calepin nous présentons les résultats de prédiction de performance énergétique des bâtiments, obtenus à l'aide de plusieurs techniques d'apprentissage automatique sur un échantillons de 780 données.

La première partie est consacrée à l'analyse exploratoire des données et visualisations ainsi qu'à l'application des techniques statistiques de clustering (Kmeans,DBSCAN) et d'analyse en composantes principales sur l'ensemble de donées. Dans la seconde partie, nous développons plusieurs méthodes de classification directe et indirecte tout en comparant les performances, dans **l'objectif de trouver le modèle optimal de prédiction de l'efficacité énergetique des bâtiments.**

Ainsi nous avons implémenté les algorithmes de **Regression linéaire/logistique, *optimal tree, random forest, boosting et SVM***

[3]:
```
#importing the data
df_energy= pd.read_csv("DataEnergy.csv")
```

2 EXPLORATION DE DONNEES

Description des données

Les variables de l'ensemble des données sont les suivantes:

- **Relative.compactness** La compacité relative du bâtiment
- **Surface.area** La surface totale de l'immeuble
- **Wall.area** La surface des murs
- **Roof.area** La surface du toit.
- **Overall.height** La hauteur totale du bâtiment : 3.5 m et 7m
- **orientation** la direction de l'immeuble : Nord, Sud, Est, Ouest
- **Glazing.area** La surface des partie en verre
- **Glazing.area.distr**
- **Energy** La performance energetique du bâtiment

- **Energy.efficiency** la variable de la performance energetique du bâtiment transformée en données qualitatives (7 classes)

Commentaires: * la variable *Glazing.area* contient des valeurs négatives. Ce qui semble être un problème, car nous avons ici une surface. Nous remplaçons toutes les valeurs négatives par 0. * On a 2 uniques valeurs de *Overall.height* et 6 uniques valeurs de *Glazing.area.distr*. Il serait peut être intéressant de catégoriser ces variables.

[4]:
```
#look the number of negative values
print("we have", len(df_energy[df_energy["Glazing.area"]<0]["Glazing.area"]),␣
 ↪"negative values")
```

```
we have 24 negative values
```

[10]:
```
#replacing negative value with 0
df_energy.loc[df_energy["Glazing.area"]<0, "Glazing.area"]=0

#categorise Overall.height variables
df_energy["Height_cat"] = 0
df_energy.loc[df_energy["Overall.height"]==3.5,"Height_cat"] = "3.5_Height"
df_energy.loc[df_energy["Overall.height"]==7.0,"Height_cat"] = "7_Height"
```

2.1 Analyse descriptive unidimentionnelle

Observations: * Le nombre de maisons avec une éfficacité énergetique = A est en moyenne deux fois supérieure aux nombre de maisons des autres catégories. * Pour chaque orientation on a le même nombre immeubles.

2.2 Analyse descriptive multidimentionnelle

Observations des analyses :

- 50% des immeubles ont un “roof area” compris entre 210 et 230. 25% entre 140 et 150.
- une moitié des immeubles a une hauteur (“overall height”) de 3.5 mètres et l’autre moitié 7 mètres.
- on a le même nombre d’immeuble pour les “Glazing area distr” = 1,2,3,4,5.
- Toutes les maisons d’éfficacité energetique E, F et G ont 3.5 mètres d’hauteur.
- Toutes les maisons d’éfficacité énergetique A ont 7 mètre d’hauteur
- On semble voir deux clusters. Le premier pour la hauteur 3.5 m et le deuxième de 7m.
- On remarque également une grande variabilité pour les hauteurs de 3.5
- un bâtiment avec une faible “Relative.compatcness” aura une petite éfficacité énergétique.
- un bâtiment avec une grande “Surface.area” aura une forte efficacité énergetique
- un bâtiment avec une grande “Roof.area” aura une forte efficacité énergetique.
- Tous les bâtiments avec *Wall.area* supérieur à 380 sont de classe des classes G, F, E, et D.

- Les immeubles avec une grande compacité relative (supérieure à 0.75) et une hauteur de 7, ont une grande éfficacité énergetique.
- Tous les immeubles de la classe A ont une hauteur de 3.5m et une compacité rélative inférieure à 0.75.
- Tous les immeubles avec une grande surface (supérieure à 660) ont une hauteur de 3.5 m et sont tous de classes A, B et C.
- Tous les immeubles avec une hauteur = 3.5 ont un Roof.area supérieur à 200 et sont tous dans les classes A, B et C
- **Existence d'une relation linéaire entre la Surface.area et le plan formé par le Wall.area et Roof.area.** Ce qui semble logique car la surface totale du bâtiment contient la surface des murs et la surface du toit. Vérifions cela avec la formule ci-dessous.

[7]:
```
#computed surface area which is the sum of wall area, roof area and ground area.
 ↪ Ground and roof have approximatively same area
df_surf_tot= 2*df_energy["Roof.area"] + df_energy["Wall.area"]

#comparison between computed surface area and Surface.area
diff_surf = df_energy["Surface.area"] - df_surf_tot


print("The maximum of the absolute difference between computed and measured␣
 ↪area is : ", np.max(diff_surf))
```

```
The maximum of the absolute difference between computed and measured area is :
1.1368683772161603e-12
```

Commentaires La vérification éffectuée ci-dessus valide les observations faites sur le graphique précédent. la Surface totale caluculée est égale à la surface mésurée *Surface.area* avec une marge d'erreur de 1e-12.

Scatter plot matrix of numerical variable

[8]:
```
fig = px.scatter_matrix(df_energy,
    dimensions=['Relative.compactness', 'Surface.area', 'Wall.area', 'Roof.
 ↪area',
       'Overall.height', 'Glazing.area','Glazing.area.distr', 'Energy'])

fig.update_layout(
                  width=1000,
                  height=1000 )

fig.show()
```

Commentaires: Relation linéaire entre les variables *Relative.compactness* et *Surface.area*. Lorsque l'une augmente l'autre dimunie. Plus la surface est grande, plus la compacité est faible.

Correlation matrix

[9]:
```
df_corr = df_energy[["Energy","Relative.compactness", "Surface.area", "Wall.
↪area",
                     "Roof.area", "Overall.height", "Glazing.area", "Glazing.
↪area.distr"]].corr()
df_corr_matrix = df_corr.round(2)

#plt.figure(figsize=(10,10))
colorscale = [[1, '#001f3f'], [0, '#3D9970']]  # custom colorscale
fig = ff.create_annotated_heatmap(z = df_corr_matrix.values, x =df_corr_matrix.
↪index.values.tolist(),
                                  y= df_corr_matrix.index.values.tolist(),␣
↪colorscale=colorscale)
fig.show()
```

Commentaires:

- **influence des variables explicatives sur la variable à expliquer:** *Relative.Compactness, Surface.area, Roof.area et Overhall.height* semblent être les plus influents sur l'éfficacité énergetique au regards de cette matrice corrélation.
- **interaction entre les variables**
- Il y a une forte liaison entre :
  - la *Relative.Compactness et Overhall.height.* Ce qui semble logique car la compacité relative d'un bâtiment est le rapport entre la surface de déperdition (l'enveloppe extérieure) et le volume protégé, qui dépend de la hauteur du bâtiment. La variable *Overhall.height* pourrait ne pas être prise en compte dans nos modélisation. Nous allons évaluer la pertinence de cette remarque.
  - *Surface.area et Roof.area.* Ce qui paraît logique car le *Roof.area* est contenu dans la surface totale *Surface.area*, qui sera retiré dans nos modèles.
- Il y a également une forte correlation négative entre:
  - *Relative.Compactness* et les variables *Surface.area et Roof.area.* Quand la première variable croit, les autres vont décroitre.
  - *Overhall.height* et les variables *Surface.area et Roof.area*

3 Analyse en composantes principales

Sur la base des résultats obtenus lors de l'analyse des variables, les variables quantitatives sont les suivantes : **Compacité relative, Surface, Surface des murs, Surface du toit, Surface du vitrage**, **Energie**. Nous conservons toutes les variables sauf **Energie**.

Réduction des variables

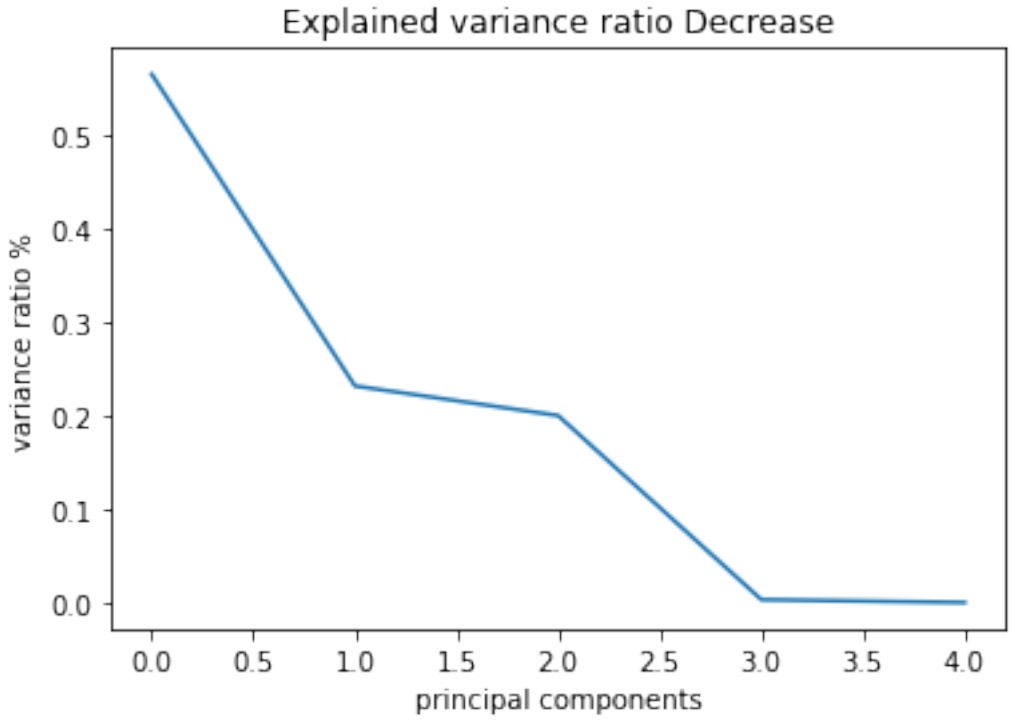
[28]:
```
X=scale(df_energy[["Relative.compactness","Surface.area","Wall.area","Roof.
↪area","Glazing.area"]])
```

Estimation, calcul des principales composantes

[29]:
```
pca = PCA()

C = pca.fit(X).transform(X)
```

Décroissance du Ratio de la variance expliqué

[30]:
```
plt.plot(pca.explained_variance_ratio_)
plt.xlabel("principal components")
plt.ylabel("variance ratio %")
plt.title("Explained variance ratio Decrease")
plt.show()
```



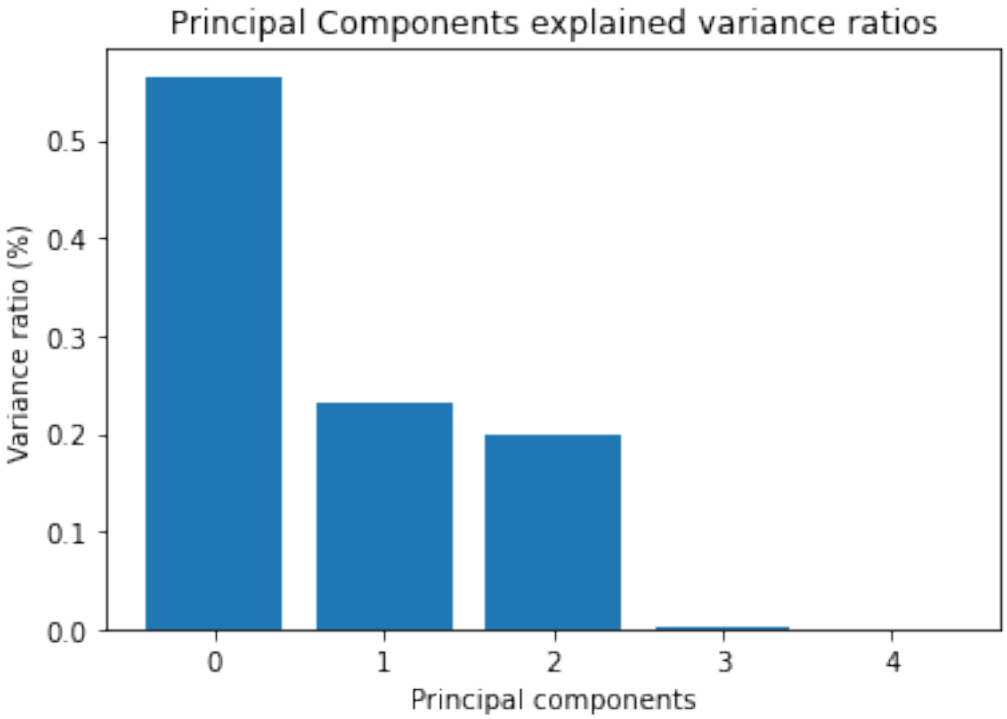
La première composante principale permet de conserver plus de 50% de la variance expliquée. Elle est donc très importante par rapport aux deux autres composantes. Les deux composantes ont des contributions comparables. Nous avons une meilleure vue en utilisant les diagrammes à barres.

[31]:
```
ratios = pca.explained_variance_ratio_
(ratios[0:3].sum()*100).round(2)
```

[31]:
```
99.7
```

[32]:
```
plt.title("Principal Components explained variance ratios")
plt.bar(range(len(ratios)), pca.explained_variance_ratio_)
plt.xticks(range(len(ratios)))
plt.xlabel("Principal components")
```

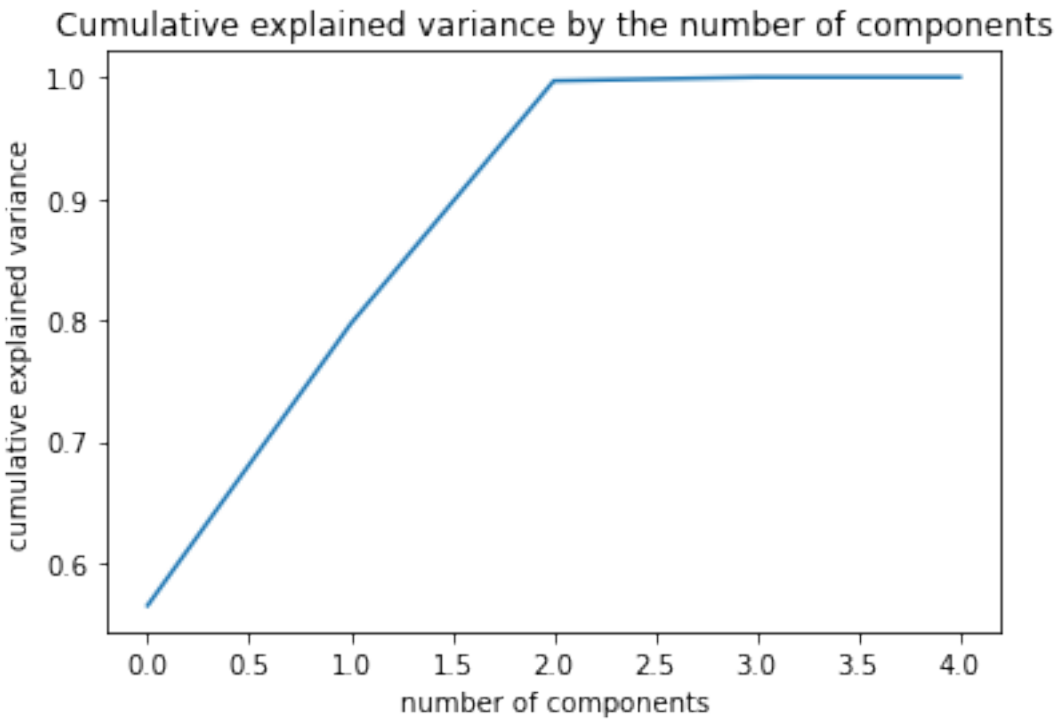
```
plt.ylabel("Variance ratio (%)")
plt.show()
```



En se basant sur la diminution soudaine de la variance expliquée au-delà de la composante numéro 2, nous avons déjà une idée du nombre de composantes à conserver. De plus, nous observons qu'en ne conservant que les 3 premières composantes principales, nous parvenons à conserver un pourcentage de variance expliquée de **99,7**.

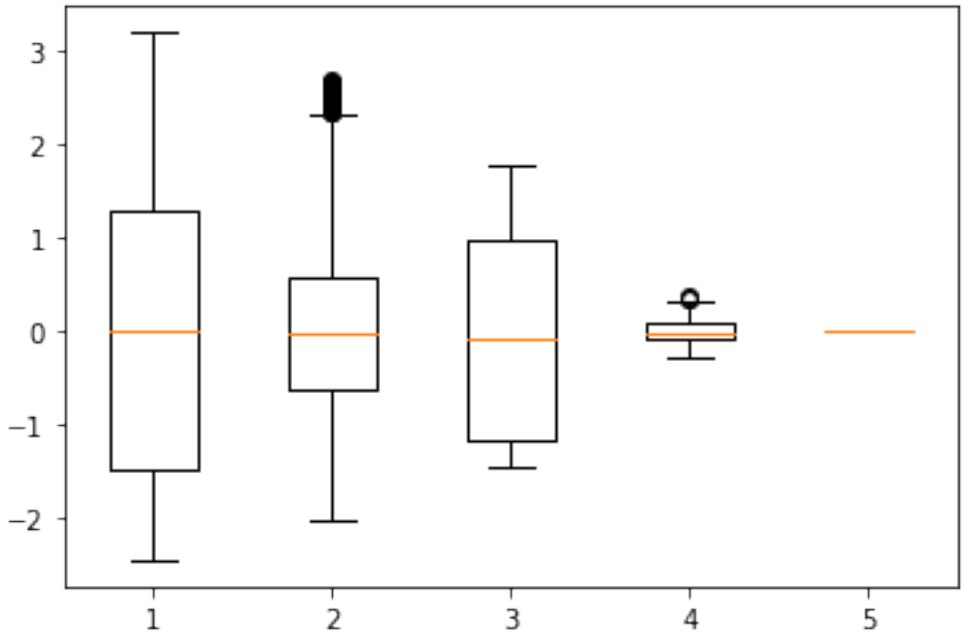
[33]:
```
plt.plot(range(len(pca.explained_variance_ratio_)),np.cumsum(pca.
 ↪explained_variance_ratio_) )
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
plt.title('Cumulative explained variance by the number of components')
```

[33]:
```
Text(0.5, 1.0, 'Cumulative explained variance by the number of components')
```



Principal components distribution

[34]:
```
plt.boxplot(C[:,0:6])
plt.show()
```



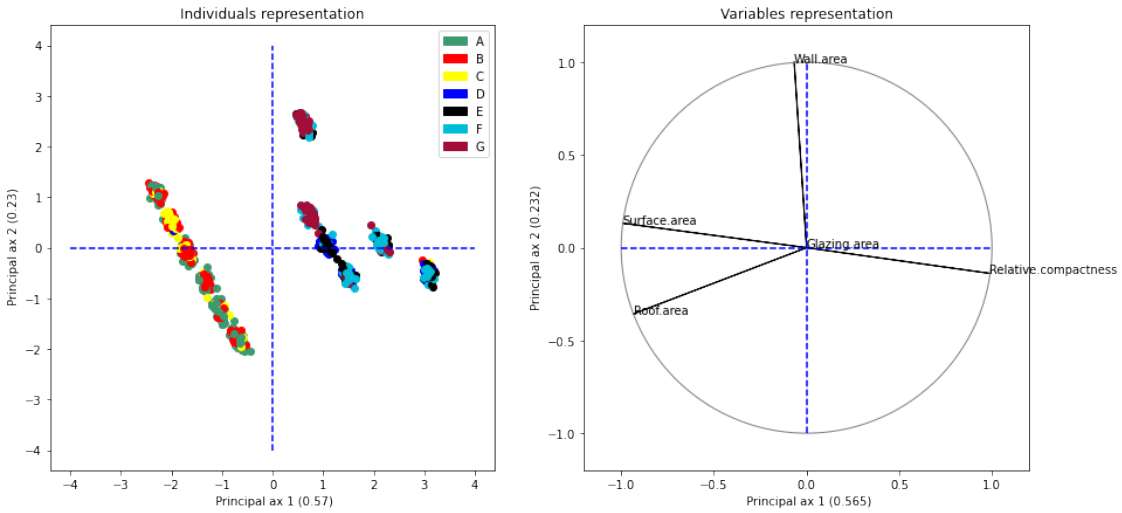
La distribution des données est symétrique pour les principales composantes importantes. En revanche, la composante principale 2 présente un grand nombre de valeurs atypiques. Il en va de même pour la composante 4 mais celle-ci ne pose pas de problème car elle sera écartée dans les analyses suivantes en raison de sa faible contribution à la variance expliquée.

