
UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS

MAESTRÍA EN CIENCAS DE DATOS

APRENDIZAJE AUTOMÁTICO

MAESTRO: JOSÉ ALBERTO BENAVIDES VÁZQUEZ

TAREA #5 APRENDIZAJE NO SUPERVISADO

ALUMNO: EDWIN MARTÍN ROMERO SILVA

MATRÍCULA: 1731276

**Índice**

### Introducción

Mi dataframe contiene 20,508 registros, 4 variables categóricas y 9 variables continuas. Investigué y tengo 2 opciones para aplicar una clusterización:

- Kmeans (Transformando las variables categóricas en numéricas)
- KPrototypes

```
categorical_cols = df.select_dtypes(include = ['object']).columns.tolist()
numerical_cols   = df.select_dtypes(include = ['float64', 'int64']).columns.tolist()
```

```
['CNT_CHILDREN',
 'AMT_INCOME_TOTAL',
 'DAYS_BIRTH',
 'DAYS_EMPLOYED',
 'FLAG_MOBIL',
 'FLAG_WORK_PHONE',  ['NAME_EDUCATION_TYPE',
 'FLAG_PHONE',        'NAME_FAMILY_STATUS',
 'FLAG_EMAIL',        'NAME_HOUSING_TYPE',
 'CNT_FAM_MEMBERS']   'Cat_Tipo_Ingresos']
```

### Desarrollo

#### Kmeans

El método K-Means es un algoritmo de aprendizaje automático no supervisado que agrupa datos en clusters o grupos de manera que los puntos de datos en un mismo grupo sean más similares entre sí que con los puntos en otros grupos. Utiliza la métrica de distancia euclidiana para determinar la similitud entre los puntos y seleccionar los grupos.

Desafortunadamente el algoritmo K Means solo funciona con variables numéricas, por lo que debo transformar mis variables categóricas de alguna forma, tengo 2 opciones:

- Reemplazar cada categoría con un número (Esto nunca me ha gustado)
- Crear variables dummy para las variables categórica, escogí esta opción.

#### Variables Dummy

La función get_dummies transforma una columna en 'n' columnas, una por cada categoría o posible valor que la variable pueda tomar. Estas 'n' columnas contienen valores únicamente de 0 y 1, lo que las convierte en variables numéricas. Este formato numérico permite su uso en algoritmos como K-Means.

```
df_dummy = pd.get_dummies(df, columns=categorical_cols)
```

### Escalar entre 0 y 1

Posteriormente lo que hacemos con las variables es aplicarles una transformación, para que todas tengan escala entre 0 y 1.

Esto nos ayuda bastante para que la distancia euclidiana no depende únicamente de las variables con mayor escala, si no que tome en cuenta todas las variables.

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
df_estandar = scaler.fit_transform(df_dummy)

df_estandar = pd.DataFrame(df_estandar, columns = df_dummy.columns)
```

### Elegir el número de clusters

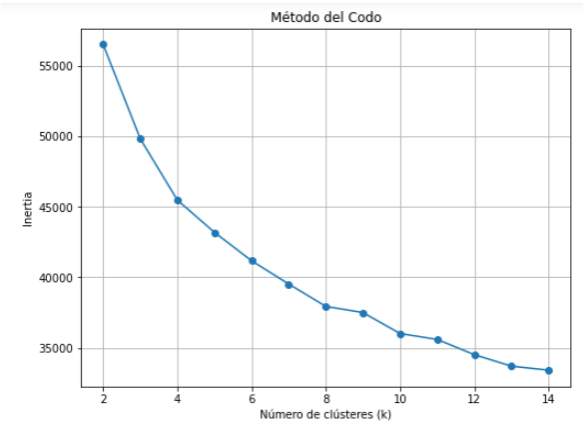
El dataframe ahora está listo para aplicarle el método de clusterización, pero hay que definir el número de grupos en el que se separará la base, para esto yo utilicé 2 métricas.

### Inertia

La inertia es una medida que evalúa la cohesión de los clusters generados por K-Means. La inertia es la sumatoria de la distancia al cuadrado entre el vector de variables de cada punto y su vector centroide.

La inertia disminuye cada que aumenta el número de grupos, pero tener un número muy grande de grupos podría no tener mucho sentido, todo depende del problema que se quiere resolver.

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

k_values = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
inertia_values = []  # Inicializa la lista de inercias

for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=0)
    kmeans.fit(df_estandar)
    inertia_values.append(kmeans.inertia_)

# Traces el gráfico de la suma de distancias en función de k
plt.figure(figsize=(8, 6))
plt.plot(k_values, inertia_values, marker='o')
plt.title('Método del Codo')
plt.xlabel('Número de clústeres (k)')
plt.ylabel('Inertia')
plt.grid()
plt.show()
```




