**Segmentación de Datos con Métodos Tradicionales y Autoencoder + KMeans**

**Asignatura:** Business Analytics & Big Data II / ING-SOF8-N  
**Programa y semestre:** Ingeniería de Software VIII

**Integrantes:** Ryan Hernandez – Felipe Ortiz – Kevin González  
**Docente:** Oscar Castiblanco

**Objetivos de la Actividad**

* Analizar un dataset real con variables numéricas y categóricas.
* Aplicar clustering tradicional (KMeans) y compararlo con un flujo avanzado basado en Autoencoder + KMeans.
* Evaluar la calidad de los clústeres mediante la métrica silhouette.
* Visualizar los resultados y perfilar los grupos encontrados.

**Metas del flujo**

* Transformar columnas numéricas y categóricas en una matriz numérica adecuada para redes neuronales.
* Entrenar un Autoencoder no supervisado y extraer un embedding por fila.
* Seleccionar el número de clústeres (k) óptimo mediante silhouette.
* Visualizar en 2D (PCA) y perfilar cada clúster (promedios y modas).

**Requisitos y materiales**

* Archivo de datos: Salary\_Data.csv (u otro CSV con columnas similares).
* Entorno: Python 3.x con paquetes: pandas, numpy, scikit-learn, torch (PyTorch), matplotlib (o plotly), scipy (opcional).
* Mantener estrictamente los scripts provistos en el documento original; no modificar los bloques de código.
* Semilla (random\_state) fija para reproducibilidad en KMeans y particionados.

**Etapa 1 Preparación del entorno**

* Importar correctamente las librerías y verificar que se encuentren instaladas.

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import torch

from torch import nn

from torch.utils.data import TensorDataset, DataLoader, random\_split

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import OneHotEncoder, StandardScaler

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

from sklearn.decomposition import PCA

**Etapa 2 Carga y exploración de los datos**

* Identificar las variables presentes.
* Revisar el tamaño del dataset y los tipos de datos.

df = pd.read\_csv("Salary\_Data.csv")

print(df.head())

print(df.info())

print(df.describe())

**Etapa 3 Exploración y visualización básica**

* Detectar valores faltantes.
* Comentar sobre las relaciones gráficas observadas.

print(df.isna().sum())

sns.scatterplot(x="Years of Experience", y="Salary", data=df)

plt.title("Experiencia vs Salario")

plt.show()

sns.scatterplot(x="Age", y="Salary", data=df)

plt.title("Edad vs Salario")

plt.show()

**Etapa 4 Preprocesamiento de datos**

* Justificar por qué es necesario imputar y escalar antes del clustering.

num\_cols = ["Age", "Years of Experience", "Salary"]

cat\_cols = ["Gender", "Education Level", "Job Title"]

num\_cols = [c for c in num\_cols if c in df.columns]

cat\_cols = [c for c in cat\_cols if c in df.columns]

num\_tf = Pipeline([

("imputer", SimpleImputer(strategy="median")),

("scaler", StandardScaler())

])

cat\_tf = Pipeline([

("imputer", SimpleImputer(strategy="most\_frequent")),

("onehot", OneHotEncoder(handle\_unknown="ignore"))

])

preprocess = ColumnTransformer([

("num", num\_tf, num\_cols),

("cat", cat\_tf, cat\_cols)

])

**Etapa 5 Clustering tradicional con KMeans**

* Seleccionar el número óptimo de clústeres basado en silhouette.