3.0.1 Représentation des individus, coordonnées et représentation des variables

La représentation se fait selon les deux premiers axes principaux

[35]:
```
################################################
# Individuals representation
################################################
colorDict= {'A':"#3D9970", 'B':"#FF0000" , 'C':"#FFFF00", 'D': "#0000FF", 'E':␣
 ↪"#000000", 'F': "#00BCD8", 'G': "#A20E37"}
Legends = [mpatches.Patch(color=colorDict[clas], label=clas) for clas in␣
 ↪colorDict.keys()]

fig = plt.figure(figsize=(15,7))
ax = fig.add_subplot(1, 2, 1)

for i, j, nom in zip(C[:,0], C[:,1], df_energy["Energy.efficiency"]):
    color = colorDict[nom]
    plt.plot(i, j, "o",color=color)


plt.legend(handles=Legends)
plt.plot(np.linspace(-4,4,1000), np.zeros(1000), linestyle="dashed",␣
 ↪color="blue")
plt.plot(np.zeros(1000),np.linspace(-4,4,1000),  linestyle="dashed",␣
 ↪color="blue")
plt.xlabel(f" Principal ax 1 ({ratios[0].round(2)})")
plt.ylabel(f" Principal ax 2 ({ratios[1].round(2)})")
plt.title("Individuals representation")

################################################
# coordinates and representation of variables
################################################
coord1=pca.components_[0]*np.sqrt(pca.explained_variance_[0])
coord2=pca.components_[1]*np.sqrt(pca.explained_variance_[1])


ax = fig.add_subplot(1, 2, 2)
for i, j, nom in zip(coord1,coord2, df_energy[["Relative.compactness","Surface.
 ↪area",
```

```
                                                    "Wall.area","Roof.area","Glazing.
 ↪area"]].columns):
    plt.text(i, j, nom)
    plt.arrow(0,0,i,j,color='black')
ax.axis((-1.2,1.2,-1.2,1.2))
# cercle

c=plt.Circle((0,0), radius=1, color='gray', fill=False)
plt.xlabel(f" Principal ax 1 ({ratios[0].round(3)})")
plt.ylabel(f" Principal ax 2 ({ratios[1].round(3)})")
plt.title("Variables representation")

ax.add_patch(c)
plt.plot(np.linspace(-1,1,1000), np.zeros(1000), linestyle="dashed",␣
 ↪color="blue")
plt.plot(np.zeros(1000),np.linspace(-1,1,1000),  linestyle="dashed",␣
 ↪color="blue")
plt.show()
```



L'axe 1 isole les bâtiments en fonction de leur efficacité énergétique en deux groupes principaux : Faible efficacité énergétique (D, E, F, G) et bonne efficacité énergétique (A, B, C). La compacité relative joue un rôle énorme dans l'efficacité énergétique. Plus ce rapport augmente, plus la surface perdue à travers les murs augmente, ce qui se traduit par un faible rendement. C'est tout à fait compréhensible car plus un bâtiment est compact, plus il est performant. La compacité est mesurée par le rapport entre la surface perdue (mur, toit, etc.) et le volume à chauffer : Rapport S/V. Plus ce rapport est élevé, plus la surface perdue est importante, moins le bâtiment est à efficace.

En outre, les bâtiments ayant une bonne efficacité énergétique (A, B, C) sont généralement de petite surface.

De plus, nous pouvons voir que la surface et la compacité relative ont une très forte corrélation

négative le long du premier axe principal. Cela peut également s'expliquer par la relation indiquée ci-dessus. Sur la base de cette relation, on peut même se passer de la variable explicative surface.area.

Comme prévu, nous constatons que les variables surface.area , roof.area sont fortement corrélées.

En n'utilisant que les deux composantes principales, il est difficile de remarquer des regroupements dans la représentation des individus. On peut se faire une idée de l'importance de certaines variables telles que les variables compacité relative, surface…

La composante principale 2 définit l'importance de la surface des murs mais ne donne pas une idée directe de l'efficacité énergétique des bâtiments. Néanmoins, une valeur élevée de la surface des murs donne une idée de l'efficacité énergétique, qui est généralement faible (E,F,G).

Enfin, notons que certaines valeur aberrantes sont observées dans les données de B et C. Cela peut etre du à la distribution atypique que nous avons observée dans les boites à moustaches.

3.1 Utilisation de la technique de Clustering sur les variables quantitatives

Dans cette partie, nous utiliserons des algorithmes de clustering tels que DBSCAN, K-means. Comme nous connaissons le nombre de clusters **idéalement** attendus (A,B,…,G), ce dernier est utilisable. En fait, nous nous attendons à avoir 7 clusters. Pour rappel, les variables quantitatives mises à l'échelle sont "Compacité relative", "Surface", "Surface murale", "Surface de toit", "Surface de vitrage".

3.1.1 K-Means

k-means est un algorithme itératif qui minimise la somme des distances entre chaque individu et le centroïd.

[36]:
```
clust=KMeans(n_clusters=7, random_state=0)
clust.fit(X)
predicted_class = clust.predict(X)
```

Evaluation

[37]:
```
actual_class_list = df_energy["Energy.efficiency"].tolist()
categories = np.unique(actual_class_list).tolist()
```

a. Score aléatoire ajusté

L'indice Rand est une fonction qui calcule une mesure de similarité entre deux groupements. Pour ce calcul, l'indice aléatoire prend en compte toutes les paires d'échantillons et de comptage qui sont attribuées dans les grappes similaires ou différentes dans la mise en grappes prédite et réelle.L'étiquetage parfait serait noté 1.

[38]:
```
adjusted_rand_score(predicted_class, actual_class_list)
```

[38]: 0.25091093729904557

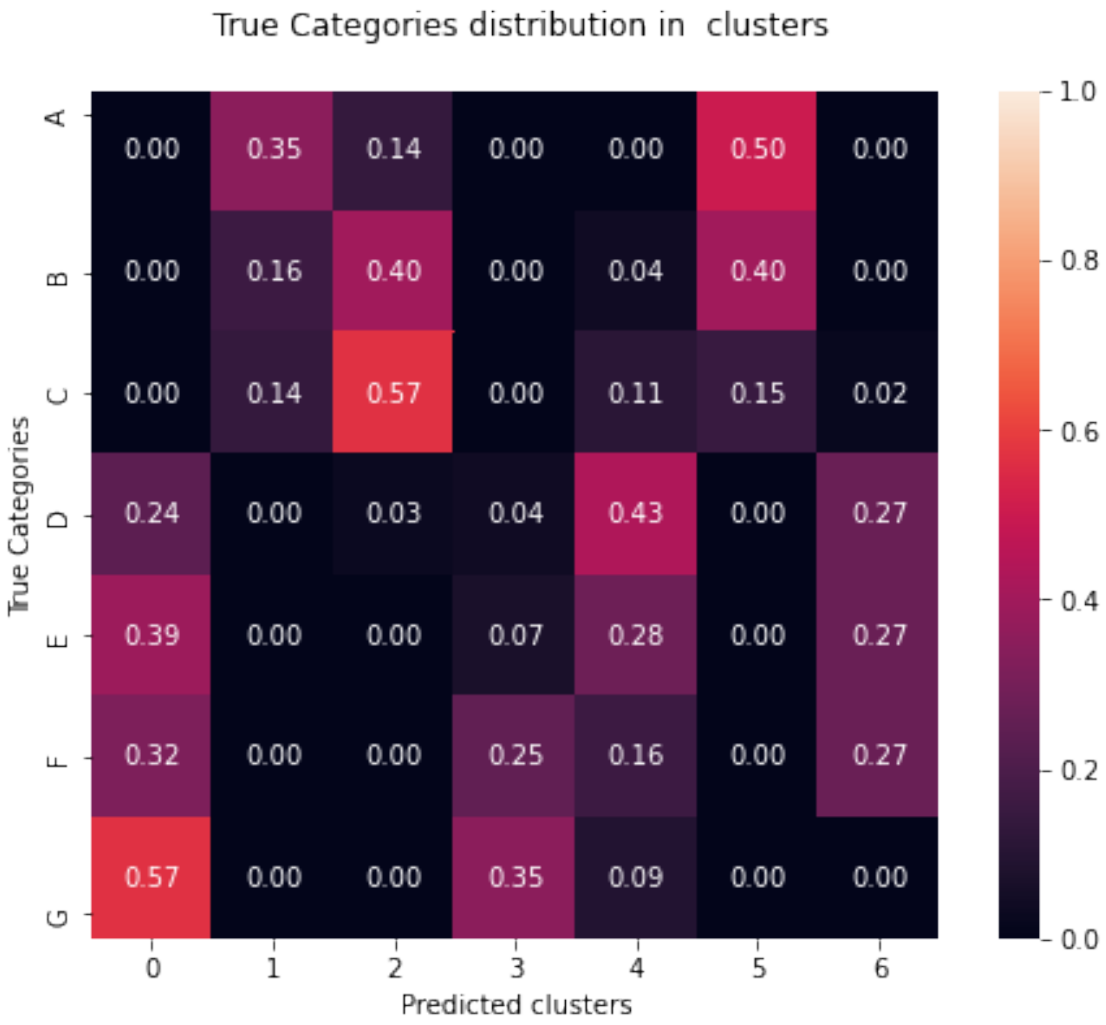
This metric gives catastrophic results. We will verify the reasons for this with the following metrics including the contingency matrix.

b. Matrice de contingence

Cette matrice indiquera la cardinalité de l'intersection pour chaque paire de confiance (vraie, prédite). La distribution des classes n'étant pas équilibrée, les comparaisons sont plus pertinentes en pourcentage.

[39]:
```
def plot_confusion_matrix(categories, actual_class_list, predicted_class):
    conting_matrix = contingency_matrix(actual_class_list,predicted_class)
    distr_rate_dict = {cat: list((distr/sum(distr)).round(2)) for cat, distr in␣
↪zip(categories, conting_matrix)}

    df_cm = pd.DataFrame(distr_rate_dict.values(),
                         index = categories,
                         columns = [str(i) for i in␣
↪range(len(set(predicted_class)))])

    fig = plt.figure(figsize=(8,6))

    plt.clf()

    ax = fig.add_subplot(111)

    ax.set_aspect(1)

    res = sns.heatmap(df_cm, annot=True, vmin=0.0, vmax=1.0, fmt='.2f')
    posit = [i+0.5 for i in range(len(categories))]
    posit[0]= 0.2
    posit[-1]= posit[-1] + 0.3
    plt.yticks(posit, categories,va='center')
    plt.xlabel("Predicted clusters")
    plt.ylabel("True Categories")
    plt.title(" True Categories distribution in  clusters\n")
```

[40]:
```
plot_confusion_matrix(categories, actual_class_list, predicted_class)
```



La figure ci-dessus nous permet de faire la correspondance entre les classes obtenues par regroupement et les classes que nous avons établies avec la variable de colonne *Energy.efficiency*. En supposant que la mise les clusters coincident, sur la base du pourcentage maximum d'éléments dans un cluster, nous définissons le nom de la cluster. Ainsi, A appartiendrait à la catégorie 0, B à la catégorie 4, C à la catégorie 6, D à la catégorie 1, E à la catégorie 5, F à la catégorie 5 et G à la catégorie 5. Seule la catégorie C a une distribution assez acceptable avec 68% de bonnes prédictions.

On remarque que les classes E, F et G se trouvent principalement dans le même groupe.

Une façon d'observer la pertinence de la division de l'efficacité énergétique en 7 classes pourrait être d'effectuer un réglage du nombre de catégories en hyperparamètres (en utilisant la méthode du Elbow) et d'observer la diminution de l'inertie. Avant de faire cette étude, on peut déjà deviner que le nombre de catégories ne sera pas de 7 puisque E, F et G se trouvent principalement ensemble et que la matrice de contingence conduit donc à des résultats catastrophiques.

Méthode Elbow Afin de choisir le nombre le plus précis de clusters, nous allons iterer sur toutes nos differentes valeur de k afin d'observer l'évolution de l'inertie. Confère le python Notebook

3.1.2 DBSCAN Clustering

voir notebook python

Conclusion Etant donné toutes les etudes menées, nous pouvons concure qu'utiliser les algorithmes de clustering pour predire connaitre exactement l'efficacité energétique des batiments n'est pas une bonné idée. Toutefois, etant donné que deux tendances se degagent a savoir (A, B,C) et (D,E,F,G), ces algorithmes permettent d'avoir une idée grossière idéee sur l'éfficacité d'un batiment. On pourrait également les utiliser pour entrainer deux modèles de classification différents pour les deux tendances.

4 MODEL

Dans un premier temps, nous transformons les données catégorielles (la variable *Orientation*) en données binaires. Etant donné que la *Overall.height* compte 2 unique valeurs, nous la binarisons également.

Nous procédons ensuite à un tirage aléatoire d'un échantillon test qui sera utilisé lors de la phase de test ou d'évaluation des modèles. La partie restante est l'échantillon d'apprentissage qui sera utilisé pour l'estimation des paramètres des modèles.

Data preprocessing

[303]:
```
# Variable explicative
EnergyDum = pd.get_dummies(df_energy[["orientation","Height_cat"]])
EnergyQuant=df_energy[["Relative.compactness","Wall.area","Roof.area","Glazing.
↪area","Glazing.area.distr"]]
df_ener=pd.concat([EnergyQuant, EnergyDum],axis=1)

df_ener.head(10)
```

[303]:

	Relative.compactness	Wall.area	Roof.area	Glazing.area	\
0	0.982928	306.484593	112.002683	0.016095	
1	0.983547	299.776324	110.048028	0.000000	
2	0.979453	303.374358	106.408431	0.000000	
3	0.977733	292.812213	113.055938	0.000010	
4	0.903029	316.236102	118.366409	0.000000	
5	0.890910	314.916242	121.843724	0.000000	
6	0.903150	320.944671	119.402232	0.000000	
7	0.898619	322.671008	122.973037	0.000000	
8	0.858579	298.629324	146.646975	0.009041	
9	0.855403	298.715776	148.776384	0.000000	

	Glazing.area.distr	orientation_East	orientation_North	orientation_South	\
0	0	0	1	0	

```
1                    0                    1                    0                    0
2                    0                    0                    0                    1
3                    0                    0                    0                    0
4                    0                    0                    1                    0
5                    0                    1                    0                    0
6                    0                    0                    0                    1
7                    0                    0                    0                    0
8                    0                    0                    1                    0
9                    0                    1                    0                    0

   orientation_West  Height_cat_3.5_Height  Height_cat_7_Height
0                 0                      0                    1
1                 0                      0                    1
2                 0                      0                    1
3                 1                      0                    1
4                 0                      0                    1
5                 0                      0                    1
6                 0                      0                    1
7                 1                      0                    1
8                 0                      0                    1
9                 0                      0                    1
```

[304]:
```
# variable à expliquer categorielle pour la classification
Ycat=df_energy["Energy.efficiency"]
# variable à expliquer réelle pour la regression
Yrel=df_energy["Energy"]
```

[305]:
```
random_state = 13

#Split the data in training and test set
X_train,X_test,Ycat_train,Ycat_test=train_test_split(df_ener,Ycat,test_size=0.
 ↪25,random_state=random_state)
X_train,X_test,Yrel_train,Yrel_test=train_test_split(df_ener,Yrel,test_size=0.
 ↪25,random_state=random_state)

classes=categories

#binarization of label for multilabel ROC curves
Ycla_test = label_binarize(Ycat_test, classes=classes)
Ycla_train = label_binarize(Ycat_train, classes=classes)
```

[306]:
```
#Normalization

scaler = StandardScaler()
scaler.fit(X_train)
Xr_train = scaler.transform(X_train)
Xr_test = scaler.transform(X_test)
```

4.1 Problème de Classification directe

Pour chaque méthode de classification, nous optimisons le modèle par validation croisée et l'évaluons avec la métrique du taux de mal classés et la matrice de confusion. Nous comparons également les scores des différentes méthodes.

4.1.1 Regression logistique sans pénalisation

[54]:
```
#optimize parameters to perform classifier
param_grid = [
    {
    'solver' : ['lbfgs','newton-cg','liblinear','sag','saga'],
    'max_iter' : [1000, 3000, 4000, 5000, 7000]
    }
]

logModel=LogisticRegression(penalty='none', random_state= random_state)

clf=GridSearchCV(logModel, param_grid,cv=5,n_jobs=-1)

logRegOpt=clf.fit(Xr_train, Ycat_train)

#  optimal parameter
print("best score = %f, best paramater = %s" % (logRegOpt.best_score_,logRegOpt.
 ↪best_params_))
```

```
best score = 0.541664, best paramater = {'max_iter': 1000, 'solver': 'lbfgs'}
```

[55]:
```
# error on test sample
logRegTestError = 1-logRegOpt.score(Xr_test, Ycat_test)
print(logRegTestError)
```

```
0.47395833333333337
```

[56]:
```
# Prediction
y_chap = logRegOpt.predict(Xr_test)
# normalized confusion matrix
table=pd.crosstab(y_chap,Ycat_test,normalize='columns')

print(table.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                  0.88  0.35  0.28  0.00  0.00  0.00  0.00
B                  0.08  0.39  0.24  0.00  0.00  0.00  0.00
C                  0.04  0.22  0.40  0.04  0.03  0.00  0.00
D                  0.00  0.04  0.08  0.43  0.41  0.17  0.07
E                  0.00  0.00  0.00  0.43  0.22  0.30  0.07
F                  0.00  0.00  0.00  0.00  0.31  0.39  0.14
G                  0.00  0.00  0.00  0.09  0.03  0.13  0.71
```

Commentaires : 88% des immeubles de la **classe A** et 71% de la **classe G** ont été correctement prédits. Les autres catégories ont été correctement prédites avec un pourcentage inférieure à 50%. Malgré la dominance de la classe A sur la classe G, le modèle a pu prédire correctement 71% des immeubles de la classe G.