Existe un método para elegir el número de grupos optimo con base en la inertia, es el método del codo:

```
from kneed import KneeLocator
prueba = KneeLocator(
x = k_values,
y = inertia_values,
S = 0.1, curve = 'convex', direction = 'decreasing', online = True)
prueba.elbow
```

```
6
```

En esta ocasión el método del codo nos dice que el número optimo es 6 grupos

### Calinski-Harabasz

Otro método para calcular el número de clusters es el índice de Calinski-Harabasz (también conocido como el índice de Varianza entre Grupos sobre Varianza dentro de Grupos).

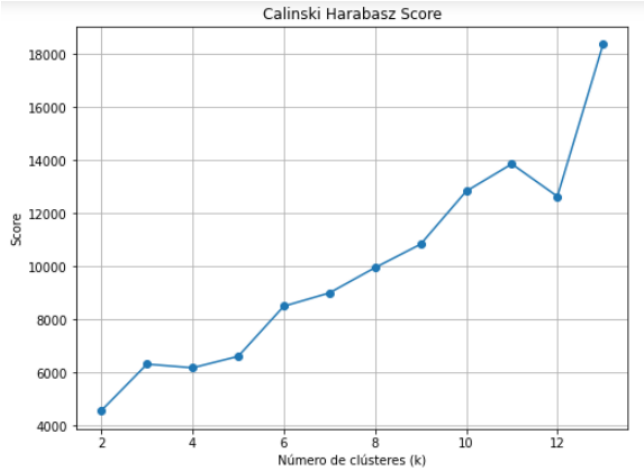
Esta es una métrica utilizada para evaluar la calidad de los clusters generados por un algoritmo de agrupamiento. Proporciona una medida de cuán bien definidos y separados están los clusters en un conjunto de datos.

Calinski-Harabasz = (Varianza entre grupos / Varianza dentro de grupos) * (N - k) / (k - 1)

- A mayor varianza entre grupos respecto al centroide global, mayor separados estan los clusters.
- A menor varianza de los puntos dado el centroide de cada cluster, mas unidos estan los puntos.

```python
from sklearn.metrics import calinski_harabasz_score
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

k_values = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
calinski_score = []  # Inicializa la lista de inercias

for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=0)
    kmeans.fit(df_estandar)
    df_estandar_copy = df_estandar.copy()

    df_estandar_copy['Cluster']= kmeans.labels_
    score = calinski_harabasz_score(df_estandar_copy,
                                    df_estandar_copy['Cluster'].values)
    calinski_score.append(score)
```

Un valor más alto indica una mejor separación entre los clusters. Por lo tanto, al elegir el número óptimo de clusters, se busca el valor de k que maximiza este índice.




