Clustering with K-means

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

import seaborn as sns

from sklearn.cluster import KMeans

from sklearn.model\_selection import cross\_val\_score

from xgboost import XGBRegressor

*# Set Matplotlib defaults*

plt.style.use("seaborn-whitegrid")

plt.rc("figure", autolayout=True)

plt.rc(

"axes",

labelweight="bold",

labelsize="large",

titleweight="bold",

titlesize=14,

titlepad=10,

)

def score\_dataset(X, y, model=XGBRegressor()):

*# Label encoding for categoricals*

for colname **in** X.select\_dtypes(["category", "object"]):

X[colname], \_ = X[colname].factorize()

*# Metric for Housing competition is RMSLE (Root Mean Squared Log Error)*

score = cross\_val\_score(

model, X, y, cv=5, scoring="neg\_mean\_squared\_log\_error",

)

score = -1 \* score.mean()

score = np.sqrt(score)

return score

*# Prepare data*

df = pd.read\_csv("../input/fe-course-data/ames.csv")

The k-means algorithm is sensitive to scale. This means we need to be thoughtful about how and whether we rescale our features since we might get very different results depending on our choices. As a rule of thumb, if the features are already directly comparable (like a test result at different times), then you would *not* want to rescale. On the other hand, features that aren't on comparable scales (like height and weight) will usually benefit from rescaling. Sometimes, the choice won't be clear though. In that case, you should try to use common sense, remembering that features with larger values will be weighted more heavily.

1) Scaling Features

Consider the following sets of features. For each, decide whether:

* they definitely should be rescaled,
* they definitely should *not* be rescaled, or
* either might be reasonable

Features:

1. Latitude and Longitude of cities in California
2. Lot Area and Living Area of houses in Ames, Iowa
3. Number of Doors and Horsepower of a 1989 model car

No, since rescaling would distort the natural distances described by Latitude and Longitude.

Either choice could be reasonable, but because the living area of a home tends to be more valuable per square foot, it would make sense to rescale these features so that lot area isn't weighted in the clustering out of proportion to its effect on SalePrice, if that is what you were trying to predict.

Yes, since these don't have comparable units. Without rescaling, the number of doors in a car (usually 2 or 4) would have negligible weight compared to its horsepower (usually in the hundreds).

What you should take away from this is that the decision of whether and how to rescale features is rarely automatic -- it will usually depend on some domain knowledge about your data and what you're trying to predict. Comparing different rescaling schemes through cross-validation can also be helpful. (You might like to check out the preprocessing module in scikit-learn for some of the rescaling methods it offers.)

2) Create a Feature of Cluster Labels

Creating a k-means clustering with the following parameters:

* features: LotArea, TotalBsmtSF, FirstFlrSF, SecondFlrSF,GrLivArea
* number of clusters: 10
* iterations: 10
* X = df.copy()
* y = X.pop("SalePrice")
* *# YOUR CODE HERE: Define a list of the features to be used for the clustering*
* features = [
* "LotArea",
* "TotalBsmtSF",
* "FirstFlrSF",
* "SecondFlrSF",
* "GrLivArea",
* ]
* *# Standardize*
* X\_scaled = X.loc[:, features]
* X\_scaled = (X\_scaled - X\_scaled.mean(axis=0)) / X\_scaled.std(axis=0)
* *# YOUR CODE HERE: Fit the KMeans model to X\_scaled and create the cluster labels*
* kmeans = KMeans(n\_clusters=10, n\_init=10, random\_state=0)
* X["Cluster"] = kmeans.fit\_predict(X\_scaled)

run this cell to see the result of the clustering

Xy = X.copy()

Xy["Cluster"] = Xy.Cluster.astype("category")

Xy["SalePrice"] = y

sns.relplot(

x="value", y="SalePrice", hue="Cluster", col="variable",

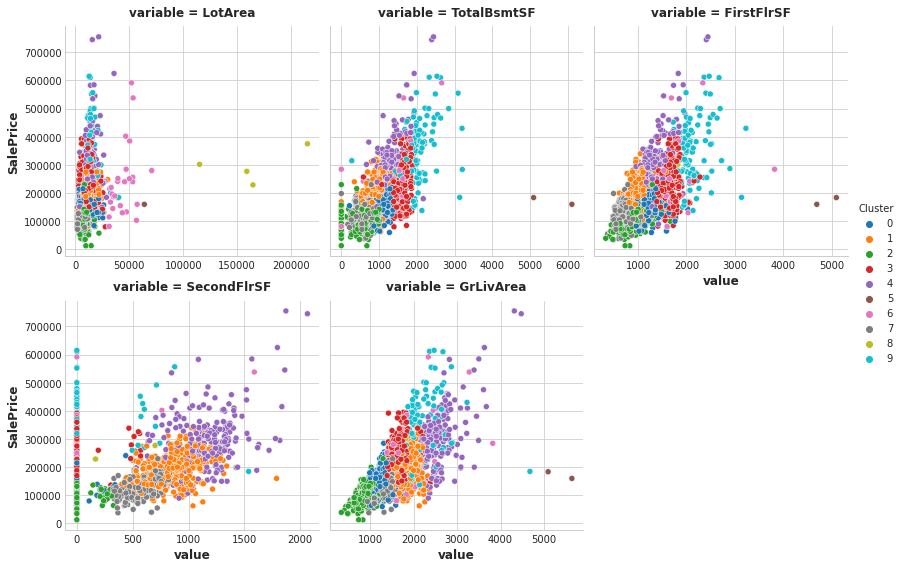
height=4, aspect=1, facet\_kws={'sharex': False}, col\_wrap=3,

data=Xy.melt(

value\_vars=features, id\_vars=["SalePrice", "Cluster"],

),

);



score\_dataset(X, y)

Out[6]:

0.142525791221533

The k-means algorithm offers an alternative way of creating features. Instead of labelling each feature with the nearest cluster centroid, it can measure the distance from a point to all the centroids and return those distances as features.

# 3) Cluster-Distance Features

Now add the cluster-distance features to your dataset. You can get these distance features by using the fit\_transform method of kmeans instead of fit\_predict.

kmeans = KMeans(n\_clusters=10, n\_init=10, random\_state=0)

*# YOUR CODE HERE: Create the cluster-distance features using `fit\_transform`*

X\_cd = kmeans.fit\_transform(X\_scaled)

*# Label features and join to dataset*

X\_cd = pd.DataFrame(X\_cd, columns=[f"Centroid\_**{**i**}**" for i **in** range(X\_cd.shape[1])])

X = X.join(X\_cd)