Malgré ce résultat nous allons éssayer de voir si un réquilibrage de l'échantillon améliore la qualité de la classification. Nous allons utiliser le *Condensed Nearest Neighbor (CNN) Rule Undersampling* qui est une technique de sous-échantillonnage qui recherche un sous-ensemble d'une collection d'échantillons qui n'entraîne aucune perte de performances du modèle, appelée ensemble cohérent minimal. (Voir notebook python)

Commentaires : Le résultat montre une augmentation du pourcentages des mals classés des classes A, B, D et G, et une dimunition des mals classés des classes C, E et F. Malgré cet amélioration de bien classés sur ces trois classes, nous poursuivrons avec les échantillons initiaux, qui nous donnent une taux d'erreur de test inférieur à celle avec le *CondensedNearestNeighbour*

4.1.2 Regression logistique avec pénalisation

Regression logistique avec une pénalisation de Lasso Dans une regression logistique multinomiale, 'saga' est le seul solver qui supporte la penalisation de Lasso.

[60]:
```
#Optimize parameter of penalization
param_grid = [
    {
    'C' : np.logspace(-4, 4, 10),
    'max_iter' : [3000, 4000, 5000, 7000]
    }
]

logRegModel=LogisticRegression(penalty='l1', solver='saga',␣
 ↪random_state=random_state)

clf=GridSearchCV(logRegModel, param_grid,cv=5,n_jobs=-1)

logLassOpt=clf.fit(Xr_train, Ycat_train)

#optimal parameter
print("best score = %f, best parameters = %s" % (logLassOpt.
 ↪best_score_,logLassOpt.best_params_))
```

```
best score = 0.543433, best parameters = {'C': 21.54434690031882, 'max_iter':
3000}
```

[61]:
```
# error on test sample
lassoTestError = 1-logLassOpt.score(Xr_test, Ycat_test)
print(lassoTestError)
```

```
0.47395833333333337
```

[62]:
```
# Prediction
y_chap = logLassOpt.predict(Xr_test)

# normalized confusion matrix
tableLasso =pd.crosstab(y_chap,Ycat_test,normalize='columns')
print(tableLasso.round(2))
```

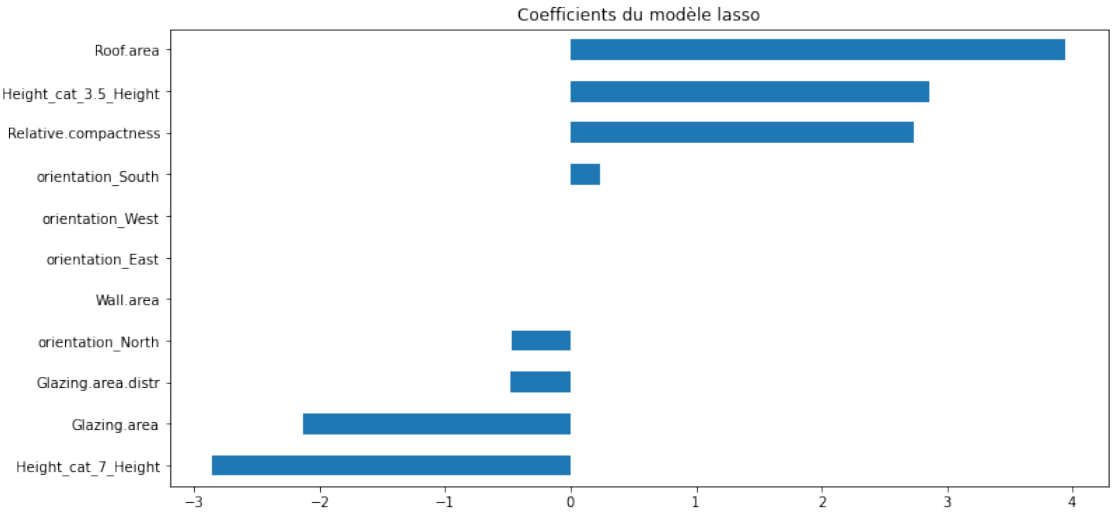
Energy.efficiency	A	B	C	D	E	F	G
row_0							
A	0.88	0.35	0.28	0.00	0.00	0.00	0.00
B	0.08	0.39	0.20	0.00	0.00	0.00	0.00
C	0.04	0.22	0.40	0.04	0.03	0.00	0.00
D	0.00	0.04	0.12	0.48	0.41	0.13	0.07
E	0.00	0.00	0.00	0.39	0.22	0.39	0.07
F	0.00	0.00	0.00	0.00	0.31	0.35	0.14
G	0.00	0.00	0.00	0.09	0.03	0.13	0.71

Commentaires : Lasso dimunie légèrement le taux des mals classés de la classe D et tout en augmentant le taux des mals classés de la classe F, qui malgré tout restent inférieur à 0.5. Le taux de mal classés global reste approximativement le même avec celui sans pénalité.

Recherche des coefficients de Lasso (voir pyhton notebook)

[66]:
```
imp_coef = coef.sort_values()
plt.rcParams['figure.figsize'] = (12.0, 6.0)
imp_coef.plot(kind = "barh")
plt.title(u"Coefficients du modèle lasso")
```

[66]:
```
Text(0.5, 1.0, 'Coefficients du modèle lasso')
```



Interpretation de l'effet des variables retenues : Les variables *Roof.area*, *Overhall.height*, *Relative.compactness et Glazing.area* ont une très grande importance dans la classification de la

catégorie B.

Regression logistique avec une pénalisation de Ridge

[67]:
```
#Optimize parameter of penalization
param_grid = [
    {
    'solver' : ['lbfgs','newton-cg','sag','saga'],
    'C' : np.logspace(-4, 4, 10),
    'max_iter' : [3000, 4000, 5000, 7000]
    }
]

logModel=LogisticRegression(penalty='l2', random_state = random_state)

clf=GridSearchCV(logModel, param_grid,cv=5,n_jobs=-1)

logitRidgeOpt=clf.fit(Xr_train, Ycat_train)

#  optimal parameter
print("best score = %f, best parameter = %s" % (logitRidgeOpt.
 ↪best_score_,logitRidgeOpt.best_params_))
```

```
best score = 0.545142, best parameter = {'C': 21.54434690031882, 'max_iter':
3000, 'solver': 'lbfgs'}
```

[68]:
```
# error on test sample
ridgeTestError = 1-logitRidgeOpt.score(Xr_test, Ycat_test)
print(ridgeTestError)
```

```
0.47395833333333337
```

[69]:
```
# Prediction
y_chap = logitRidgeOpt.predict(Xr_test)

# normalized confusion matrix
table=pd.crosstab(y_chap,Ycat_test,normalize='columns')
print(table.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                  0.88  0.39  0.24  0.00  0.00  0.00  0.00
B                  0.08  0.30  0.24  0.00  0.00  0.00  0.00
C                  0.04  0.26  0.40  0.04  0.03  0.00  0.00
D                  0.00  0.04  0.12  0.52  0.38  0.13  0.07
E                  0.00  0.00  0.00  0.35  0.25  0.39  0.07
F                  0.00  0.00  0.00  0.00  0.31  0.35  0.14
G                  0.00  0.00  0.00  0.09  0.03  0.13  0.71
```

Commentaires : On obtient le même résultat que celui obtenu avec Lasso. Ridge n'améliore pas

significativement la classsification.

4.1.3 Optimal decision tree

[70]:
```
# Optimize the depth of tree
param=[{"max_depth":[None, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 50],
        'criterion': ['gini', 'entropy']
       }]

tree= GridSearchCV(DecisionTreeClassifier(random_state = random_state),
                   param,cv=10,n_jobs=-1)
treeOpt=tree.fit(Xr_train, Ycat_train)

# paramètre optimal
print("best score = %f, best parameter = %s" % (treeOpt.best_score_,treeOpt.
 ↪best_params_))
```

```
best score = 0.617967, best parameter = {'criterion': 'entropy', 'max_depth': 5}
```

[71]:
```
# error on test sample
optTestError = 1.-treeOpt.score(Xr_test, Ycat_test)
print(optTestError)
```

```
0.34375
```

[72]:
```
# Prediction
y_chap = treeOpt.predict(Xr_test)

# normalized confusion matrix
tableTree=pd.crosstab(y_chap,Ycat_test,normalize='columns')
print(tableTree.round(2))
```

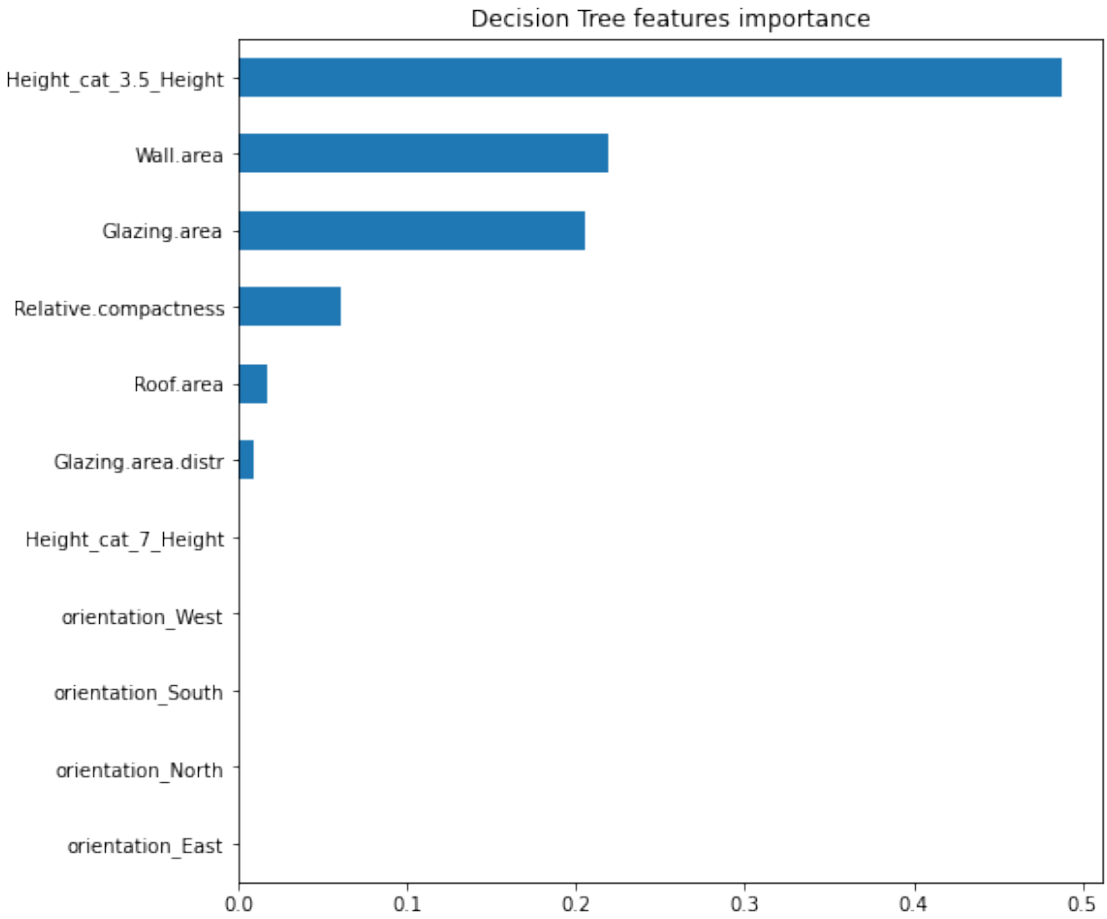
```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                  0.96  0.39  0.08  0.00  0.00  0.00  0.00
B                  0.04  0.57  0.40  0.00  0.00  0.00  0.00
C                  0.00  0.00  0.40  0.04  0.00  0.00  0.00
D                  0.00  0.00  0.04  0.35  0.03  0.00  0.00
E                  0.00  0.04  0.08  0.61  0.66  0.35  0.00
F                  0.00  0.00  0.00  0.00  0.31  0.61  0.29
G                  0.00  0.00  0.00  0.00  0.00  0.04  0.71
```

Commentaires: L'arbre de decision améliore la classification. Comparé à la regression logistique, on a ici un une augmentaion du taux des biens classés des classes A, B, E et F. Il n y a plus que les classes C et D qui ont des taux de bien classés inférieurs à 0.6. Avec l'ajout de la non-linéarité, le taux de mal classé est passé de 0.47 à 0.34. Ce qui améliore considérablement la classification

[76]:
```
#plot features importance
featImport = pd.Series(treeClf.feature_importances_, index = X_train.columns)
```

```
imp_featImport = featImport.sort_values()
plt.rcParams['figure.figsize'] = (8.0, 8.0)
imp_featImport.plot(kind = "barh")
plt.title(u"Decision Tree features importance")
```

[76]: `Text(0.5, 1.0, 'Decision Tree features importance')`



Commentaires : Avec le modèle de l'arbre de decision,toutes les variables *orientations* sont supprimés, tandis que les autres variables ont plus ou moins une grande importance dans la classification des immeubles. Remarquons néanmoins que la variable *hauteur=3.5m" a une grande importance dans le modèle.

4.1.4 Random Forest Classification

[78]:
```
#Optimize parameter of penalization
param_grid = [
    {
    'n_estimators' : [10, 20, 30],
    'max_depth' : [None, 1, 2, 4, 8, 16, 32],
```

```
    'max_features' : ['auto', 1.0, 0.3, 0.1],
    'min_samples_leaf': [1, 3, 5],
    'bootstrap': [True, False],
    'criterion': ['gini', 'entropy']
    }
]

RFModel=RandomForestClassifier(random_state = random_state)

clf=GridSearchCV(RFModel, param_grid,cv=5,n_jobs=-1)

RFOpt=clf.fit(Xr_train, Ycat_train)
```

[79]:
```
#  optimal parameter
print("best score = %f, best parameter = %s" % (RFOpt.best_score_,RFOpt.
 ↪best_params_))
```

```
best score = 0.638876, best parameter = {'bootstrap': False, 'criterion':
'entropy', 'max_depth': None, 'max_features': 'auto', 'min_samples_leaf': 5,
'n_estimators': 20}
```

[80]:
```
# error on test sample
RFTestError = 1-RFOpt.score(Xr_test, Ycat_test)
print(RFTestError)
```

```
0.38020833333333337
```

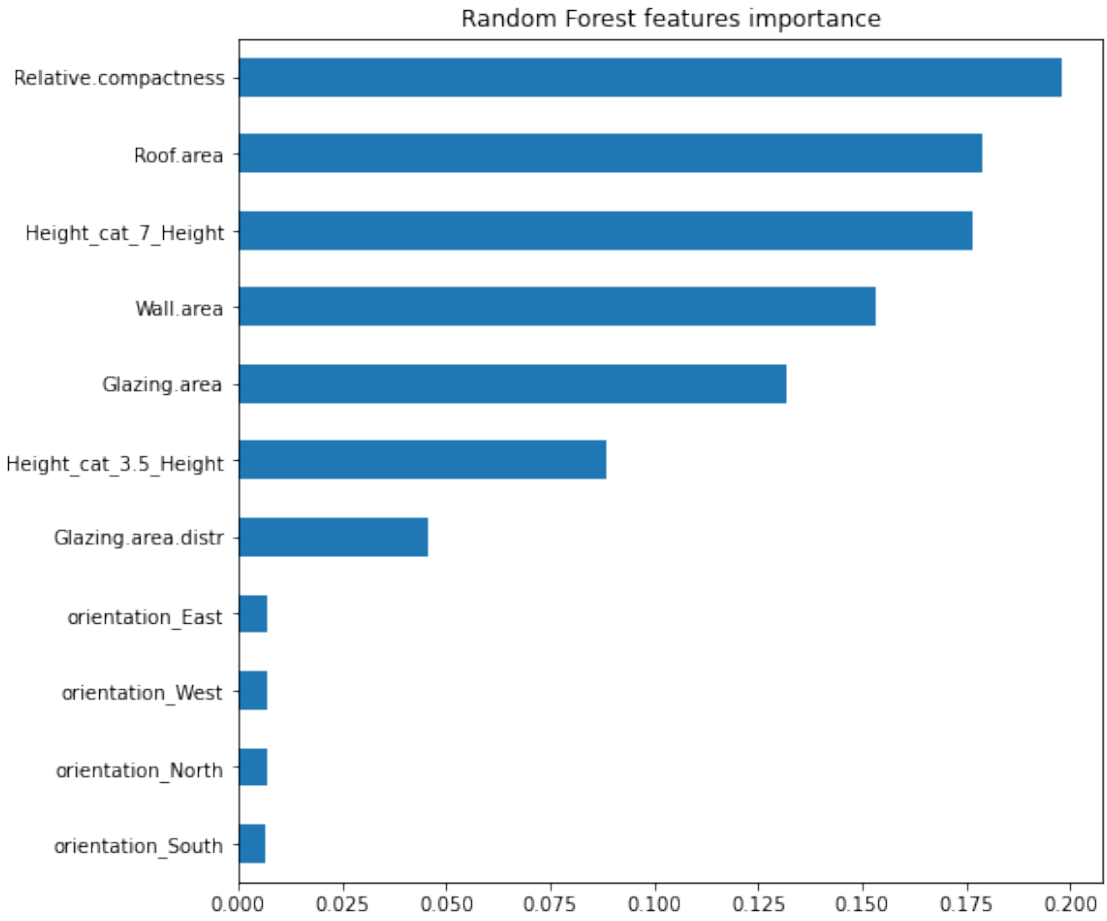
[81]:
```
# Prediction
y_chap = RFOpt.predict(Xr_test)

# normalized confusion matrix
tableRF=pd.crosstab(y_chap,Ycat_test,normalize='columns')
print(tableRF.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                  0.88  0.30  0.20  0.00  0.00  0.00  0.00
B                  0.08  0.43  0.16  0.00  0.00  0.00  0.00
C                  0.04  0.26  0.52  0.09  0.00  0.00  0.00
D                  0.00  0.00  0.08  0.57  0.22  0.00  0.00
E                  0.00  0.00  0.04  0.35  0.47  0.35  0.00
F                  0.00  0.00  0.00  0.00  0.31  0.57  0.36
G                  0.00  0.00  0.00  0.00  0.00  0.09  0.64
```

Commentaires: Comparé au *Decison tree*, le *Random Forest* dimunie le taux des mals classés des classes A B, E, F et G, mais améliore également le taux des bien classés des classes C et D qui sont au dessus de 50%. Le taux de mal classés global de RF (0,38) est supérieure à celui de *Decison Tree*, mais rest meilleure que celui de la regression logistique.