Calculando el porcentaje de incremento del score para cada unidad adicional de

cluster, me parece que la mejor opción también es k = 6, ya que incrementa 28% respecto a k=5 y posteriormente ya no obtenemos un incremento tan grande hasta alcanzar k = 10.

| Score_calinski | k_values | %Incremento |
|---|---|---|
| 4,560.79 | 2 | NA |
| 6,308.41 | 3 | 38.3% |
| 6,169.64 | 4 | -2.2% |
| 6,601.97 | 5 | 7.0% |
| 8,490.65 | 6 | 28.6% |
| 8,990.85 | 7 | 5.9% |
| 9,942.27 | 8 | 10.6% |
| 10,824.52 | 9 | 8.9% |
| 12,822.18 | 10 | 18.5% |
| 13,849.95 | 11 | 8.0% |
| 12,626.92 | 12 | -8.8% |
| 18,352.97 | 13 | 45.3% |

Con ambos métodos puedo concluir que el número optimo de clusters es 6, pero bien podríamos utilizar otro número, siempre acorde a lso objetivos del ejercicio.

**Algoritmo en Python**

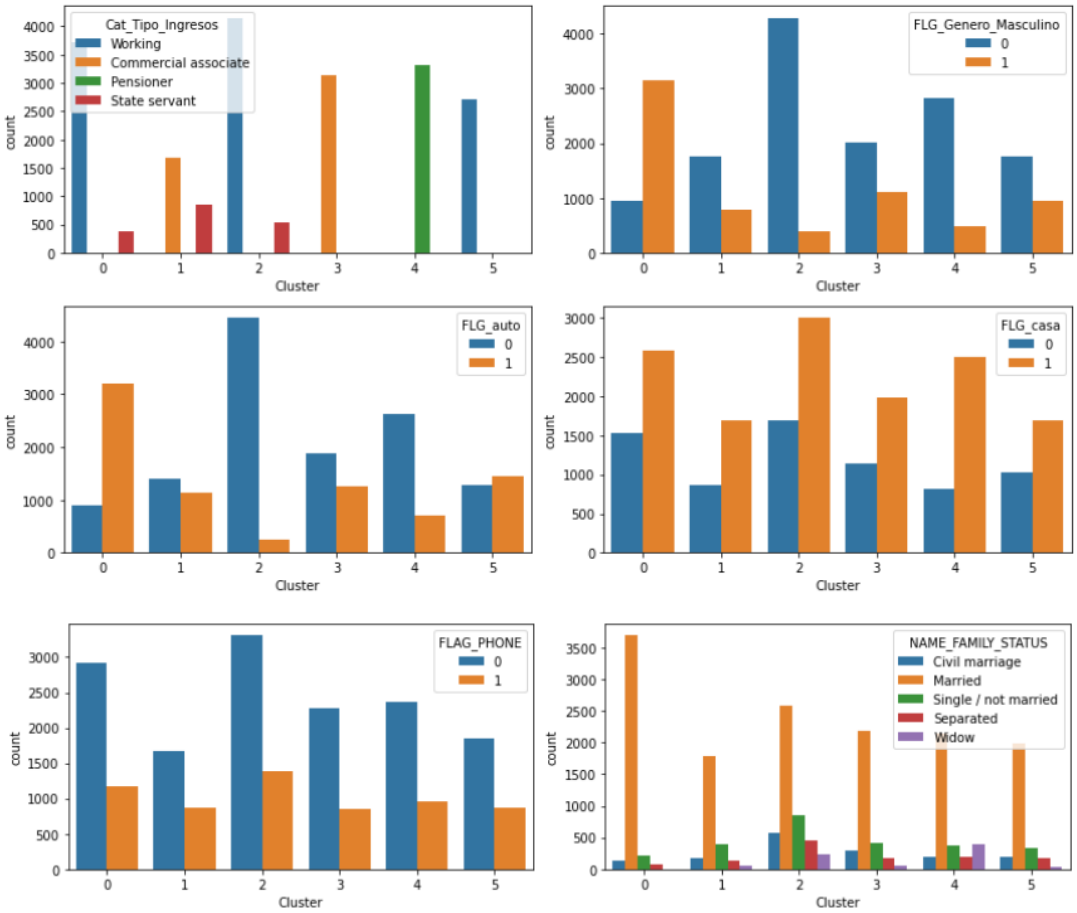
Con este simple código se aplica el método de k means.