X = df[num\_cols + cat\_cols].copy()

inertias, sils, ks = [], [], range(2, 7)

for k in ks:

pipe = Pipeline([

("prep", preprocess),

("kmeans", KMeans(n\_clusters=k, random\_state=42, n\_init=10))

])

labels = pipe.fit\_predict(X)

inertias.append(pipe.named\_steps["kmeans"].inertia\_)

sils.append(silhouette\_score(pipe.named\_steps["prep"].transform(X), labels))

print("k\tInertia\t\tSilhouette")

for k, i, s in zip(ks, inertias, sils):

print(f"{k}\t{round(i,2)}\t\t{round(s,3)}")

chosen\_k = 3

final\_pipe = Pipeline([

("prep", preprocess),

("kmeans", KMeans(n\_clusters=chosen\_k, random\_state=42, n\_init=10))

])

df["Cluster"] = final\_pipe.fit\_predict(X)

print(df[["Years of Experience","Salary","Cluster"]].head())

**Etapa 6 Visualización y perfilado de clústeres**

* Interpretar el gráfico PCA.
* Describir las características principales de cada clúster.

Z = final\_pipe.named\_steps["prep"].transform(X)

pca = PCA(n\_components=2, random\_state=42)

Z2 = pca.fit\_transform(Z)

viz = pd.DataFrame(Z2, columns=["PC1","PC2"])

viz["Cluster"] = df["Cluster"]

sns.scatterplot(x="PC1", y="PC2", hue="Cluster", data=viz, palette="Set1", s=80)

plt.title("Clusters (PCA 2D)")

plt.show()

def moda\_serie(s):

return s.mode(dropna=True).iloc[0] if not s.mode(dropna=True).empty else np.nan

resumen\_num = df.groupby("Cluster")[num\_cols].mean().round(2)

resumen\_cat = df.groupby("Cluster")[cat\_cols].agg(moda\_serie)

perfil = resumen\_num.join(resumen\_cat, how="outer")

perfil["Conteo"] = df.groupby("Cluster").size()

print("\nPerfil de clusters:\n", perfil)

**Etapa 7 Segmentación avanzada con Autoencoder + KMeans**

* Definir Autoencoder
* Entrenar con early stopping
* Obtener embeddings latentes
* Selección de k con silhouette
* Visualización PCA
* Perfilado por clúster

import pandas as pd

import numpy as np

import torch

from torch import nn

from torch.utils.data import TensorDataset, DataLoader, random\_split

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import OneHotEncoder, StandardScaler

from sklearn.impute import SimpleImputer

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

df = pd.read\_csv("Salary\_Data.csv")

num\_cols = ["Age", "Years of Experience", "Salary"] # si NO quieres usar Salary, quítalo de esta lista

cat\_cols = ["Gender", "Education Level", "Job Title"]

num\_cols = [c for c in num\_cols if c in df.columns]

cat\_cols = [c for c in cat\_cols if c in df.columns]

X = df[num\_cols + cat\_cols].copy()

num\_tf = Pipeline([

("imputer", SimpleImputer(strategy="median")),

("scaler", StandardScaler())

])

cat\_tf = Pipeline([

("imputer", SimpleImputer(strategy="most\_frequent")),

("onehot", OneHotEncoder(handle\_unknown="ignore"))

])

preprocess = ColumnTransformer([

("num", num\_tf, num\_cols),

("cat", cat\_tf, cat\_cols)

])

X\_pre = preprocess.fit\_transform(X)

if hasattr(X\_pre, "toarray"):

X\_pre = X\_pre.toarray()

device = "cuda" if torch.cuda.is\_available() else "cpu"

X\_t = torch.tensor(X\_pre, dtype=torch.float32)

class SimpleAE(nn.Module):

def \_\_init\_\_(self, in\_dim, latent\_dim=8):

super().\_\_init\_\_()

self.encoder = nn.Sequential(

nn.Linear(in\_dim, 32),

nn.ReLU(),

nn.Linear(32, latent\_dim)

)

self.decoder = nn.Sequential(

nn.Linear(latent\_dim, 32),

nn.ReLU(),

nn.Linear(32, in\_dim)