[85]:
```
#plot features importance
featImport = pd.Series(RFClf.feature_importances_, index = X_train.columns)
imp_featImport = featImport.sort_values()
plt.rcParams['figure.figsize'] = (8.0, 8.0)
imp_featImport.plot(kind = "barh")
plt.title(u"Random Forest features importance")
```

[85]:
```
Text(0.5, 1.0, 'Random Forest features importance')
```



Commentaires : les variables *Relative.compatcness, Roof.area, Height=7m, Wall.area, Glazing.area, et Height=3.5m* ont une grande importance comparées à celle des orientations. Elles influencent grandément notre classification

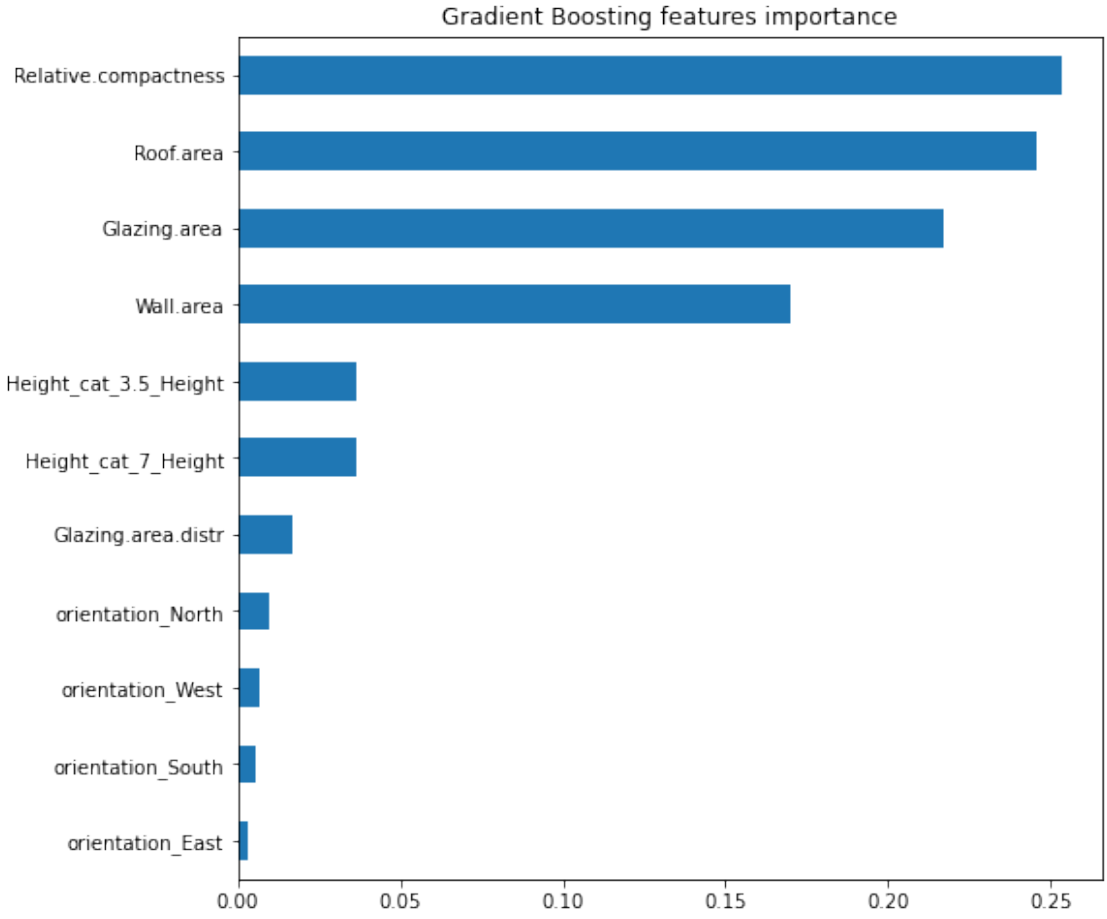
4.1.5 Boosting Classification

GradientBoostingClassifier (Voir python notebook) Commentaires: Comparé au *Random Forest*, le *Gradient Boosting* dimunie le taux des mals classés des classes C, D, E, et F, mais améliore le taux des bien classés des classes B et G qui sont au dessus de 60%. Le taux de mal classés global (0,39) est légèrement supérieure à celui de *Random Forest*, mais reste meilleure que

celui de la regression logistique. Ceci dû aux effet de non-linarité.

[93]:
```
#plot features importance
featImport = pd.Series(GBClf .feature_importances_, index = X_train.columns)
imp_featImport = featImport.sort_values()
plt.rcParams['figure.figsize'] = (8.0, 8.0)
imp_featImport.plot(kind = "barh")
plt.title(u"Gradient Boosting features importance")
```

[93]:
```
Text(0.5, 1.0, 'Gradient Boosting features importance')
```



Commentaires : les variables *Relative.compatcness, Roof.area, Glazing.area et Wall.area* ont une grande importance comparées aux autres variables. Elles influencent significativement notre modèle.

AdaBoostingClassifier (voir python notebook)

[95]:
```
#  optimal parameter
```

```
print("best score = %f, best parameter = %s" % (AdaBoosOpt.
 ↪best_score_,AdaBoosOpt.best_params_))
```

```
best score = 0.630150, best parameter = {'algorithm': 'SAMME', 'base_estimator':
DecisionTreeClassifier(max_depth=4), 'learning_rate': 0.01, 'n_estimators': 200}
```

[96]:
```
# error on test sample
AdBoosTestError = 1-AdaBoosOpt.score(Xr_test, Ycat_test)
print(AdBoosTestError)
```

```
0.38541666666666663
```

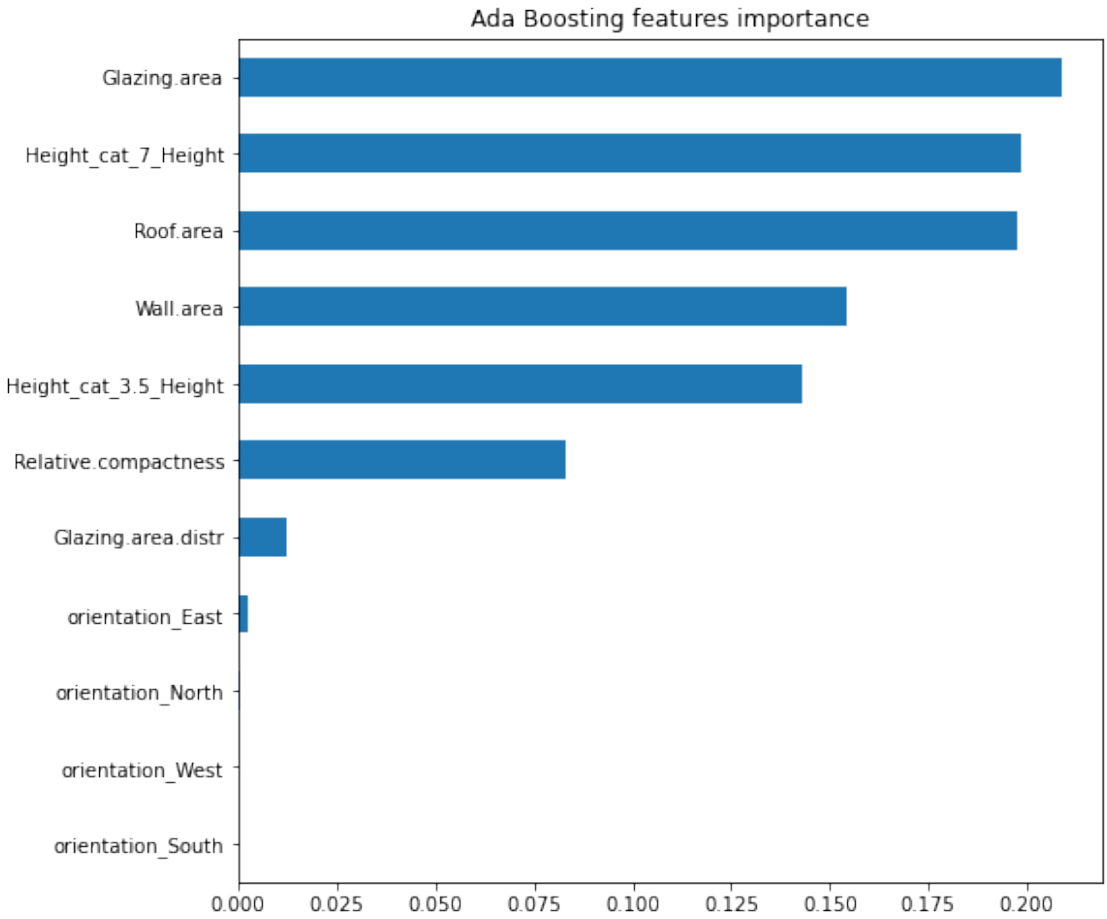
[97]:
```
# Prediction
y_chap = AdaBoosOpt.predict(Xr_test)

# normalized confusion matrix
tableAB=pd.crosstab(y_chap,Ycat_test,normalize='columns')
print(tableAB.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                  0.88  0.30  0.12  0.00  0.00  0.00  0.00
B                  0.12  0.48  0.32  0.00  0.00  0.00  0.00
C                  0.00  0.17  0.48  0.09  0.00  0.00  0.00
D                  0.00  0.00  0.00  0.35  0.16  0.00  0.00
E                  0.00  0.04  0.08  0.57  0.53  0.35  0.00
F                  0.00  0.00  0.00  0.00  0.31  0.65  0.36
G                  0.00  0.00  0.00  0.00  0.00  0.00  0.64
```

Commentaires: Comparé au *Gradient Boosting*, le *Ada Boosting* augmente le taux des mals classés des classes B, D et G, et améliore le taux des bien classés des classes F, E et C. Le taux de mal classés global (0,385) est légèrement supérieure à celui de *Random Forest*, mais est légèrement meilleure que celui du gradient bossting.

[101]:
```
#plot features importance
featImport = pd.Series(ABClf .feature_importances_, index = X_train.columns)
imp_featImport = featImport.sort_values()
plt.rcParams['figure.figsize'] = (8.0, 8.0)
imp_featImport.plot(kind = "barh")
plt.title(u"Ada Boosting features importance")
```

[101]:
```
Text(0.5, 1.0, 'Ada Boosting features importance')
```



Commentaires Toutes les variables orientations n'ont en effet pas une influence sur notre modèle de classification.

4.1.6 Support Vector Machine

Support Vector Machine for linear kernel (voir python notebook)

[104]:
```
# error on test sample
SVCLinTestError = 1-SVCOpt.score(Xr_test, Ycat_test)
print(SVCLinTestError)
```

```
0.484375
```

[105]:
```
# Prediction
y_chap = SVCOpt.predict(Xr_test)

# normalized confusion matrix
tableSVCLin=pd.crosstab(y_chap,Ycat_test,normalize='columns')
print(tableSVCLin.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
```

```
row_0
A                   0.88  0.35  0.24  0.00  0.00  0.00  0.00
B                   0.10  0.39  0.24  0.00  0.00  0.00  0.00
C                   0.02  0.22  0.36  0.00  0.00  0.00  0.00
D                   0.00  0.04  0.16  0.57  0.50  0.17  0.21
E                   0.00  0.00  0.00  0.35  0.16  0.39  0.07
F                   0.00  0.00  0.00  0.00  0.31  0.35  0.07
G                   0.00  0.00  0.00  0.09  0.03  0.09  0.64
```

Commentaires Le taux global de mal classés pour le SVM avec un kernel linéaire est supérieur aux taux de tous les modèles étudiés plus haut.

Support Vector Machine for polynomial kernel (voir python notebook)

[110]:
```
# error on test sample
SVCPolyTestError = 1-SVCPolyOpt.score(Xr_test, Ycat_test)
print(SVCPolyTestError)
```

```
0.46354166666666663
```

[111]:
```
# Prediction
y_chap = SVCPolyOpt.predict(Xr_test)

# normalized confusion matrix
tableSVCPoly=pd.crosstab(y_chap,Ycat_test,normalize='columns')
print(tableSVCPoly.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                   0.79  0.43  0.20  0.00  0.00  0.00  0.00
B                   0.15  0.30  0.36  0.00  0.00  0.00  0.00
C                   0.06  0.22  0.36  0.04  0.03  0.00  0.00
D                   0.00  0.04  0.08  0.52  0.25  0.09  0.07
E                   0.00  0.00  0.00  0.26  0.41  0.39  0.00
F                   0.00  0.00  0.00  0.09  0.31  0.43  0.14
G                   0.00  0.00  0.00  0.09  0.00  0.09  0.79
```

Commentaires Le kernel Polynomial améliore légèrement le taux de mal classés comparé à celui du kernel linéaire.

Support Vector Machine for RBF kernel

[114]:
```
#Optimize parameter

param_grid = [
    {
    'C' : [1, 1e1, 1e2, 1e3, 1e4],
    'tol' : [1e-3, 1e-4],
    'gamma' : [1e0, 1e-1, 1e-2, 1e-3]
    }
]
```

```
SvcRbfModel=SVC(kernel = 'rbf', random_state = random_state)

clf=GridSearchCV(SVCPolyModel, param_grid, cv=5, n_jobs=-1)

SvcRbfOpt=clf.fit(Xr_train, Ycat_train)
```

[115]:
```
#  optimal parameter
print("best score = %f, best parameter = %s" % (SvcRbfOpt.best_score_,SvcRbfOpt.
 ↪best_params_))
```

```
best score = 0.574558, best parameter = {'C': 10.0, 'gamma': 0.1, 'tol': 0.001}
```

[116]:
```
# error on test sample
SVCRbfTestError = 1-SvcRbfOpt.score(Xr_test, Ycat_test)
print(SVCRbfTestError)
```

```
0.46354166666666663
```

[117]:
```
# Prediction
y_chap = SvcRbfOpt.predict(Xr_test)

# normalized confusion matrix
tableSVCRBF=pd.crosstab(y_chap,Ycat_test,normalize='columns')
print(tableSVCRBF.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                  0.79  0.43  0.20  0.00  0.00  0.00  0.00
B                  0.15  0.30  0.36  0.00  0.00  0.00  0.00
C                  0.06  0.22  0.36  0.04  0.03  0.00  0.00
D                  0.00  0.04  0.08  0.52  0.25  0.09  0.07
E                  0.00  0.00  0.00  0.26  0.41  0.39  0.00
F                  0.00  0.00  0.00  0.09  0.31  0.43  0.14
G                  0.00  0.00  0.00  0.09  0.00  0.09  0.79
```

Commentaires Le kernel radial n'améliore pas le taux de mals classés obtenu avec le kernel polynomial.

4.1.7 Comparaison des performances

Dans cette partie, nous comparons le taux des mal classés obtenu avec l'échantillon de test des différents modèles. Nous comparons les modèles de regression logistique lasso, optimal tree, Random Forest, Adaboosting à l'aide d'une *validation croisée Monte Carlo* étant donné que la taille de notre dataset est 780.

[121]:
```
from sklearn.utils import check_random_state
import time

tps0=time.perf_counter()
```

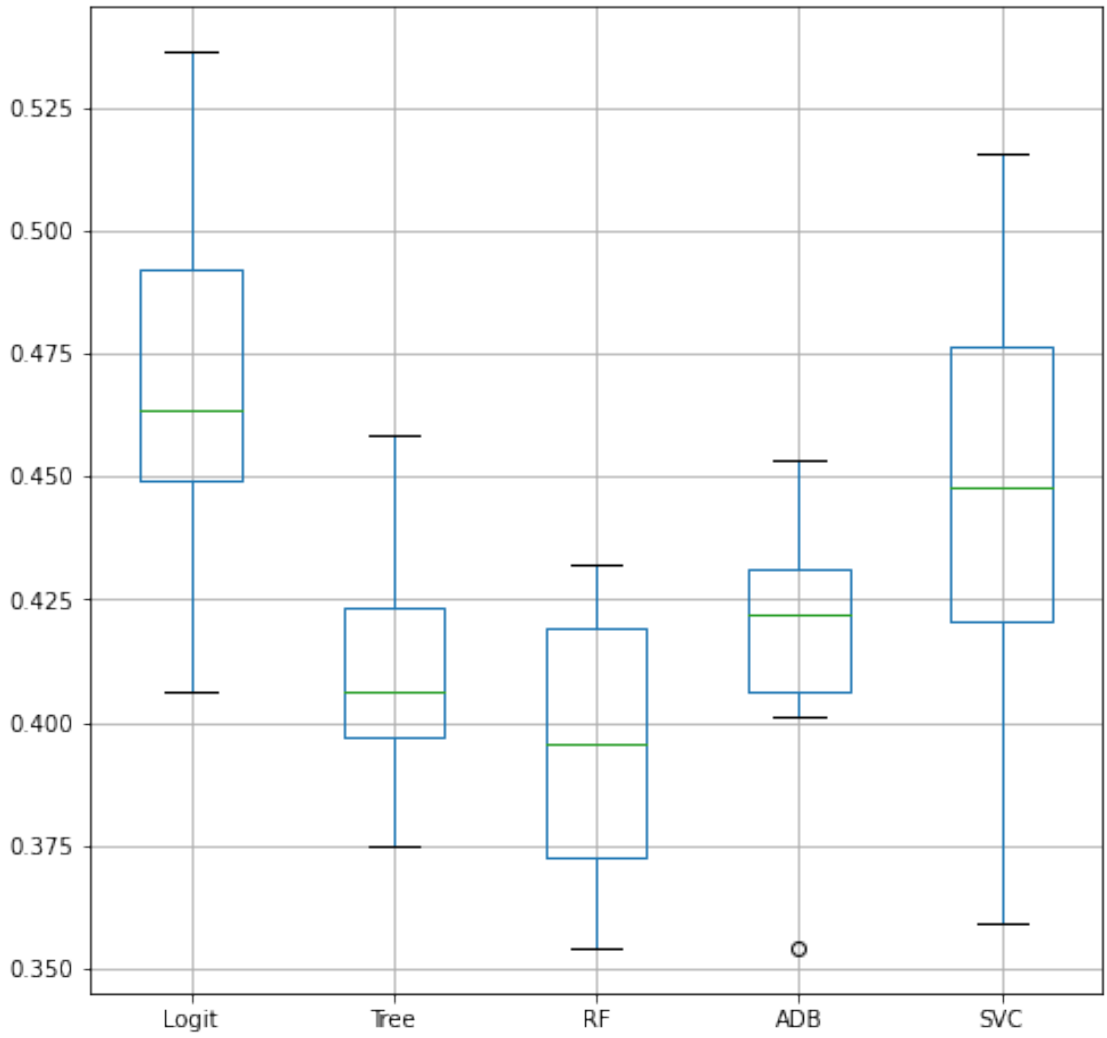
```
check_random_state(13)

# estimators
logit= LogisticRegression(penalty="l1",solver="saga")
tree = DecisionTreeClassifier()
rf   = RandomForestClassifier()
adb  = AdaBoostClassifier()
svc = SVC(kernel = 'rbf')

# Nombre d'itérations
B=10

# définition des grilles de paramètres
listMethGrid=[

    [logit,{
        'C' : np.logspace(-4, 4, 10),
        'max_iter' : [3000, 4000, 5000, 7000]
    }],

    [tree,{
        "max_depth":[None, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 50],
        'criterion': ['gini', 'entropy']
       }],

    [rf,{
        'n_estimators' : [10, 20, 30],
        'max_depth' : [None, 1, 2, 4, 8, 16, 32],
        'max_features' : ['auto', 1.0, 0.3, 0.1],
        'min_samples_leaf': [1, 3, 5],
        'bootstrap': [True, False],
        'criterion': ['gini', 'entropy']
    }],

    [adb,{
        'base_estimator' :[DecisionTreeClassifier(max_depth=3),␣
↪DecisionTreeClassifier(max_depth=4),
                        DecisionTreeClassifier(max_depth=8),␣
↪DecisionTreeClassifier(max_depth=16)],
        'learning_rate': [1e-1, 1e-2, 1e-3],
        'n_estimators' : [150,200,300],
        'algorithm': ['SAMME', 'SAMME.R']
    }],


    [svc, {
        'C' : [1, 1e1, 1e2, 1e3, 1e4],
```

```
        'tol' : [1e-3, 1e-4],
        'gamma' : [1e0, 1e-1, 1e-2, 1e-3]
    }]

]



# Initialisation à 0
arrayErreur=np.empty((B,5))

for i in range(B):   # itérations sur B échantillons test

    # extraction apprentissage et test
    X_train,X_test,Y_train,Y_test=train_test_split(df_ener,Ycat,test_size=0.25)
    scaler = StandardScaler()
    scaler.fit(X_train.astype(float))
    Xnet_train = scaler.transform(X_train.astype(float))

    # Meme transformation sur le test
    Xnet_test = scaler.transform(X_test.astype(float))

    # optimisation de chaque méthode et calcul de l'erreur sur le test
    for j,(method, grid_list) in enumerate(listMethGrid):
        methodGrid=GridSearchCV(method,grid_list,cv=10,n_jobs=-1).
 ↪fit(Xnet_train, Y_train)
        methodOpt = methodGrid.best_estimator_
        methFit=methodOpt.fit(Xnet_train, Y_train)
        arrayErreur[i,j]=1-methFit.score(Xnet_test,Y_test)
tps1=time.perf_counter()
```