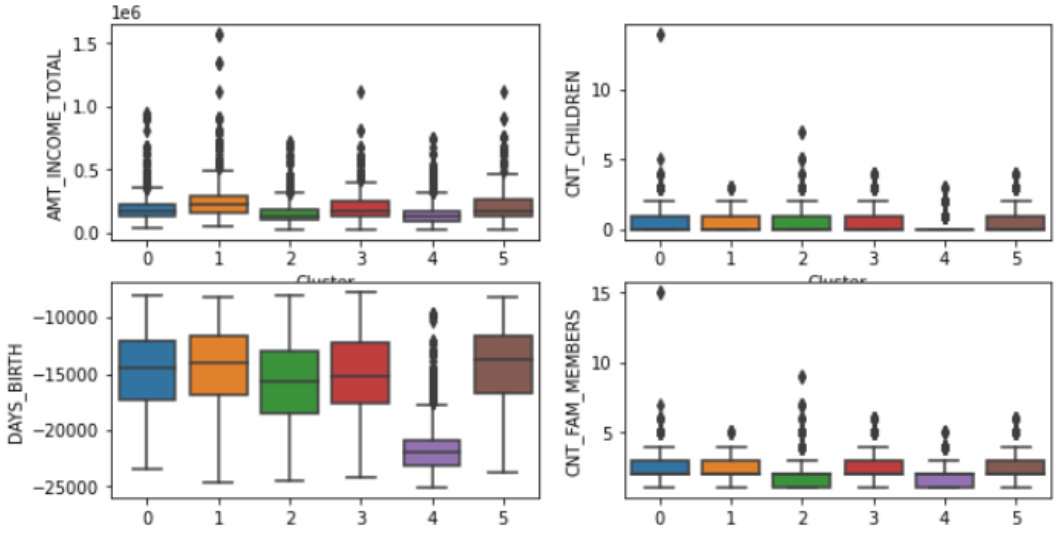
Ojo. Debe entrar el dataframe únicamente con variables en formato numérico y con los datos escalados entre 0 y 1.

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=6)

# Ajusta el modelo a tus datos
kmeans.fit(df_estandar)

# Agregar las etiquetas de clúster al DataFrame original
df['Cluster'] = kmeans.labels_
```

### Resultados

Finalmente visualizamos resultados




1e6
AMT_INCOME_TOTAL
1.5
1.0
0.5
0.0
0
1
2
3
4
5
CNT_CHILDREN
10
5
0
DAYS_BIRTH
−10000
−15000
−20000
−25000
CNT_FAM_MEMBERS
15
10
5

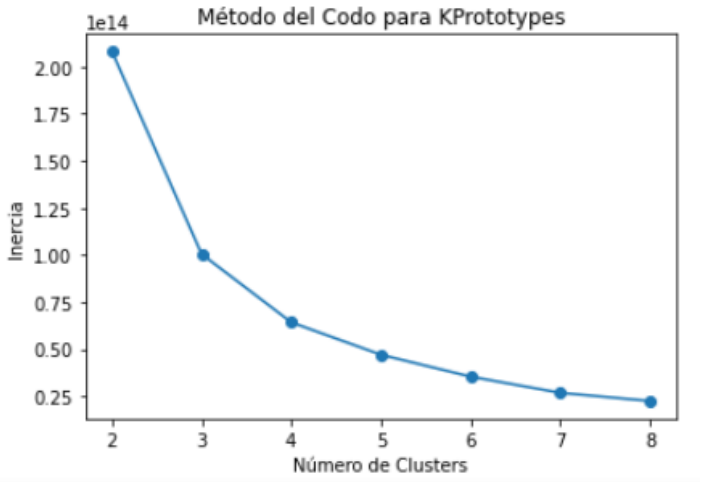
### kPrototypes

KPrototypes es una elección adecuada para mi dataframe ya que funciona para conjuntos de variables mixtos que incluyan tanto variables continuas como categóricas, sin necesidad de crear variables dummy para cada categoría posible.

### Elegir el número de clusters

Igual que en K means, necesitamos escoger el número óptimo de clusters. Utilizaré la incertia.