)

def forward(self, x):

z = self.encoder(x)

x\_hat = self.decoder(z)

return x\_hat, z

def train\_autoencoder(X\_tensor, latent\_dim=8, epochs=200, batch\_size=64, lr=1e-3, patience=15):

in\_dim = X\_tensor.shape[1]

dataset = TensorDataset(X\_tensor)

n\_total = len(dataset)

n\_val = max(1, int(0.1 \* n\_total))

n\_train = n\_total - n\_val

train\_ds, val\_ds = random\_split(dataset, [n\_train, n\_val], generator=torch.Generator().manual\_seed(42))

train\_loader = DataLoader(train\_ds, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(val\_ds, batch\_size=batch\_size, shuffle=False)

model = SimpleAE(in\_dim, latent\_dim).to(device)

opt = torch.optim.Adam(model.parameters(), lr=lr)

crit = nn.MSELoss()

best\_val = float("inf")

wait = 0

best\_state = None

for ep in range(1, epochs+1):

model.train()

total = 0.0

for (xb,) in train\_loader:

xb = xb.to(device)

opt.zero\_grad()

x\_hat, \_ = model(xb)

loss = crit(x\_hat, xb)

loss.backward()

opt.step()

total += loss.item() \* xb.size(0)

train\_loss = total / len(train\_ds)

model.eval()

vtotal = 0.0

with torch.no\_grad():

for (xb,) in val\_loader:

xb = xb.to(device)

x\_hat, \_ = model(xb)

vloss = crit(x\_hat, xb)

vtotal += vloss.item() \* xb.size(0)

val\_loss = vtotal / len(val\_ds)

if ep % 10 == 0:

print(f"Epoch {ep:03d} | train {train\_loss:.4f} | val {val\_loss:.4f}")

if val\_loss < best\_val - 1e-6:

best\_val = val\_loss

wait = 0

best\_state = {k: v.cpu().clone() for k, v in model.state\_dict().items()}

else:

wait += 1

if wait >= patience:

print(f"Early stopping en epoch {ep} (mejor val {best\_val:.4f})")

break

if best\_state is not None:

model.load\_state\_dict(best\_state)

return model

model = train\_autoencoder(X\_t, latent\_dim=8, epochs=200, batch\_size=64, lr=1e-3, patience=15)

model.eval()

with torch.no\_grad():

Z = model.encoder(X\_t.to(device)).cpu().numpy()

def select\_k\_and\_cluster(Z, k\_min=2, k\_max=6, random\_state=42):

best\_k, best\_sil, best\_labels, best\_km = None, -1, None, None

for k in range(k\_min, k\_max+1):

km = KMeans(n\_clusters=k, random\_state=random\_state, n\_init=10)

labels = km.fit\_predict(Z)

s = silhouette\_score(Z, labels)

print(f"k={k} | silhouette={s:.3f}")

if s > best\_sil:

best\_k, best\_sil, best\_labels, best\_km = k, s, labels, km

return best\_k, best\_sil, best\_labels, best\_km

best\_k, best\_sil, labels\_nn, km = select\_k\_and\_cluster(Z, 2, 6)

print(f"Mejor k={best\_k} con silhouette={best\_sil:.3f}")

df["Cluster\_NN"] = labels\_nn

Z2 = PCA(n\_components=2, random\_state=42).fit\_transform(Z)

plt.figure()

plt.scatter(Z2[:, 0], Z2[:, 1], c=labels\_nn, s=50)

plt.title(f"Clusters en espacio latente (AE + KMeans, k={best\_k})")

plt.xlabel("PC1"); plt.ylabel("PC2")

plt.show()

def moda\_serie(s):

m = s.mode(dropna=True)

return m.iloc[0] if not m.empty else np.nan

resumen\_num = df.groupby("Cluster\_NN")[num\_cols].mean().round(2) if num\_cols else pd.DataFrame()

resumen\_cat = df.groupby("Cluster\_NN")[cat\_cols].agg(moda\_serie) if cat\_cols else pd.DataFrame()

perfil\_nn = resumen\_num.join(resumen\_cat, how="outer")

perfil\_nn["Conteo"] = df.groupby("Cluster\_NN").size()

print("\nPerfil (AE + KMeans):\n", perfil\_nn)