[122]:
```
dataframeErreur=pd.DataFrame(arrayErreur,columns=["Logit","Tree","RF","ADB",␣
 ↪"SVC"])
print("Temps execution :",(tps1 - tps0))
```

```
Temps execution : 10584.771345599998
```

[123]:
```
dataframeErreur[["Logit","Tree","RF","ADB", "SVC"]].boxplot(return_type='dict')
plt.show()
```



[124]:
```
dataframeErreur.mean()
```

[124]:
```
Logit    0.469271
Tree     0.410938
RF       0.395313
ADB      0.416667
SVC      0.446354
dtype: float64
```

Commentaires : Avec une validation croisé de Monte-carlo sur une itération de **B=10, "Radom Forest"** se révèle être le meilleur modèle de classification directe pour la prédiction des classes d'énergie. Le taux de mal classé moyen sur ces 10 itérations est égal à 0.39. EN terme de variation, le SVC avec un kernel radial, varie beaucoup. Il est assez volatile et sensible.

4.2 Problème de classification indirecte

4.2.1 Prevision par Regression linéaire ou modèle Gaussien

Regression Linéaire sans pénalisation

[127]:
```
lm = LinearRegression()
lm
```

[127]:
```
LinearRegression()
```

[129]:
```
lm.fit(Xr_train,
       Yrel_train)
```

[129]:
```
LinearRegression()
```

[130]:
```
Y_test = lm.predict(Xr_test)
```

[131]:
```
lm.score(Xr_train,Yrel_train)
```

[131]:
```
0.8759844279200635
```

[132]:
```
lm.score(Xr_test,Yrel_test)
```

[132]:
```
0.876359821813611
```

[133]:
```
lm.intercept_, lm.coef_
```

[133]:
```
(47.22043342360051,
 array([-8.27493447, -0.23253181, -8.8171094 ,  4.85459163,  0.09168762,
        -0.42491197,  0.12527706,  0.22386987,  0.07528743, -8.00233881,
         8.00233881]))
```

[134]:
```
print("MSE=", mean_squared_error(Yrel_test, Y_test))
```

```
MSE= 42.4338924859722
```

[135]:
```
Y_hat = lm.predict(Xr_train)
```

[136]:
```
# Erreur quadratique moyenne

from sklearn.metrics import mean_squared_error

print("MSE=", mean_squared_error(Yrel_train, Y_hat))
```

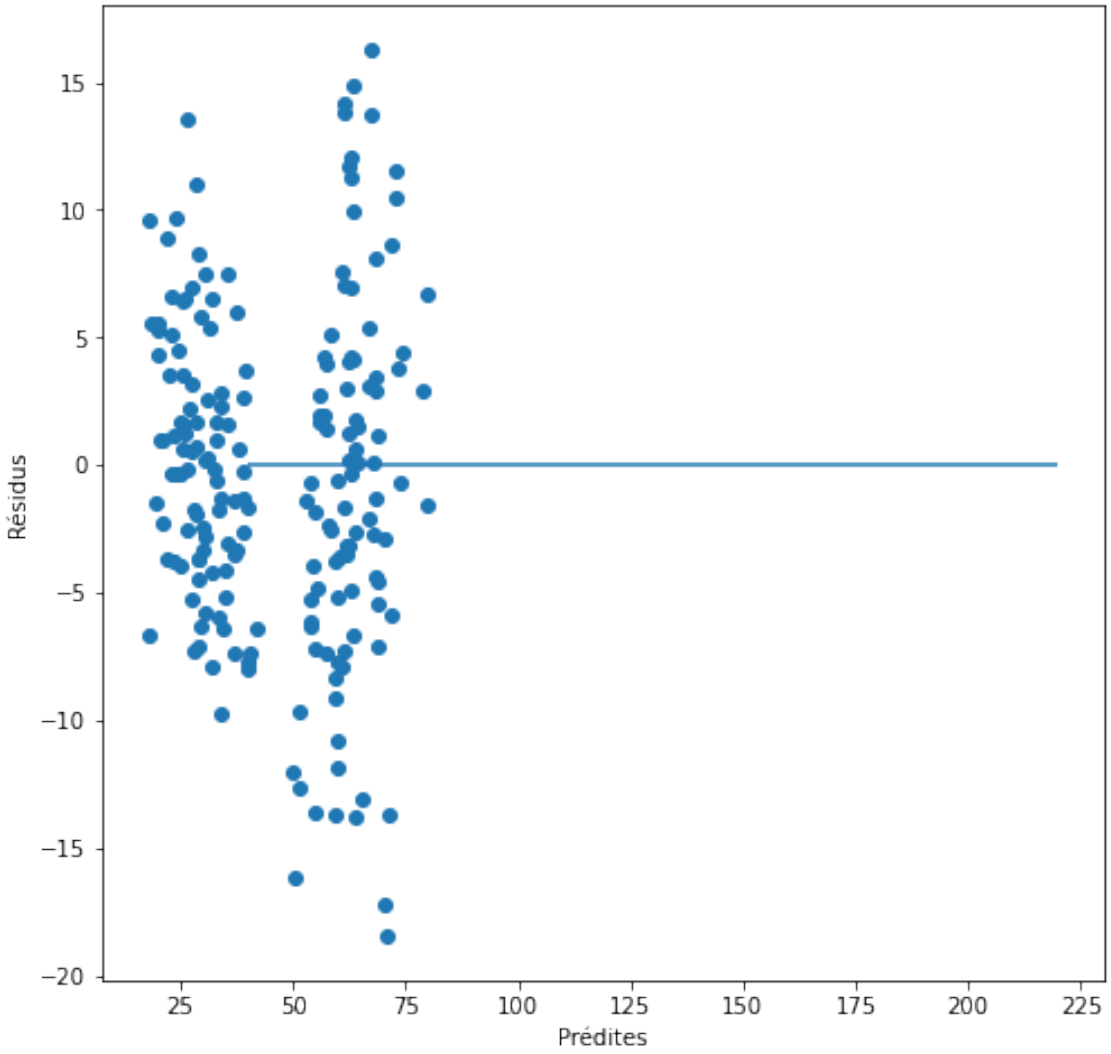
```
MSE= 50.92238547972625
```

Interpretation:

Nous constatons que ce modèle generalise bien sur les données inconnues

Graphe des Résidus

[137]:
```
plt.plot(Y_test,Yrel_test-Y_test,"o")
plt.xlabel(u"Prédites")
plt.ylabel(u"Résidus")
plt.hlines(0,40,220)
plt.show()
```

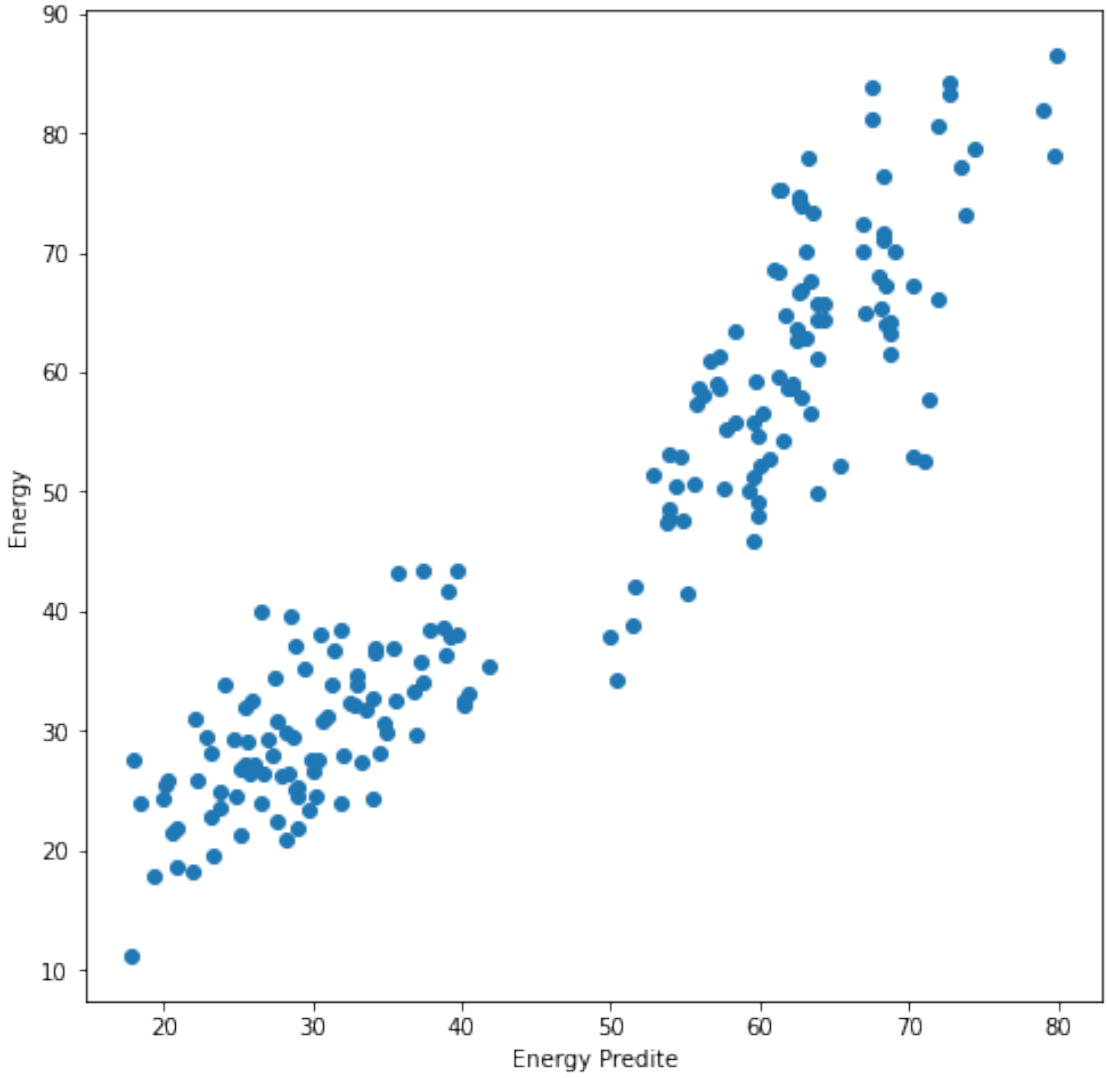


Interpretation

Nous observons que, nos residus ont des variances faibles et sont centrés autour de leur moyenne 0.Donc le modèle linéaire pour notre jeu de données à tendance à expliquer 87% des variations des energies des immeubles.

[138]:
```
plt.plot(Y_test,Yrel_test,"o")
plt.xlabel(u"Energy Predite")
plt.ylabel("Energy")
```

```
plt.show()
```



Interpretation

Nous observons des nuages de points regroupés et dont ayant une varaince faible.Donc notre modèle predit des données de manière precise avec une performance faible

[142]:
```
df_Energy_Pred = pd.DataFrame(Y_test,columns=['Prediction'])
```

[143]:
```
df_Energy_Pred['Prediction_Energy'] = pd.cut(x=df_Energy_Pred['Prediction'],
                                             bins=[-1,30, 35, 45, 55, 65, 75,␣
↪100], labels=["A", "B", "C", "D", "E", "F", "G"])
```

[144]:
```
Y_test1=df_Energy_Pred['Prediction_Energy'].values
```

[146]:
```
# Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(Y_test1,Ycat_test, normalize="columns")
print(table.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                  0.81  0.26  0.16  0.00  0.00  0.00  0.00
B                  0.15  0.43  0.20  0.00  0.00  0.00  0.00
C                  0.04  0.26  0.48  0.00  0.00  0.00  0.00
D                  0.00  0.04  0.12  0.35  0.00  0.00  0.00
E                  0.00  0.00  0.04  0.52  0.81  0.52  0.21
F                  0.00  0.00  0.00  0.13  0.19  0.48  0.57
G                  0.00  0.00  0.00  0.00  0.00  0.00  0.21
```

Interprètation :

Nous constatons qu'il, ** 81% des immeubles ayant une **energie** $<$ **30** (de la **classe A**), 81% de la **classe E** 57% de la **classe G** ont été correctement prédits. Les autres catégories ont été correctement prédites avec un pourcentage inférieure à 50%.

[149]:
```
df_Energy_CV = pd.DataFrame(Ycross,columns=['Prediction'])
```

[150]:
```
df_Energy_CV['Prediction_Energy'] = pd.cut(x=df_Energy_CV['Prediction'],
                                           bins=[-1,30, 35, 45, 55, 65, 75,␣
 ↪100], labels=["A", "B", "C", "D", "E", "F", "G"])
```

[151]:
```
Ycross1=df_Energy_Pred['Prediction_Energy'].values
```

[152]:
```
# Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(Ycross1,Ycat_test, normalize="columns")
print(table.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                  0.81  0.26  0.16  0.00  0.00  0.00  0.00
B                  0.15  0.43  0.20  0.00  0.00  0.00  0.00
C                  0.04  0.26  0.48  0.00  0.00  0.00  0.00
D                  0.00  0.04  0.12  0.35  0.00  0.00  0.00
E                  0.00  0.00  0.04  0.52  0.81  0.52  0.21
F                  0.00  0.00  0.00  0.13  0.19  0.48  0.57
G                  0.00  0.00  0.00  0.00  0.00  0.00  0.21
```

Interprètation :

Nous constatons que la cross-validation n'ameliore pas les resultats trouvées précèdemment dans la regression linéaire simple

Regression polynomial

[153]:
```
from sklearn.preprocessing import PolynomialFeatures
```

[154]:
```
pr = PolynomialFeatures(degree=3)

pr
```

[154]:
```
PolynomialFeatures(degree=3)
```

[155]:
```
Z_pr = pr.fit_transform(Xr_train)
```

[158]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

[159]:
```
Input = [('polynomial', PolynomialFeatures(include_bias=False)),␣
 ↪('model',LinearRegression())]
```

[160]:
```
pipe = Pipeline(Input)
pipe
```

[160]:
```
Pipeline(steps=[('polynomial', PolynomialFeatures(include_bias=False)),
                ('model', LinearRegression())])
```

[161]:
```
pipe.fit(Xr_train,Yrel_train)
```

[161]:
```
Pipeline(steps=[('polynomial', PolynomialFeatures(include_bias=False)),
                ('model', LinearRegression())])
```

[162]:
```
ypipe = pipe.predict(Xr_train)
ypipe[0:4]
```

[162]:
```
array([31.5625, 62.9375, 69.0625, 33.125 ])
```

[163]:
```
from sklearn.metrics import r2_score
```

[164]:
```
r_square = r2_score(Yrel_train, ypipe)

print('The R-square value is: ', r_square)

ypipe.shape
```

```
The R-square value is:  0.9027413497264247
```

[164]:
```
(576,)
```

[168]:
```
ypipe1 = pipe.predict(Xr_test)
ypipe1[0:5]
```

[168]:
```
array([29.46875, 25.375  , 51.9375 , 69.4375 , 33.6875 ])
```

[169]:
```
r_square2 = r2_score(Yrel_test, ypipe1)

print('The R-square value is: ', r_square2)
```

```
The R-square value is:  0.852926180193692
```

[170]:
```
print("True values:", Yrel_test[0:5].values)
```

```
True values: [29.88301358 28.20553275 70.03570637 73.1143689  34.16136399]
```

[171]:
```
# Erreur quadratique moyenne

from sklearn.metrics import mean_squared_error

print("MSE=", mean_squared_error(Yrel_test, ypipe1,))
```

```
MSE= 50.4764288494786
```

Interpretation :

Nous constatons qu'avec une regression polynomial degrée 3, est un modèle qui ne généralise pas avec les données tests qu'il n'a pas encore vu.Car le score sur les données d'entrainements sont superieurs au score sur les données tests et de la meme manière les erreurs sur les données d'entrainements sont moins elévés que celles des données tests

[172]:
```
df_Energy_Prr = pd.DataFrame(ypipe1,columns=['Prediction'])
```

[173]:
```
df_Energy_Prr['Prediction_Energy'] = pd.cut(x=df_Energy_Prr['Prediction'],
                                            bins=[-1,30, 35, 45, 55, 65, 75,␣
↪100], labels=["A", "B", "C", "D", "E", "F", "G"])


ypipe112=df_Energy_Prr['Prediction_Energy'].values
```

[174]:
```
# Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(ypipe112,Ycat_test)
print(table)
```

```
Energy.efficiency   A   B   C  D   E   F  G
row_0
A                  40   8   2  0   0   0  0
B                  10  11  11  0   0   0  0
C                   2   4  10  2   2   0  0
D                   0   0   2  9   7   2  0
E                   0   0   0  9  11   5  1
F                   0   0   0  3   9  14  5
G                   0   0   0  0   3   2  8
```

**Interprètation :

Cette matrice de confusion, nous permet de confirmer les resultats obtenu plus haut.Le taux de mauvais classement à augmenter.