```python
from kmodes.kprototypes import KPrototypes
from sklearn.metrics import silhouette_score
import numpy as np

# Supongamos que has combinado tus datos en una matriz llamada 'X'
x = df[categorical_cols + numerical_cols]

# Rango de números de clusters que deseas probar
k_range = [2, 3, 4, 5, 6, 7, 8]

# Lista para almacenar los valores de Silhouette
inertias = []

for k in k_range:
    kproto = KPrototypes(n_clusters=k, init='Cao', verbose=2)
    clusters = kproto.fit_predict(x, categorical=[0, 1, 2, 3])
    inertia = kproto.cost_
    inertias.append(inertia)
```




El método del codo para esta métrica nos dice que el número optimo de clusters es k = 4.

```
from kneed import KneeLocator
prueba = KneeLocator(
x = k_range,
y = inertias,
S = 0.1, curve = 'convex', direction = 'decreasing', online = True)
prueba.elbow
```

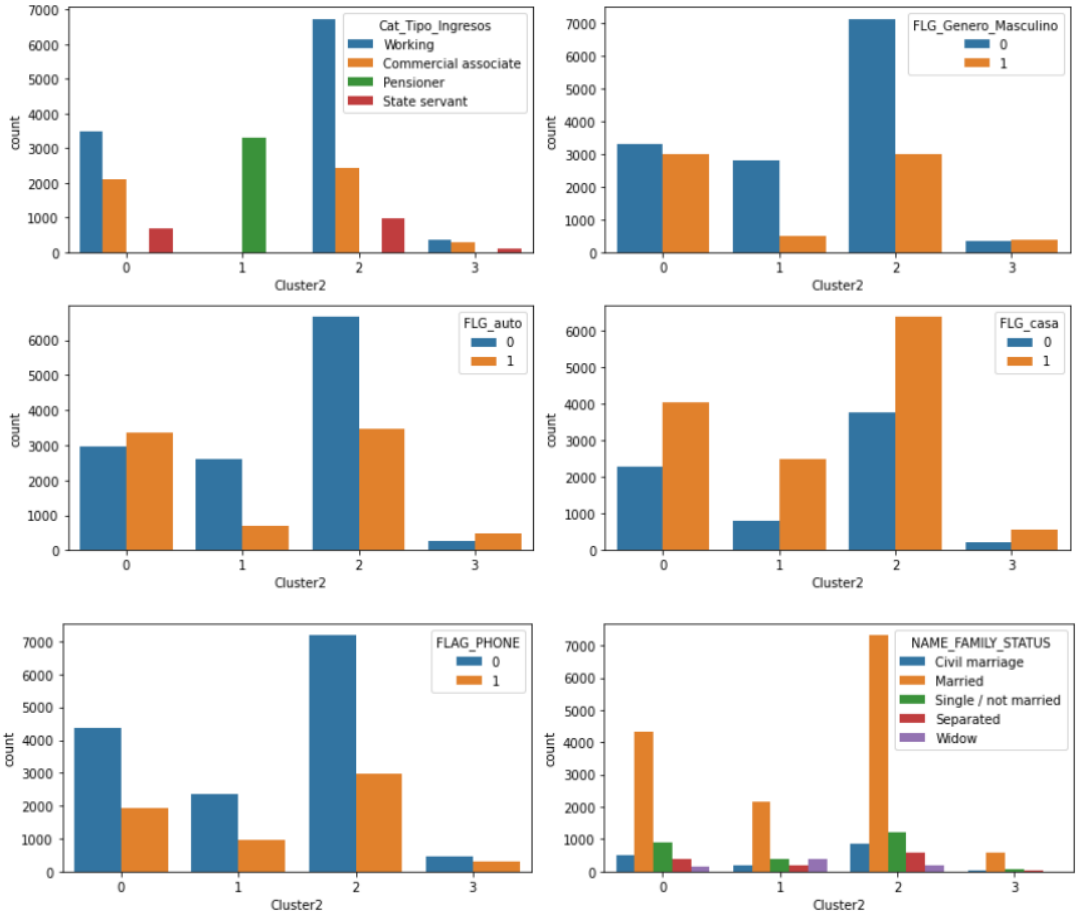
```
4
```

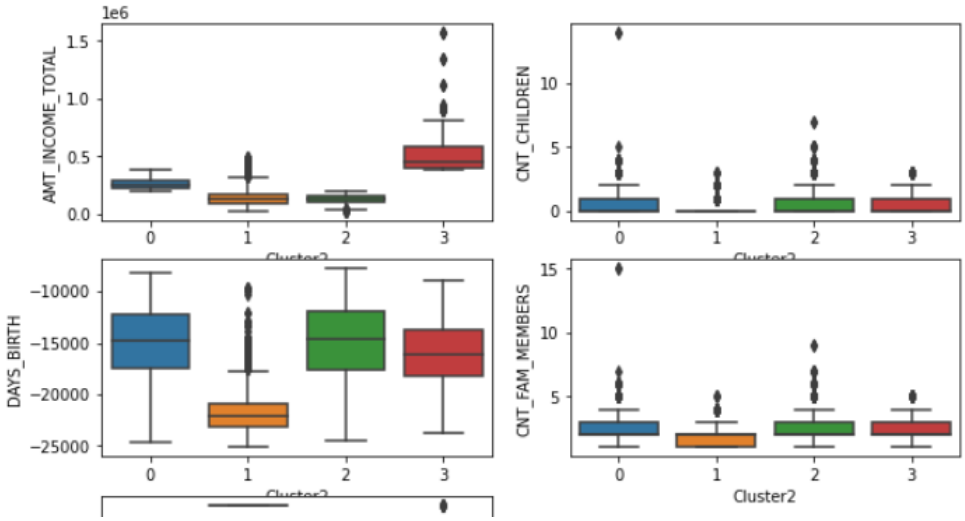
### Algoritmo de KPrototypes en Python

Para entrenar este algoritmo, debemos introducir el dataframe original, sin escalar ni variables dummy, además hay que especificar cuales son las variables categóricas, de lo contrario obtendremos un error.

```
# Supongamos que has combinado tus datos en una matriz llamada 'X'
from kmodes.kprototypes import KPrototypes


k = 4  # Número de clusters que desea
kproto = KPrototypes(n_clusters=k, init='Cao', verbose=2)
x = df[categorical_cols + numerical_cols]

clusters = kproto.fit_predict(x, categorical=[0, 1, 2, 3])
# Donde [0, 1, 2] son los índices de las columnas categóricas
```

## Resultados


Cat_Tipo_Ingresos
Working
Commercial associate
Pensioner
State servant
FLG_Genero_Masculino
0
1
FLG_auto
0
1
FLG_casa
0
1
FLAG_PHONE
0
1
NAME_FAMILY_STATUS
Civil marriage
Married
Single / not married
Separated
Widow
count
Cluster2




**Conclusión**

Para abordar esta tarea, apliqué dos métodos de agrupación que se adaptaron a mi conjunto de datos, que incluye tanto variables continuas como categóricas: K-Means (utilizando variables dummy) y K-Prototypes.

Utilicé dos métricas para determinar el número óptimo de clusters: la inercia y el índice de Calinski-Harabasz. Sin embargo, es importante recordar que, aunque podamos identificar un número 'óptimo' de clusters, siempre podremos ajustar este valor en función de los objetivos específicos de nuestro problema.