Regression Linéaire Sans Pénalisation avec suppression des Variables

[184]:
```
lm1.fit(Xre_train,
        Yreel_train)
```

[184]:
```
LinearRegression()
```

[185]:
```
Ye_test = lm1.predict(Xre_test)
```

[186]:
```
lm1.score(Xre_test,Yreel_test)
```

[186]:
```
0.8779910607146666
```

[187]:
```
print("R2=", r2_score(Yreel_test,Ye_test))
```

```
R2= 0.8779910607146666
```

[188]:
```
lm1.score(Xre_train,Yreel_train)
```

[188]:
```
0.8751445874677167
```

[189]:
```
lm1.intercept_, lm.coef_
```

[189]:
```
(47.22043342360051,
 array([-8.27493447, -0.23253181, -8.8171094 ,  4.85459163,  0.09168762,
        -0.42491197,  0.12527706,  0.22386987,  0.07528743, -8.00233881,
         8.00233881]))
```

[190]:
```
Ye_hatest = lm1.predict(Xre_test)
```

[191]:
```
print("MSE=", mean_squared_error(Yreel_test, Ye_hatest))
```

```
MSE= 41.87404359897064
```

[192]:
```
Ye_hat = lm1.predict(Xre_train)
```

[193]:
```
# Erreur quadratique moyenne

from sklearn.metrics import mean_squared_error

print("MSE=", mean_squared_error(Yreel_train, Ye_hat))
```

```
MSE= 51.267234747754486
```

[194]:
```
df_energy_PredS = pd.DataFrame(Ye_hatest,columns=['Prediction'])
df_energy_PredS['Prediction_Energy'] = pd.cut(x=df_energy_PredS['Prediction'],␣
↪bins=[-1,30, 35, 45, 55, 65, 75, 100], labels=["A", "B", "C", "D", "E", "F",␣
↪"G"])
```

```
PrevS=df_energy_PredS['Prediction_Energy'].values
# Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(PrevS,Ycato_test)
print(table)
```

```
Energy.efficiency   A   B   C   D   E   F  G
row_0
A                  44   6   4   0   0   0  0
B                   7  10   5   0   0   0  0
C                   1   6  12   0   0   0  0
D                   0   1   4   8   0   0  0
E                   0   0   0  13  26  12  3
F                   0   0   0   2   6  11  8
G                   0   0   0   0   0   0  3
```

Interpretation

Dans cette partie, nous constatons qu'en supprimant certaines variables qualitatives nous obtenons des resultats faiblements ameliorés comparés aux résultats obtenus précèdemment dans le cas de la régression simple ci-dessus sans suppression de ces variables. Donc, ces variables n'ont pas d'influence sur la variable à expliquer "Energy"

Conclusion:

Dans la suite de nos travaux, Nous n'allons pas supprimer les variables qualitatives precèdentes.

4.2.2 Regression Linéaire avec pénalisation

Regression linéaire avec une pénalisation de Lasso

[197]:
```
regLasso = linear_model.Lasso()
reg=regLasso.fit(Xr_train,Yrel_train)

prev=regLasso.predict(Xr_test)

prev1 = regLasso.predict(Xr_train)

print("MSE test= %f, MSE train=%f" % (mean_squared_error(Yrel_test,prev),␣
 ↪mean_squared_error(Yrel_train,prev1)))
```

```
MSE test= 46.613564, MSE train=55.228435
```

[198]:
```
print("R_ square test= %f, R_square train = %f" % (r2_score(Yrel_test, prev),␣
 ↪r2_score(Yrel_train, prev1)))
```

```
R_ square test= 0.864181, R_square train = 0.865498
```

Interpretation

Nous constatons que, les resultats obtenus avec la penalisation Lasso ne sont pas améliorés par rapport à ceux obtenus plus haut.

[199]:
```
from sklearn.model_selection import GridSearchCV
# grille de valeurs du paramètre alpha à optimiser
param=[{"alpha":[0.05,0.1,0.2,0.3,0.4,0.5,1]}]
regLasso = GridSearchCV(linear_model.Lasso(), param,cv=4,n_jobs=-1)
regLassOpt=regLasso.fit(Xr_train, Yrel_train)
# paramètre optimal!
regLassOpt.best_params_["alpha"]
print("Meilleur R2 = %f, Meilleur paramètre = %s" % (regLassOpt.
 ↪best_score_,regLassOpt.best_params_))
```

```
Meilleur R2 = 0.870399, Meilleur paramètre = {'alpha': 0.05}
```

[200]:
```
prev12=regLassOpt.predict(Xr_test)
print("MSE=",mean_squared_error(prev12,Yrel_test))
print("R2=",r2_score(Yrel_test,prev12))
```

```
MSE= 42.59421747901904
R2= 0.8758926808197793
```

Interpretation:

Nous constatons ici, que les resultats ne sont pas améliorés meme en utilisant des hyperparamètres (GridSearch) de la regularisation Lasso, au vu des resultats de la regression simple obtenu plus haut.

[205]:
```
# Coefficients
regLasso=linear_model.Lasso(alpha=regLassOpt.best_params_['alpha'])
model_lasso=regLasso.fit(Xr_train,Yrel_train)
model_lasso.coef_
```

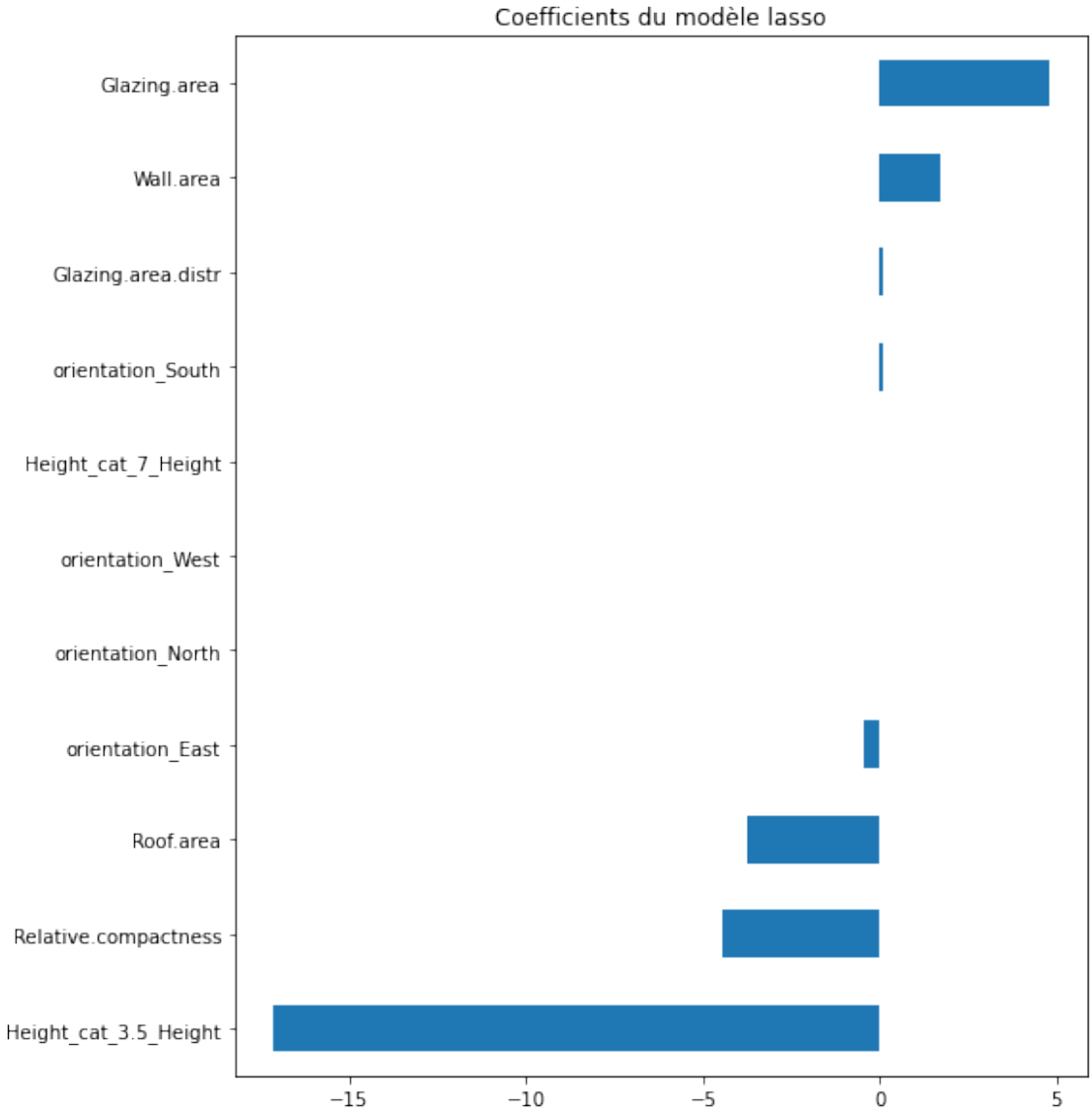
[205]:
```
array([ -4.44432357,   1.72291773,  -3.76166523,   4.80059603,
         0.08066735,  -0.46539981,   0.        ,   0.07340754,
        -0.        , -17.14330013,   0.        ])
```

[206]:
```
coef = pd.Series(model_lasso.coef_, index = X_train.columns)
print("Lasso conserve " + str(sum(coef != 0)) +
      " variables et en supprime " +  str(sum(coef == 0)))
```

```
Lasso conserve 8 variables et en supprime 3
```

[207]:
```
imp_coef = coef.sort_values()
plt.rcParams['figure.figsize'] = (8.0, 10.0)
imp_coef.plot(kind = "barh")
plt.title(u"Coefficients du modèle lasso")
```

[207]:
```
Text(0.5, 1.0, 'Coefficients du modèle lasso')
```



Interprètation

Nous observons les trois variables supprimées par Lasso qui sont : Orientation west, orientation North, Height_cat_7 et les 8 variables conservées sont : Height_cat_3.5_Height, Glazing_area, Wall_area, Glazing_area_distr, Orientation_East,Roof.area,et Relative.compactess.

Le graphe suivant permet d'identifier les bonnes et mauvaises prévisions de d'energies des differents seuils quis sont fixés: à 30 pour le type A,35 pour le type B, 45 pour le type C, 45 pour le type D, 55 pour le type E, 65 pour le type F, 75 pour le type G .

[209]:
```
df_energy_PredLasso = pd.DataFrame(prev12,columns=['Prediction'])
```

[210]:
```
df_energy_PredLasso['Prediction_Energy'] = pd.
↪cut(x=df_energy_PredLasso['Prediction'],
                                                    bins=[-1,30, 35, 45, 55, 65,␣
↪75, 100],
                                                    labels=["A", "B", "C", "D",␣
↪"E", "F", "G"])


Prevlass01=df_Energy_Pred['Prediction_Energy'].values
```

[212]:
```
# Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(Prevlass01,Ycat_test,normalize="columns")
print(table.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                  0.81  0.26  0.16  0.00  0.00  0.00  0.00
B                  0.15  0.43  0.20  0.00  0.00  0.00  0.00
C                  0.04  0.26  0.48  0.00  0.00  0.00  0.00
D                  0.00  0.04  0.12  0.35  0.00  0.00  0.00
E                  0.00  0.00  0.04  0.52  0.81  0.52  0.21
F                  0.00  0.00  0.00  0.13  0.19  0.48  0.57
G                  0.00  0.00  0.00  0.00  0.00  0.00  0.21
```

Scikit-learn propose d'autres procédures d'optimisation du paramètre de régularisation lasso par validation croisée en régression; lassoCV utilise un algorithme de coordinate descent, sans calcul de dérivée puisque la norme l1 n'est pas dérivable, tandis que lassoLarsCV est basée sur l'algorithme de least angle regression. Ces fonctions permettent de tracer également les chemins de régularisation. Voici l'exemple de lassoCV qui offre plus d'options (confère notebook python)

Regression avec Penalisation Ridge

[312]:
```
Xrr_train= lm.fit(Xr_train, Yrel_train)

Xrr_test= lm.fit(Xr_test, Yrel_test)
```

[313]:
```
from sklearn.linear_model import Ridge
```

Entrainement du modèle et prediction sur les données Tests

[314]:
```
RidgeModel = Ridge(alpha =0.1)

RidgeModel.fit(Xr_train, Yrel_train)

Ypre_test = RidgeModel.predict(Xr_test)
```

[315]:
```
print("R^2 train = %f, MSE train= %f" % (RidgeModel.score(Xr_train,␣
↪Yrel_train), 1. -RidgeModel.score(Xr_train, Yrel_train)))
```

```
print("R^2 test = %f, MSE test= %f" % (RidgeModel.score(Xr_test, Yrel_test), 1.␣
 ↪-RidgeModel.score(Xr_test, Yrel_test)))
```

```
R^2 train = 0.875983, MSE train= 0.124017
R^2 test = 0.876373, MSE test= 0.123627
```

[316]:
```
df_energy_PredRidge0 = pd.DataFrame(Ypre_test,columns=['Prediction'])
df_energy_PredRidge0['Prediction_Energy'] = pd.
 ↪cut(x=df_energy_PredRidge0['Prediction'],
                                                    bins=[-1,30, 35, 45, 55, 65,␣
 ↪75, 100],
                                                    labels=["A", "B", "C", "D",␣
 ↪"E", "F", "G"])


PrevRidge001=df_energy_PredRidge0['Prediction_Energy'].values
#
```

[317]:
```
#Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(PrevRidge001,Ycat_test)
print(table)
```

```
Energy.efficiency   A   B   C   D   E   F  G
row_0
A                  42   6   4   0   0   0  0
B                   8  10   5   0   0   0  0
C                   2   6  12   0   0   0  0
D                   0   1   3   8   0   0  0
E                   0   0   1  12  26  12  3
F                   0   0   0   3   6  11  8
G                   0   0   0   0   0   0  3
```

Interprètation:

Nous constatons que, la pénalisation de Ridge sans optimatisation n'améliore pas resultats, plus précisimment les classifications.

Nous Sélectionnons les valeurs de alpha qui minisent l'erreur sur les données tests

[318]:
```
Rsqu_test = []
Rsqu_train = []
dummy1 = []
ALFA = 10 * np.array(range(0,1000))
for alfa in ALFA:
    RidgeModel = Ridge(alpha = alfa)
    RidgeModel.fit(Xr_train, Yrel_train)
    Rsqu_test.append(RidgeModel.score(Xr_test, Yrel_test))
```

```
    Rsqu_train.append(RidgeModel.score(Xr_train, Yrel_train))
```

Ridge Regression avec GridSearch

[320]:
```
#Optimize parameter of penalization
param_Gr = [
    {'alpha':[1e-5,1e-4,1e-3,1e-2,0.1,1,10,100,1000,10000],
    'solver' :␣
 ↪['auto','svd','cholesky','lsqr','sparse_cg','sag','saga'],'random_state':[13]
    }
]

RR=Ridge()

Grid_RR=GridSearchCV(RR, param_Gr,cv=5,n_jobs=-1, verbose=1)

Grid_RR1=Grid_RR.fit(Xr_train, Yrel_train)

#  optimal parameter
print("best score = %f, best parameters = %s" % (Grid_RR1.best_score_,Grid_RR1.
 ↪best_params_))
```

```
Fitting 5 folds for each of 70 candidates, totalling 350 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  68 tasks      | elapsed:    4.2s

best score = 0.870136, best parameters = {'alpha': 0.1, 'random_state': 13,
'solver': 'sag'}

[Parallel(n_jobs=-1)]: Done 350 out of 350 | elapsed:    4.7s finished
```

[321]:
```
#Error on train data

1.-Grid_RR1.best_score_
```

[321]:
```
0.12986416618622276
```

[322]:
```
#Prediction on tests data
y_predrr = Grid_RR1.predict(Xr_test)

#Score on tests data

print("Score test = %f " % (Grid_RR1.score(Xr_test, Yrel_test)))

# error on test sample
print("MSE test = %f" % (1.-Grid_RR1.score(Xr_test, Yrel_test)))
```

```
Score test = 0.876328
MSE test = 0.123672
```

Interpretation

Nous constatons que, la pénalisation de ridge permet d'avoir un modèle aussi précis que celui de la regression linéaire simple sans pénalisation sur les données de tests, mais n'ameliore pas les resultats, donc les performances de manière significative.

[323]:
```
df_energy_PredRidge = pd.DataFrame(y_predrr,columns=['Prediction'])
df_energy_PredRidge['Prediction_Energy'] = pd.
 ↪cut(x=df_energy_PredRidge['Prediction'],
                                                    bins=[-1,30, 35, 45, 55, 65,␣
 ↪75, 100],
                                                    labels=["A", "B", "C", "D",␣
 ↪"E", "F", "G"])


PrevRidge01=df_energy_PredRidge['Prediction_Energy'].values
#
```

Le graphe suivant permet d'identifier les bonnes et mauvaises prévisions de d'energies des differents seuils quis sont fixés: à 30 pour le type A,35 pour le type B, 45 pour le type C, 45 pour le type D, 55 pour le type E, 65 pour le type F, 75 pour le type G .

[325]:
```
#Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(PrevRidge01,Ycat_test, normalize="columns")
print(table.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                  0.81  0.26  0.16  0.00  0.00  0.00  0.00
B                  0.15  0.43  0.20  0.00  0.00  0.00  0.00
C                  0.04  0.26  0.48  0.00  0.00  0.00  0.00
D                  0.00  0.04  0.12  0.35  0.00  0.00  0.00
E                  0.00  0.00  0.04  0.52  0.81  0.52  0.21
F                  0.00  0.00  0.00  0.13  0.19  0.48  0.57
G                  0.00  0.00  0.00  0.00  0.00  0.00  0.21
```

Interprètation

Nous constatons, d'après la matrice de confusion que la regression linéaire avec pénalisation Ridge optimisation, à un taux de faux positif moins que le modèle de regression linéaire avec pénalisation Ridge.

[327]:
```
# Coefficients de Ridge
RR3=Ridge(alpha=Grid_RR1.best_params_['alpha'], solver=Grid_RR1.
 ↪best_params_['solver'])
GrGrid_RR5=RR3.fit(Xr_train,Yrel_train)
GrGrid_RR5.coef_

#print(hh)
```

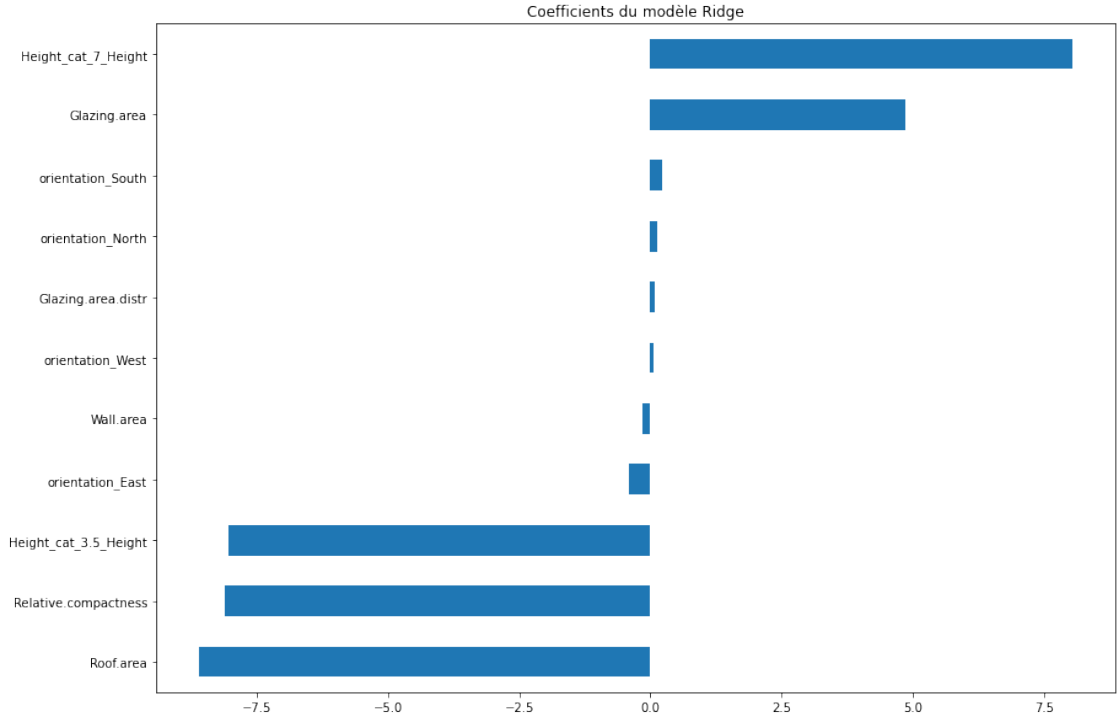
[327]:
```
array([-8.10590587, -0.15433035, -8.58979689,  4.8551846 ,  0.08855303,
       -0.415701  ,  0.12566105,  0.21743818,  0.07213059, -8.03539478,
        8.03539478])
```

[328]:
```
coef = pd.Series(GrGrid_RR5.coef_, index = X_train.columns)
print("Ridge conserve " + str(sum(coef != 0)) +
      " variables et en supprime " +  str(sum(coef == 0)))
```

```
Ridge conserve 11 variables et en supprime 0
```

[329]:
```
imp_coef = coef.sort_values()
plt.rcParams['figure.figsize'] = (14, 10.0)
imp_coef.plot(kind = "barh")
plt.title(u"Coefficients du modèle Ridge")
```

[329]:
```
Text(0.5, 1.0, 'Coefficients du modèle Ridge')
```



Interpretation Nous obtenons ci-dessus les coefficients de ridge. Et observons que, aucun de ces coefficients n'est nul.Ce qui nous permet de confirmer les propriètes de la pénalisation Ridge, à savoir celle de pénaliser les coefficients des variables ayant des grandes valeurs, sans en supprimer aucun des coefficients.

Optimal decision tree

[330]:
```
from sklearn.tree import DecisionTreeRegressor
from sklearn import tree

# Optimize the depth of tree
epochs=5
param=[{"max_depth":list(range(2,4)),
        'splitter':['best'],
        "criterion": ['mse','friedman_mse''mae'],"max_leaf_nodes":
 ↪list(range(2,3)),
        "min_samples_split":list(range(270,283)),"min_samples_leaf":
 ↪list(range(101,120)),
        "max_features": ['sqrt','auto'],"ccp_alpha":
 ↪list(range(0,1)),"min_impurity_split":[30], "random_state":[13]
       }]

tree= GridSearchCV(DecisionTreeRegressor(),param,cv=4,n_jobs=-1)
treeOpt=tree.fit(Xr_train, Yrel_train)

# paramètre optimal
print("best score = %f, best parameter = %s" % (treeOpt.best_score_,treeOpt.
 ↪best_params_))
```

```
best score = 0.776501, best parameter = {'ccp_alpha': 0, 'criterion': 'mse',
'max_depth': 2, 'max_features': 'sqrt', 'max_leaf_nodes': 2,
'min_impurity_split': 30, 'min_samples_leaf': 101, 'min_samples_split': 270,
'random_state': 13, 'splitter': 'best'}

C:\Users\ghomsik\anaconda3\envs\aa-projet\lib\site-
packages\sklearn\tree\_classes.py:306: FutureWarning:

The min_impurity_split parameter is deprecated. Its default value has changed
from 1e-7 to 0 in version 0.23, and it will be removed in 0.25. Use the
min_impurity_decrease parameter instead.
```

[331]:
```
# Error on train data

print("Error = %f" % (1-treeOpt.best_score_))
```

```
Error = 0.223499
```

[332]:
```
# Previction de l'echantillon test
y_chap = treeOpt.predict(Xr_test)

# Score and error on test sample

print("Score = %f, Error =%f" % (treeOpt.score(Xr_test, Yrel_test), 1.-treeOpt.
 ↪score(Xr_test, Yrel_test)))
```

```
Score = 0.739835, Error =0.260165
```

Interpretation

Nous observons que les arbres de décision n'ameliore pas les resultats tant sur les données d'apprentissages que sur les données, ceci autant sur le plan des performances que de la précision.Donc ce modèle n'est pas adapté.

[334]:
```
df_energy_ODT = pd.DataFrame(y_chap,columns=['Prediction'])
df_energy_ODT['Prediction_Energy'] = pd.cut(x=df_energy_ODT['Prediction'],
                                            bins=[-1,30, 35, 45, 55, 65, 75,␣
 ↪100],
                                            labels=["A", "B", "C", "D", "E",␣
 ↪"F", "G"])


PrevODT=df_energy_ODT['Prediction_Energy'].values
# Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(PrevODT,Ycat_test)
print(table.round(2))
```

```
Energy.efficiency   A   B   C   D   E   F   G
row_0
A                  52  22  21   0   0   0   0
F                   0   1   4  23  32  23  14
```

[335]:
```
print(y_chap[0:20])
```

```
[29.40116255 29.40116255 65.1638804  65.1638804  29.40116255 29.40116255
 29.40116255 29.40116255 65.1638804  65.1638804  65.1638804  65.1638804
 29.40116255 65.1638804  65.1638804  65.1638804  65.1638804  29.40116255
 29.40116255 29.40116255]
```

[336]:
```
{'ccp_alpha': 0, 'criterion': 'mse',
 'max_depth': 2,
 'max_features': 'auto', 'max_leaf_nodes': 2,
 'min_impurity_split': 30, 'min_samples_leaf': 101,
 'min_samples_split': 270, 'random_state': 13,
 'splitter': 'best'}
```

[336]:
```
{'ccp_alpha': 0,
 'criterion': 'mse',
 'max_depth': 2,
 'max_features': 'auto',
 'max_leaf_nodes': 2,
 'min_impurity_split': 30,
 'min_samples_leaf': 101,
 'min_samples_split': 270,
```

```
 'random_state': 13,
 'splitter': 'best'}
```

[337]:
```
#Decsion Tree with optimal parameters
treeOptCR = DecisionTreeRegressor(max_depth=treeOpt.best_params_['max_depth'],
                                  ccp_alpha=treeOpt.best_params_['ccp_alpha'],
                                  max_features=treeOpt.
 ↪best_params_['max_features'],
                                  max_leaf_nodes=treeOpt.
 ↪best_params_['max_leaf_nodes'],
                                   min_impurity_split=treeOpt.
 ↪best_params_['min_impurity_split'],
                                  min_samples_leaf=treeOpt.
 ↪best_params_['min_samples_leaf'],
                                  min_samples_split=treeOpt.
 ↪best_params_['min_samples_split'],
                                  random_state=treeOpt.
 ↪best_params_['random_state'],
                                   criterion=treeOpt.best_params_['criterion'])
treeR = treeOptCR.fit(Xr_train, Yrel_train)
```

```
C:\Users\ghomsik\anaconda3\envs\aa-projet\lib\site-
packages\sklearn\tree\_classes.py:306: FutureWarning:

The min_impurity_split parameter is deprecated. Its default value has changed
from 1e-7 to 0 in version 0.23, and it will be removed in 0.25. Use the
min_impurity_decrease parameter instead.
```

[338]:
```
#features importances
treeOptCR.feature_importances_
```

[338]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.])
```

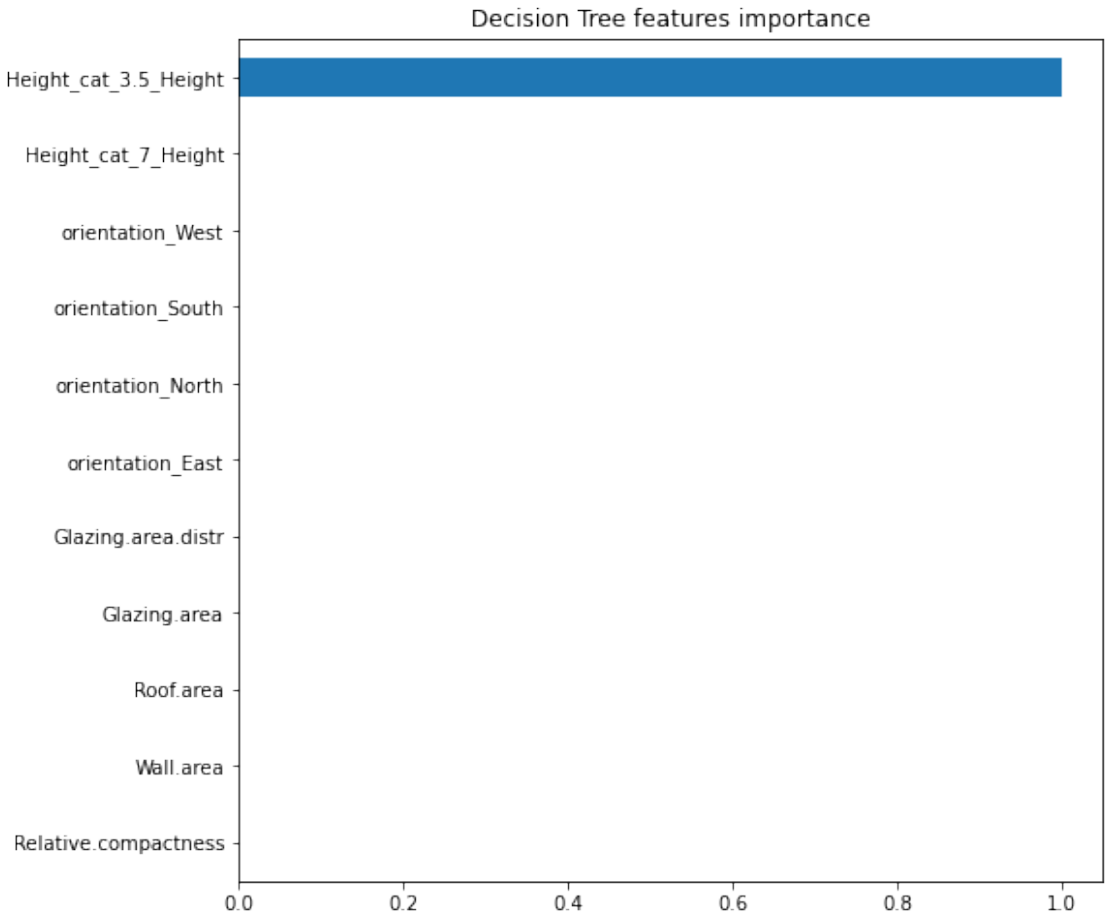
[339]:
```
importances = treeOptCR.feature_importances_
indices = np.argsort(importances)[::-1]
for f in range(Xr_train.shape[1]):
    print(df_ener.columns[indices[f]], importances[indices[f]])
```

```
Height_cat_3.5_Height 1.0
Height_cat_7_Height 0.0
orientation_West 0.0
orientation_South 0.0
orientation_North 0.0
orientation_East 0.0
Glazing.area.distr 0.0
Glazing.area 0.0
Roof.area 0.0
```

```
Wall.area 0.0
Relative.compactness 0.0
```

[340]:
```
#plot features importance
featImport = pd.Series(importances, index = X_train.columns)
imp_featImport = featImport.sort_values()
plt.rcParams['figure.figsize'] = (8.0, 8.0)
imp_featImport.plot(kind = "barh")
plt.title(u"Decision Tree features importance")
```

[340]:
```
Text(0.5, 1.0, 'Decision Tree features importance')
```



Interprètation: ** L'arbre de decision n'améliore pas la classification. Comparé à la regression linéaire, on a ici un une augmentaion du taux des mals classés des classes B,C,D, E et G. Il n y a plus que les classes A, F qui ont des taux de bien classés égale à 1.

RANDOM FOREST

[345]:
```
from sklearn.ensemble import RandomForestRegressor
```

[346]:
```
# définition des paramètres
forest = RandomForestRegressor(max_depth=4, random_state=0,bootstrap=True,␣
↪oob_score=True,

                                ␣
↪verbose=1,ccp_alpha=2,max_leaf_nodes=4,max_features='auto',

                                min_samples_split=80, min_samples_leaf=10,

                                criterion='mse')
"""(n_estimators=500,
   criterion='mse', max_depth=None,
   min_samples_split=2, min_samples_leaf=1, max_leaf_nodes=None,
   bootstrap=True, oob_score=True)"""
# apprentissage
rgrFit = forest.fit(Xr_train,Yrel_train)
#print(rgrFit.oob_score_)

print("best score = %f" % (rgrFit.oob_score_))

print("Error = %f" % (1.-rgrFit.oob_score_))
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.1s finished
```

```
best score = 0.882461
Error = 0.117539
```

[347]:
```
Pred_RF = rgrFit.predict(Xr_test)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.0s finished
```

[349]:
```
df_energy_RF = pd.DataFrame(Pred_RF,columns=['Prediction'])
df_energy_RF['Prediction_Energy'] = pd.cut(x=df_energy_RF['Prediction'],␣
↪bins=[-1,30, 35, 45, 55, 65, 75, 100], labels=["A", "B", "C", "D", "E", "F",␣
↪"G"])


PrevRF=df_energy_RF['Prediction_Energy'].values
# Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(PrevRF,Ycat_test, normalize="columns")
print(table.round(2))
```

```
Energy.efficiency    A     B     C     D     E     F     G
row_0
A                  1.0  0.96  0.84  0.00  0.00  0.00  0.00
D                  0.0  0.04  0.16  0.65  0.22  0.00  0.00
```

```
E                  0.0  0.00  0.00  0.26  0.44  0.35  0.00
F                  0.0  0.00  0.00  0.09  0.19  0.39  0.07
G                  0.0  0.00  0.00  0.00  0.16  0.26  0.93
```

[350]:
```
# Score de prevision sur le test

rgrFit.score(Xr_test,Yrel_test)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.0s finished
```

[350]:
```
0.8608307478128002
```

[351]:
```
#erreur de prévision sur le test

1-rgrFit.score(Xr_test, Yrel_test)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.0s finished
```

[351]:
```
0.13916925218719978
```

[352]:
```
rgrFit.feature_importances_
```

[352]:
```
array([0.00762043, 0.07894   , 0.47669072, 0.0310249 , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.18520692,
       0.22051703])
```

Interprètation

Le RandomForest sans optimisation, ameliore faiblement les resultats (biais et variance) obtenus avec les modèles précèdents, tant sur les données d'entrainement que sur les données tests.Nous constatons qu'il ne prédit que5 classes:A,D,E,F et G.Donc n'améliore pas la classification energique de certaines classes (B et C) comparé à la régression linéaire.

Aussi,nous constatons que les variables importantes dans notre modèle sont: Relative compactness, Wall area, Roof area, glazing area, height_cat_3.5_height,height_cat_7_height

Random Forest avec GridSearchCV

[355]:
```
param=[{ "max_features":list(range(2,10,1))}]
rf= GridSearchCV(RandomForestRegressor(n_estimators=100,random_state=13),
        param,cv=5,n_jobs=-1)
rfOpt=rf.fit(Xr_train, Yrel_train)
# paramètre optimal
print("Meilleur score = %f, Erreur sur l'entrainement =%f, Meilleur paramètre =␣
 ↪%s" % (rfOpt.best_score_,
                                                                              ␣
 ↪       1. -rfOpt.best_score_,
                                                                              ␣
 ↪       rfOpt.best_params_))
```

```
Meilleur score = 0.945080, Erreur sur l'entrainement =0.054920, Meilleur
paramètre = {'max_features': 6}
```

[356]:
```
#Prevision sur les données tests

Ypre_RF= rfOpt.predict(Xr_test)

# Score de prévision sur le test

rfOpt.score(Xr_test,Yrel_test)
```

[356]:
```
0.9405234005569907
```

[357]:
```
# erreur de prévision sur le test

1.-rfOpt.score(Xr_test,Yrel_test)
```

[357]:
```
0.059476599443009315
```

[359]:
```
df_energy_RFo = pd.DataFrame(Ypre_RF,columns=['Prediction'])
df_energy_RFo['Prediction_Energy'] = pd.cut(x=df_energy_RFo['Prediction'],
                                            bins=[-1,30, 35, 45, 55, 65, 75,␣
 ↪100],
                                            labels=["A", "B", "C", "D", "E",␣
 ↪"F", "G"])


PrevRFo=df_energy_RFo['Prediction_Energy'].values
# Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(PrevRFo,Ycat_test, normalize="columns")
print(table.round(2))
```

```
Energy.efficiency     A     B     C     D     E     F     G
row_0
A                  0.87  0.17  0.16  0.00  0.00  0.00  0.00
B                  0.13  0.65  0.28  0.00  0.00  0.00  0.00
C                  0.00  0.17  0.48  0.04  0.00  0.00  0.00
D                  0.00  0.00  0.08  0.52  0.09  0.00  0.00
E                  0.00  0.00  0.00  0.43  0.59  0.22  0.00
F                  0.00  0.00  0.00  0.00  0.31  0.70  0.14
G                  0.00  0.00  0.00  0.00  0.00  0.09  0.86
```

Interpretation Avec optimaisation des paramètres du modèle Random Forest, nous constatons une nette amélioraton de la classification energetique (classes B et C).

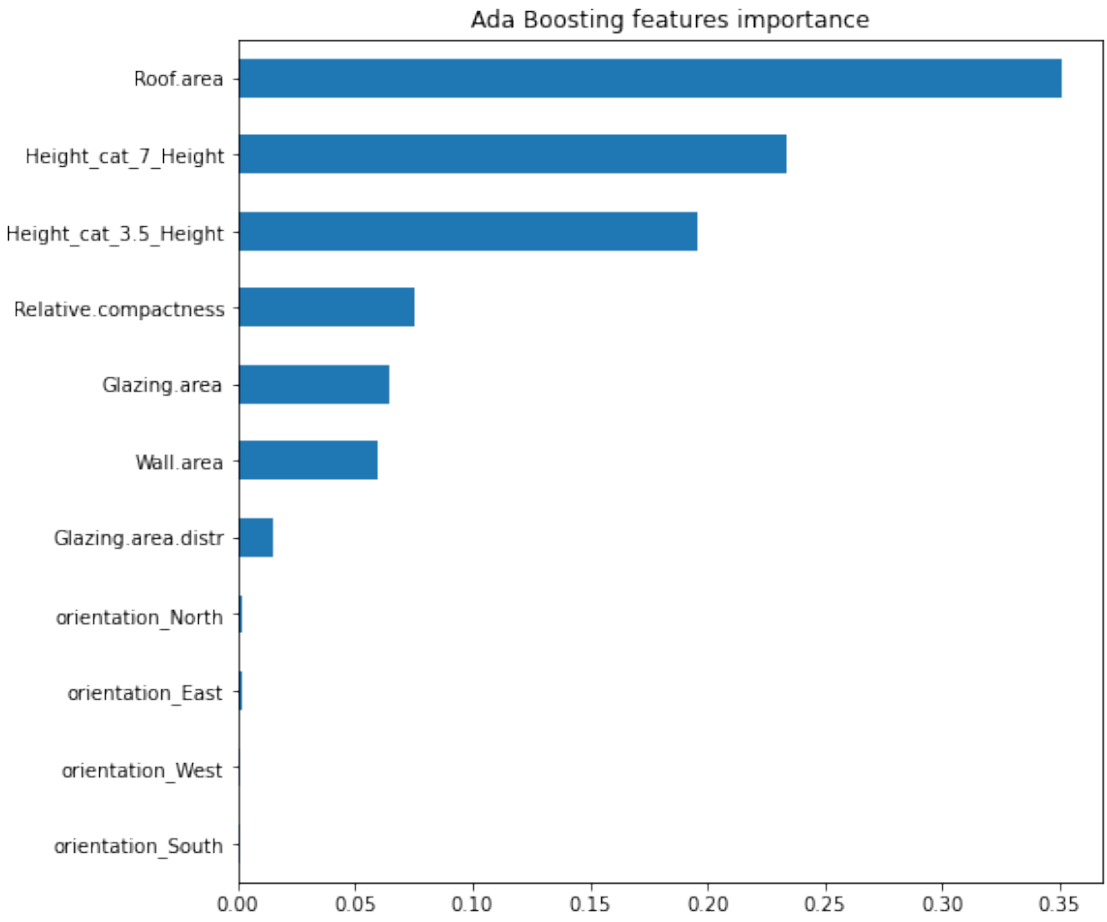
Comparé au *Decison tree*, le *Random Forest* dimunie le taux des mals classés des classes B,C, D,E, et F. Il fait également croître le taux de mals classés des classes A et F.

Ce modèle explique 94% de la variation de nos classes d'energies.

[360]:
```
#Random Forest with optimal parameters
RFR = RandomForestRegressor(max_features=rfOpt.best_params_['max_features'],
                                    random_state = 0
                                    )
RFR1 = RFR.fit(Xr_train, Yrel_train)

Ypre_RF1= rfOpt.predict(Xr_test)
```

[364]:
```
#plot features importance
featImport = pd.Series(RFR1 .feature_importances_, index = X_train.columns)
imp_featImport = featImport.sort_values()
plt.rcParams['figure.figsize'] = (8.0, 8.0)
imp_featImport.plot(kind = "barh")
plt.title(u"Random Forest features importance")
```

[364]:
```
Text(0.5, 1.0, 'Ada Boosting features importance')
```



Interprètation

Apès optimisation du modèle RandomForest, nous obtenon des resultats qui sont meilleurs que ceux obtenus avec les modèles precèdents (biais et variance) tant sur les données d'entrainement que sur les données tests.Car ce modèle generalise bien sur les données tests (score test très proche de celui du score train et erreur faible)

Aussi,nous constatons que les variables importantes dans notre modèle sont: Relative compactness, Wall area, Roof area, glazing area,glazing area distr height_cat_3.5_height,height_cat_7_height.Donc, elles influencent grandément notre classification, comparées à celle des orientations.

4.2.3 BOOSTING

AdaBoostRegressor

[365]:
```
from sklearn.ensemble import AdaBoostRegressor
# définition des paramètres
AdaBr = AdaBoostRegressor(base_estimator=DecisionTreeRegressor(ccp_alpha= 0,␣
↪max_depth= 2, max_leaf_nodes= 2),
                          n_estimators =10 ,learning_rate=1e-3,
                         loss="linear",random_state=13)

# apprentissage
AdaBrFit = AdaBr.fit(Xr_train,Yrel_train)
#print(AdaBr.score(Xr_train,Yrel_train))

print("best score = %f, Erreur sur les données d'entrainement=%f" % (AdaBr.
↪score(Xr_train,Yrel_train),
                                                                    1.-AdaBr.
↪score(Xr_train,Yrel_train) ))
```

```
best score = 0.778646, Erreur sur les données d'entrainement=0.221354
```

[366]:
```
#Prevision sur les données Tests
Ypre_Ada= AdaBr.predict(Xr_test)

# Score de prévision sur le test
AdaBr.score(Xr_test,Yrel_test)
```

[366]:
```
0.741274605039201
```

[367]:
```
# erreur de prévision sur le test
1.-AdaBr.score(Xr_test,Yrel_test)
```

[367]:
```
0.25872539496079905
```

Interpretation

Nous constatons que ce modèle produit des resultats(Performance et précision faibles tant sur les données d'apprentissages que les données tests.

[368]:
```
AdaBr.feature_importances_
```

[368]:
```
array([0.        , 0.        , 0.69990231, 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.09886021,
       0.20123748])
```

Interpretation

Nous constatons que les paramètres importants pour ce modèle sont : Roof area, height_cat_3.5_height,height_cat_7_height

[371]:
```
df_energy_AB = pd.DataFrame(Ypre_Ada,columns=['Prediction'])
df_energy_AB['Prediction_Energy'] = pd.cut(x=df_energy_AB['Prediction'],
                                           bins=[-1,30, 35, 45, 55, 65, 75,␣
 ↪100],
                                           labels=["A", "B", "C", "D", "E",␣
 ↪"F", "G"])


PrevAB=df_energy_AB['Prediction_Energy'].values
# Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(PrevAB,Ycat_test, normalize="columns")
print(table.round(2))
```

Energy.efficiency	A	B	C	D	E	F	G
row_0							
A	1.0	0.96	0.84	0.0	0.0	0.0	0.0
E	0.0	0.04	0.16	1.0	1.0	1.0	1.0

Interpretation

Nous constatons que notre modèle ne predit que deux (02) classes : A et E. Ce qui entraine une augmentation du taux des mals classés des classes B, C,D,F et G.

Donc n'améliore pas les résultats des bien classés

AdaBoost Optimisé avec GridSearchCV

[372]:
```
param=[{"n_estimators": [50, 150],"learning_rate": [0.01,0.05,0.1,0.3,1],
 "loss" : ["linear", "square", "exponential"]}]
AdaBrG= GridSearchCV(AdaBoostRegressor(DecisionTreeRegressor(ccp_alpha=0,␣
 ↪criterion="mse", max_depth= 2,
                                                              max_features␣
 ↪="sqrt", max_leaf_nodes= 2, random_state=13,
                                                             ␣
 ↪min_samples_leaf=109, min_samples_split=275, splitter = "best")),
                      param,cv=5,n_jobs=-1, verbose=1)
AdaBrGr=AdaBrG.fit(Xr_train, Yrel_train)
# paramètre optimal
print("Meilleur score = %f, Erreur sur l'entrainement =%f, Meilleur paramètre =␣
 ↪%s" % (AdaBrGr.best_score_,
```

```
↪       1. -AdaBrGr.best_score_,
↪       AdaBrGr.best_params_))
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  76 tasks      | elapsed:    7.5s
[Parallel(n_jobs=-1)]: Done 143 out of 150 | elapsed:   11.9s remaining:    0.5s
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:   13.2s finished

Meilleur score = 0.795889, Erreur sur l'entrainement =0.204111, Meilleur
paramètre = {'learning_rate': 0.3, 'loss': 'exponential', 'n_estimators': 150}
```

[373]:
```
#Prediction du modèle sur les données tests

Ypre_AdaBrGr=AdaBrGr.predict(Xr_test)

#Score de prevision test

AdaBrGr.score(Xr_test, Yrel_test)
```

[373]:
```
0.7585620473002963
```

[374]:
```
#Erreur de prevision test
1.-AdaBrGr.score(Xr_test, Yrel_test)
```

[374]:
```
0.2414379526997037
```

Interpretation

Nous constatons qu'après optimisation des paramètres d'AdaBoost à l'aide du GridSearch, nous obtenons un modèle ayant des performances et précision par aussi meilleurs que les modèles précèdents (biais et variance) tant sur les données d'entrainement que sur les données tests.

[376]:
```
df_energy_AdaBr = pd.DataFrame(Ypre_AdaBrGr,columns=['Prediction'])
df_energy_AdaBr['Prediction_Energy'] = pd.cut(x=df_energy_AdaBr['Prediction'],
                                              bins=[-1,30, 35, 45, 55, 65, 75,␣
↪100],
                                              labels=["A", "B", "C", "D", "E",␣
↪"F", "G"])


PrevAdaBr=df_energy_AdaBr['Prediction_Energy'].values
# Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(PrevAdaBr,Ycat_test, normalize="columns")
print(table.round(2))
```

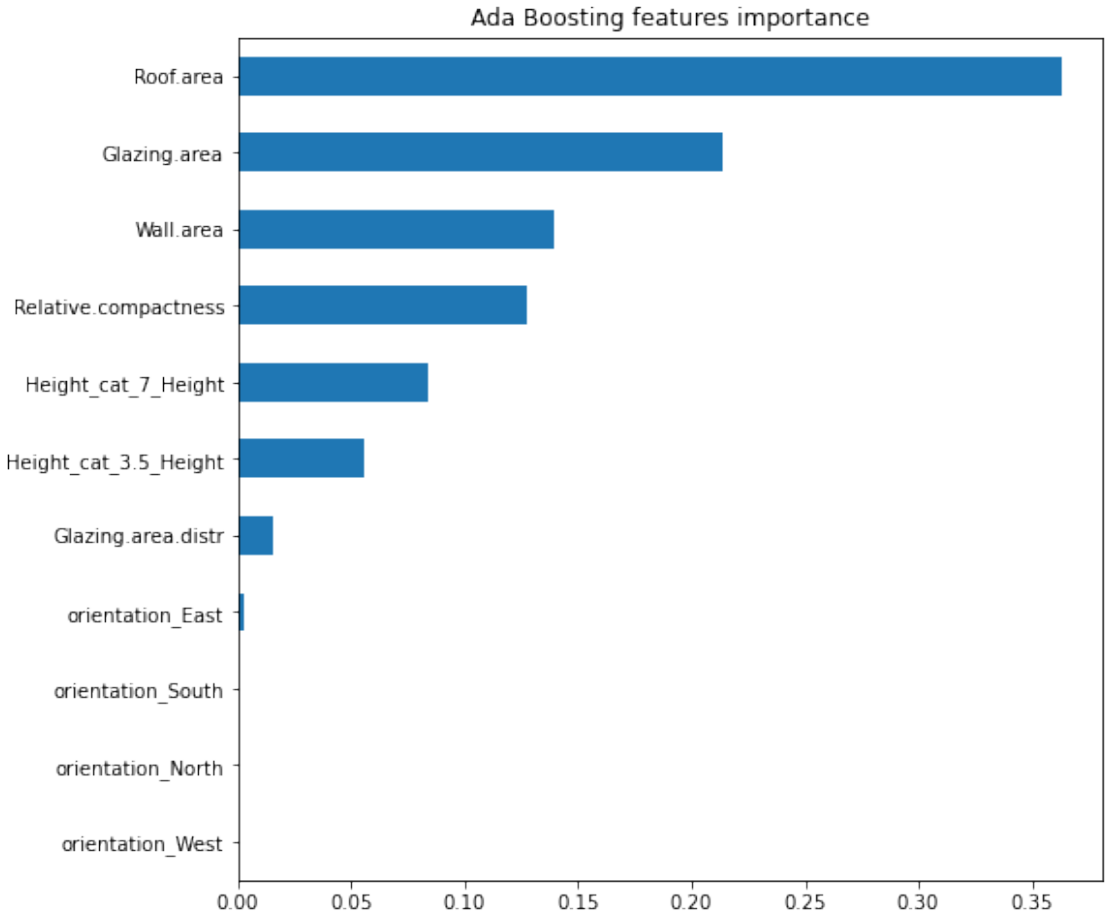
Energy.efficiency	A	B	C	D	E	F	G
row_0							
A	0.12	0.00	0.00	0.00	0.0	0.0	0.0
B	0.88	0.96	0.80	0.00	0.0	0.0	0.0
C	0.00	0.00	0.04	0.00	0.0	0.0	0.0
E	0.00	0.04	0.04	0.09	0.0	0.0	0.0
F	0.00	0.00	0.12	0.91	1.0	1.0	1.0

Interpètation

AdaBoostRegressor dimunie le taux des mals classés des classes C et F. Il ne prédit que les classes B,C,E et F.Mais diminue le taux des biens classés de la classe B et des autres classes A,D,et G

[381]:
```
#plot features importance
featImport = pd.Series(AdaBoostOp .feature_importances_, index = X_train.
 ↪columns)
imp_featImport = featImport.sort_values()
plt.rcParams['figure.figsize'] = (8.0, 8.0)
imp_featImport.plot(kind = "barh")
plt.title(u"Ada Boosting features importance")
```

[381]:
```
Text(0.5, 1.0, 'Ada Boosting features importance')
```



Interprètation Nous constatons que les variables importantes dans notre modèle sont: Relative compactness, Wall area, Roof area, glazing area,glazing area distr height_cat_3.5_height,height_cat_7_height.Donc, elles influencent grandément notre classification, comparées à celle des orientations.

GradientBoostingRegressor (voir notebook python)

4.2.4 Support Vector Machine

[ ]:
```
from sklearn.svm import SVR


param =[{"kernel": ("linear", "poly", "rbf", "sigmoid"),"C":[0.4,0.5,0.
↪8,1,2,5],"degree": [3,8],
        "gamma":[1**-8,1**-5,1**-3,1**-2,0.1,0.4,0.5],"tol":[0.001],"epsilon":
↪[0.1,0.2,0.3,,0.5,0.8]}]


SVRE = SVR()

GSCvSVr = GridSearchCV((SVRE), param, cv=5 ,n_jobs=-1, verbose=1)

SvR = GSCvSVr.fit(Xr_train, Yrel_train)

print("Meilleur score = %f, Erreur sur l'entrainement =%f, Meilleur paramètre =␣
↪%s" % (SvR.best_score_,
                                                                              ␣
↪       1. -SvR.best_score_,SvR.best_params_))
```

[387]:
```
#Score on train data

print("Score on tests data =%f" % (SvR.score(Xr_test, Yrel_test)))
```

```
Score on tests data =0.864163
```

[388]:
```
#Error on prevision data or tests data

print("Error on tests data = %f" % (1. -SvR.score(Xr_test, Yrel_test)))
```

```
Error on tests data = 0.135837
```

[389]:
```
# Prediction values

Y_predSVR = SvR.predict(Xr_test)
```

Interpretation

Les resultats obtenus par ce modèle, permettent de constater qu'il géneralise bien au niveau des données tests que d'apprentissage tant au niveau Variance que Biais.Mais, n'ameliore pas de manière significative les perfromances (biais et variance), obtenus par les autres modèles ci-dessus.

[391]:
```
df_energy_SVR= pd.DataFrame(Y_predSVR,columns=['Prediction'])
df_energy_SVR['Prediction_Energy'] = pd.cut(x=df_energy_SVR['Prediction'],
                                            bins=[-1,30, 35, 45, 55, 65, 75,␣
 ↪100],
                                            labels=["A", "B", "C", "D", "E",␣
 ↪"F", "G"])


PrevSVRr=df_energy_SVR['Prediction_Energy'].values
# Dénombrement des erreurs par
# matrice de confusion
table=pd.crosstab(PrevSVRr,Ycat_test, normalize="columns")
print(table.round(2))
```

Energy.efficiency	A	B	C	D	E	F	G
row_0							
A	0.79	0.35	0.16	0.00	0.00	0.00	0.00
B	0.17	0.43	0.20	0.00	0.00	0.00	0.00
C	0.04	0.22	0.52	0.09	0.03	0.00	0.00
D	0.00	0.00	0.08	0.30	0.25	0.09	0.00
E	0.00	0.00	0.04	0.52	0.44	0.22	0.00
F	0.00	0.00	0.00	0.09	0.28	0.57	0.43
G	0.00	0.00	0.00	0.00	0.00	0.13	0.57

Interpretation

Nous observons, que nos résidus sont centrés autour de la moyenne 0, et explique avec 86% la variations de données avec une erreur faible.

Conclusion partielle

Dans cette partie nous avons utilisé la méthode de classification indirecte pour prédire les différentes classes d'énergie de ce problème. Nous avons tout d'abord fait une prédiction basée sur la regression et ensuite appliqué la technique de seuil sur les valeurs prédites, pour obtenir des classes d'énergie. **Il ressort donc de cette analyse que le modèle de Random Forest avec une optimisation de paramètres par GridSearchCV a obtenu le meilleur taux de bien classés avec un pourcentage de 94% et un taux d'ereur de 0.05 sur les données d'entrainement et de test**. Les paramètres optimales sont : *'max_features'= 5, n_estimators=100*. Nous avons par ailleurs utilisé les méthodes de regression linéaire avec ou sans pénalisation,Arbre de décision optimal, Boosting et SVM.

5 CONCLUSION GENERALE

Ce travail présente une étude réalisé pour résoudre un problème de classification dans le but de prédire l'efficacité énergetique d'un bâtiment basé sur un ensemble de données simulées de 780 bâtiments.

Après avoir utilisé deux approches de classification, il en ressort que l'approche de classification indirecte (regression +application de seuil) avec le modèle de *Radom Forest* et avec une optimisation des paramètres, prédit avec le plus grand score de test, les classes d'énergie des bâtiments. Nous avons obtenu un taux d'erreur sur l'échantillon de test d'environ 0.05 avec cet approche indirecte tandis que qu'avec le même modèle (Random Forest) en utilisant l'approche directe, nous obtenons un taux de mal classés d'environ 0.36.

En raison des effets de non-linéarités présents dans notre jeu de données, les modèles de regression logistique et linéaire sous-performent dans nos résultats.

Dans notre jeu de données, certaines variables sont très fortement correlés. C'était le cas entre les surfaces totales, du toit et des murs. Nous avons donc après vérification, enlevé dans nos analyses, la variable redondante *Surface.area*.

De nos analyses, il en ressort également qu'il existe un lien entre la surrface des murs et la variable de compacité relative. Au-delà du coefficient de correlation qui nous apporte cette information, la formule mathématique de la compacité dépend de la surface des murs. Il serait intéressant dans les futurs travaux de jeter un regards sur les méthodes de réseau de neuronnes artificiels et en particulier sur *Multilayer Perceptron* qui pourrait donner un résultat de classification directe meilleur.

[ ]: