# Tarea 2 IA

Bruno Figueroa

# Introducción

1. La entrega debe realizarse en:
   - Google Colab
   - Jupyter Notebook
   - Archivo .py

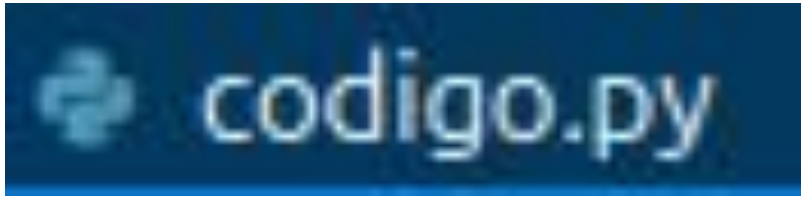
2. La entrega debe incluir un **video explicativo** con una duración máxima de **15 minutos**, acompañado de diapositivas.

4. El dataset debe contener al menos **10.000 filas** y **7 columnas utilizables**. La etiqueta $Y$ debe ser **discreta con más de 2 clases**.

# Google colab / Jupyter notebook / python

No se utilizo google colab, ya que no me deja importar archivos para la configuración (necesario en el punto 2).

Se utilizo python para crear el código inicialmente, luego se creo el archivo ipynb en júpiter notebook

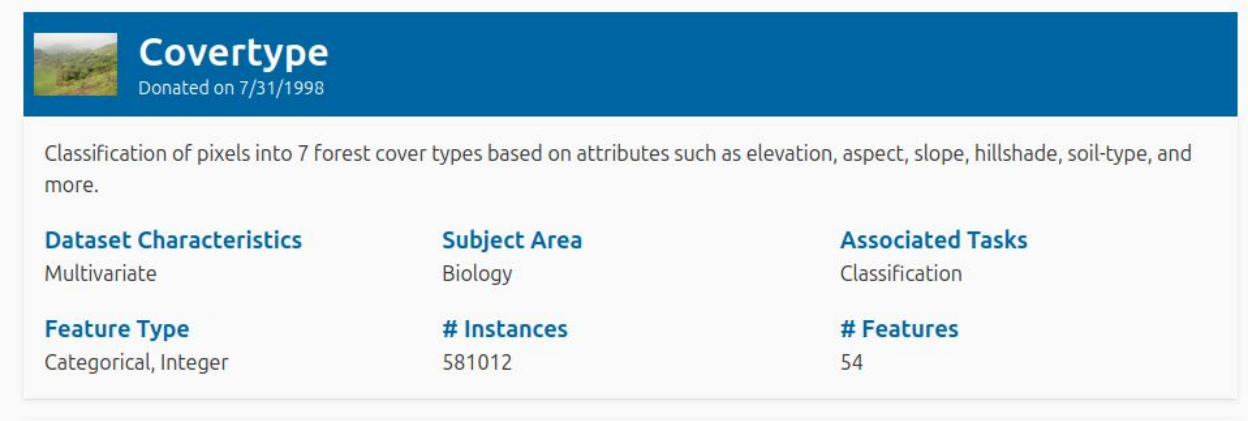



# Dataset utilizado “Covertype dataset”

https://archive.ics.uci.edu/dataset/31/covertype

580k filas (truncadas a 10k).

54 features, se utilizan 53 cómo features, y la última como etiqueta discreta de 7 clases.



# Codigo cargar dataset

```
data = fetch_covtype(as_frame=True)
df = data.frame
df = df.sample(n=10000, random_state=42)

print(df.shape)
print(df['Cover_Type'].value_counts())

X = df.drop(columns=['Cover_Type'])
y = df['Cover_Type']

results = []
```

```
(10000, 55)
Cover_Type
2    4841
1    3683
3     623
7     347
6     299
5     160
4      47
Name: count, dtype: int64
```

# Punto 1

Deberá ejecutar tres algoritmos de clustering: K-Means, K-Means++ y MeanShift, utilizando al menos cuatro configuraciones distintas para cada técnica (deberá justificar la elección de parámetros).

Para el entrenamiento, use únicamente el 80% de los datos, omitiendo la etiqueta Y.

A continuación, evalúe las doce configuraciones obtenidas mediante una métrica de su elección (por ejemplo, Silhouette Score) y seleccione las tres de mejor desempeño.

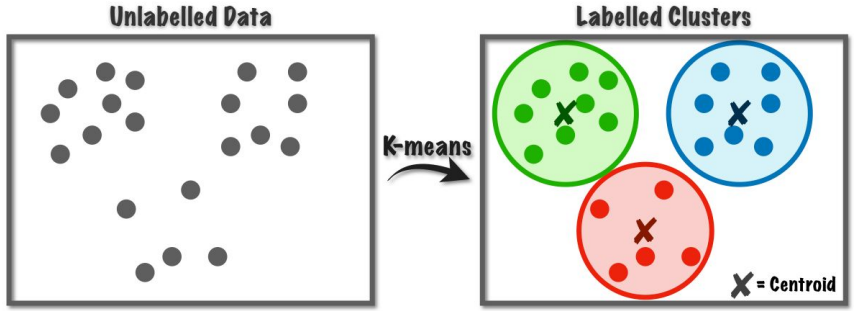
Luego, aplique cada una de estas configuraciones al 20% restante de los datos, asignando a cada muestra el cluster correspondiente (obtenidos desde el entrenamiento).

Finalmente, compare la etiqueta Y real de cada muestra con la etiqueta dominante dentro del cluster al que pertenece y analice si este procedimiento resulta razonable para asignar etiquetas faltantes.

# K-Means

K-means, posiciona n centroides aleatoriamente y se mueve hacia el centro de los datos.

Tiene como parámetros, la cantidad de centroides, la cantidad de iteraciones, y la cantidad de repeticiones del experimento, la métrica más importante es la cantidad de centroides.



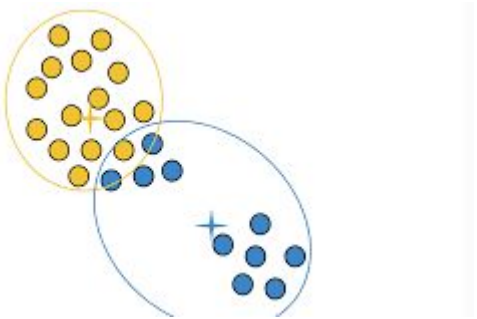
```python
print()
print("K-MEANS")

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = tts(
    X_scaled, y, test_size=0.2, random_state=42, stratify=y
)

kmeans_configs = [
    {"n_clusters": 3, "init": "random", "n_init": 10, "max_iter": 300},
    {"n_clusters": 5, "init": "random", "n_init": 15, "max_iter": 300},
    {"n_clusters": 7, "init": "random", "n_init": 20, "max_iter": 500},
    {"n_clusters": 9, "init": "random", "n_init": 25, "max_iter": 400},
]

for i, cfg in enumerate(kmeans_configs, 1):
    model = KMeans(**cfg, random_state=42)
    model.fit(X_train)
    labels = model.labels_

    sil_score = silhouette_score(X_train, labels)
    results.append({
        "model_type": "KMeans",
        "config": cfg,
        "silhouette": sil_score
    })
    print(f"Config {i}: {cfg}, Silhouette Score = {sil_score:.4f}")
```

```
K-MEANS
Config 1: {'n_clusters': 3, 'init': 'random', 'n_init': 10, 'max_iter': 300}, Silhouette Score = 0.1021
Config 2: {'n_clusters': 5, 'init': 'random', 'n_init': 15, 'max_iter': 300}, Silhouette Score = 0.0807
Config 3: {'n_clusters': 7, 'init': 'random', 'n_init': 20, 'max_iter': 500}, Silhouette Score = 0.1387
Config 4: {'n_clusters': 9, 'init': 'random', 'n_init': 25, 'max_iter': 400}, Silhouette Score = 0.1700
```

# K-Means++

Es lo mismo que K-means, pero se cambia “Random” por “k-means++” en la configuración, esto cambia la implementación de elegir 3 centroides aleatorios, a elegir el primer centroide aleatoriamente, y los demás, en base a la medición de los puntos al centroide inicial, asignando “mejores” centroides.

Ejemplo de malos centroides.



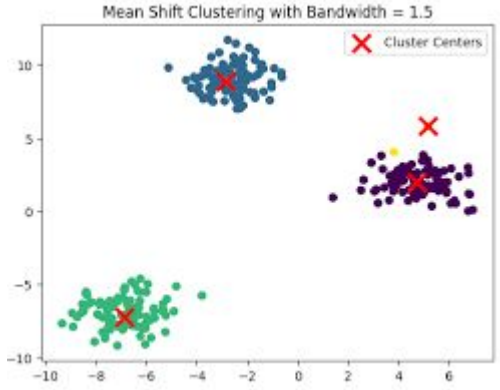
```python
print()
print("K-MEANS++")

kmeanspp_configs = [
    {"n_clusters": 3, "init": "k-means++", "n_init": 10, "max_iter": 300},
    {"n_clusters": 5, "init": "k-means++", "n_init": 20, "max_iter": 300},
    {"n_clusters": 7, "init": "k-means++", "n_init": 15, "max_iter": 500},
    {"n_clusters": 9, "init": "k-means++", "n_init": 25, "max_iter": 400},
]

for i, cfg in enumerate(kmeanspp_configs, 1):
    model = KMeans(**cfg, random_state=42)
    model.fit(X_train)
    labels = model.labels_
    sil_score = silhouette_score(X_train, labels)
    results.append({
        "model_type": "KMeans++",
        "config": cfg,
        "silhouette": sil_score
    })
    print(f"Config {i}: {cfg}, Silhouette Score = {sil_score:.4f}")
```

```
K-MEANS++
Config 1: {'n_clusters': 3, 'init': 'k-means++', 'n_init': 10, 'max_iter': 300}, Silhouette Score = 0.1021
Config 2: {'n_clusters': 5, 'init': 'k-means++', 'n_init': 20, 'max_iter': 300}, Silhouette Score = 0.1154
Config 3: {'n_clusters': 7, 'init': 'k-means++', 'n_init': 15, 'max_iter': 500}, Silhouette Score = 0.0970
Config 4: {'n_clusters': 9, 'init': 'k-means++', 'n_init': 25, 'max_iter': 400}, Silhouette Score = 0.1159
```

# MeanShift

Mean shift, a diferencia de kmeans, llena todo el espacio con centroides, y los fusiona cuando están suficientemente cerca.

El parámetro a modificar en este caso es el bandwith, que es la distancia para que se fusionen los centroides.



```python
print()
print("MeanShift")

bandwidth_estimate = estimate_bandwidth(X_train, quantile=0.2, n_samples=500)
print(f"Bandwidth estimado base: {bandwidth_estimate:.4f}")

meanshift_configs = [
    {"bandwidth": bandwidth_estimate * 0.5},
    {"bandwidth": bandwidth_estimate},
    {"bandwidth": bandwidth_estimate * 1.5},
    {"bandwidth": bandwidth_estimate * 2.0},
]

for i, cfg in enumerate(meanshift_configs, 1):
    model = MeanShift(**cfg)
    model.fit(X_train)
    labels = model.labels_

    if len(np.unique(labels)) > 1:
        sil_score = silhouette_score(X_train, labels)
    else:
        sil_score = -1

    results.append({
        "model_type": "MeanShift",
        "config": cfg,
        "silhouette": sil_score
    })
    print(f"Config {i}: {cfg}, Silhouette Score = {sil_score:.4f}, Clusters encontrados = {len(np.unique(labels))}")
```

```
MeanShift
Bandwidth estimado base: 7.5653
Config 1: {'bandwidth': np.float64(3.782648713514269)}, Silhouette Score = 0.3996, Clusters encontrados = 84
Config 2: {'bandwidth': np.float64(7.565297427028538)}, Silhouette Score = 0.3058, Clusters encontrados = 26
Config 3: {'bandwidth': np.float64(11.347946140542806)}, Silhouette Score = 0.4296, Clusters encontrados = 21
Config 4: {'bandwidth': np.float64(15.130594854057076)}, Silhouette Score = 0.5170, Clusters encontrados = 16
```

# Justificación de parámetros.

Kmeans y Kmeans++:

n_clusters: 3 5 7 9 -> cantidades no muy grandes, debido a la pequeña cantidad de clases de etiqueta.

n_init y max iter -> solo reducen el caos, y mejoran la convergencia, valores muy altos no ayudan en nada, y valores muy bajos hacen que el proceso no funcione, estos rondan los 300-500 en los parámetros utilizados.

Meanshift:

bandwith -> se calcula el quantil 0.2, y se usa como base,
Utilizando este valor con diferentes factores: 0.5, 1, 1.5 y 2.

# Evaluación global de las 12 configuraciones

Se toman los 12 modelos entrenados y se ordenan en base a su puntaje de silhouette, obteniendo así un top 3.

Luego se comparan estos 3 mejores modelos, con el resto de datos (20%), para obtener la precisión de cada modelo.

```python
results = sorted(results, key=lambda x: x["silhouette"], reverse=True)
top3 = results[:3]

print()
print("Top 3 configuraciones globales por Silhouette Score")
for res in top3:
    print(res)

for i, res in enumerate(top3, 1):
    algo = res["model_type"]
    cfg = res["config"]
    print(f"\nModelo {i}: {algo}, Config: {cfg}")

    if algo in ["KMeans", "KMeans++"]:
        model = KMeans(**cfg, random_state=42)
    else:
        model = MeanShift(**cfg)

    model.fit(X_train)
    test_clusters = model.predict(X_test)

    train_clusters = model.labels_
    cluster_labels = {}
    for cluster_id in np.unique(train_clusters):
        mask = train_clusters == cluster_id
        dominant_label = y_train.iloc[mask].mode()[0]
        cluster_labels[cluster_id] = dominant_label

    y_pred = [cluster_labels[c] for c in test_clusters if c in cluster_labels]
    valid_idx = [i for i, c in enumerate(test_clusters) if c in cluster_labels]
    match_ratio = np.mean(np.array(y_pred) == y_test.values[valid_idx])
    print()
    print(f"Coincidencia entre etiquetas reales y dominantes = {match_ratio:.4f}")
```

```
Top 3 configuraciones globales por Silhouette Score
{'model_type': 'MeanShift', 'config': {'bandwidth': np.float64(15.130594854057076)}, 'silhouette': 0.5170171193037753}
{'model_type': 'MeanShift', 'config': {'bandwidth': np.float64(11.347946140542806)}, 'silhouette': 0.4296315428885252}
{'model_type': 'MeanShift', 'config': {'bandwidth': np.float64(3.782648713514269)}, 'silhouette': 0.3996245643432745}

Modelo 1: MeanShift, Config: {'bandwidth': np.float64(15.130594854057076)}

Coincidencia entre etiquetas reales y dominantes = 0.4920

Modelo 2: MeanShift, Config: {'bandwidth': np.float64(11.347946140542806)}

Coincidencia entre etiquetas reales y dominantes = 0.4960

Modelo 3: MeanShift, Config: {'bandwidth': np.float64(3.782648713514269)}

Coincidencia entre etiquetas reales y dominantes = 0.6350
```

# Análisis de resultados

Estos datos, demuestran cómo el modelo está funcionando correctamente, siendo capaz de predecir hasta cierto punto el resultado, llegando hasta casi un 50/50 en el TOP1 y TOP2, y con un 64% en el mejor caso.

Teniendo en cuenta que la etiqueta tiene 7 clases, estos modelos permiten pasar de un 14% de éxito (selección aleatoria) a un 49% o 64%, según el modelo utilizado.

Ahora, respondiendo la pregunta ¿Este procedimiento resulta razonable para asignar etiquetas faltantes?

La respuesta es un depende, ya que en el mejor de los casos, se obtuvo solo un rendimiento del 64%, lo que no se puede considerar demasiado confiable, por ende, se debería buscar un método algo más exacto (rondando el 85+% de aciertos) para que sea razonable de aplicar, en caso de no tener disponible otro método, si es razonable el utilizarlo, ya que al ser un 64%, el modelo permite predecir la mayoría de situaciones, en vez de un modelo puramente aleatorio con un 14%.

# Punto 2

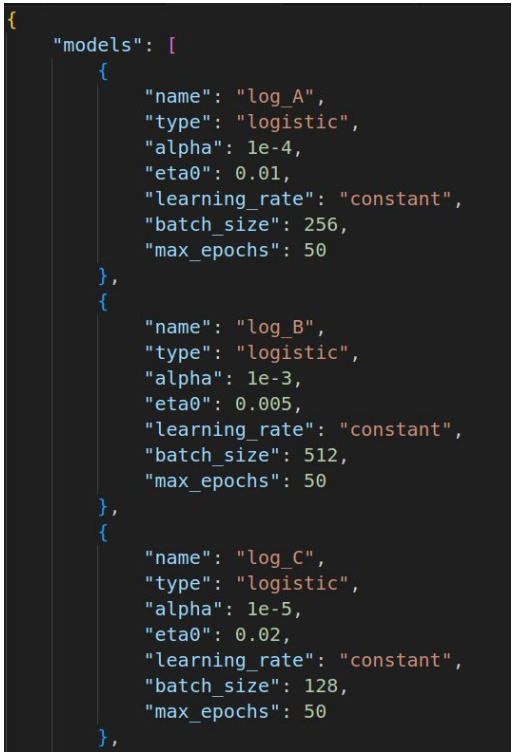
Utilizando el mismo conjunto de datos previamente seleccionado, diseñe e implemente una técnica que permita entrenar en paralelo múltiples instancias (al menos tres por técnica) de Regresión Logística y SVM, variando sus hiperparámetros (por ejemplo: batch size, tasa de aprendizaje, etc, según sea el caso).

Los parámetros de cada configuración deberán definirse en un archivo de configuración externo, y el entrenamiento deberá realizarse utilizando el 80% de los datos.

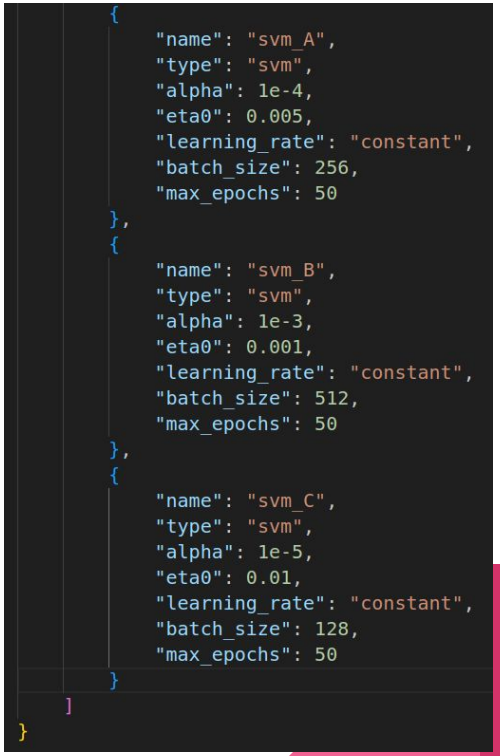
Cada modelo se evaluará periódicamente (solo con datos de entrenamiento), y por cada cinco épocas deberá descartarse la configuración con peor desempeño entre todas las configuraciones restantes. Para las dos mejores configuraciones, presente métricas de evaluación utilizando el conjunto de testing, es decir, el 20% de los datos.

Analice los resultados obtenidos en función de los hiperparámetros seleccionados.

# Archivo config.json



```
{
    "models": [
        {
            "name": "log_A",
            "type": "logistic",
            "alpha": 1e-4,
            "eta0": 0.01,
            "learning_rate": "constant",
            "batch_size": 256,
            "max_epochs": 50
        },
        {
            "name": "log_B",
            "type": "logistic",
            "alpha": 1e-3,
            "eta0": 0.005,
            "learning_rate": "constant",
            "batch_size": 512,
            "max_epochs": 50
        },
        {
            "name": "log_C",
            "type": "logistic",
            "alpha": 1e-5,
            "eta0": 0.02,
            "learning_rate": "constant",
            "batch_size": 128,
            "max_epochs": 50
        },
```



```
        {
            "name": "svm_A",
            "type": "svm",
            "alpha": 1e-4,
            "eta0": 0.005,
            "learning_rate": "constant",
            "batch_size": 256,
            "max_epochs": 50
        },
        {
            "name": "svm_B",
            "type": "svm",
            "alpha": 1e-3,
            "eta0": 0.001,
            "learning_rate": "constant",
            "batch_size": 512,
            "max_epochs": 50
        },
        {
            "name": "svm_C",
            "type": "svm",
            "alpha": 1e-5,
            "eta0": 0.01,
            "learning_rate": "constant",
            "batch_size": 128,
            "max_epochs": 50
        }
    ]
}
```

# Cargar configuración

Cargar configuracion desde archivo externo (configs.json) tiene que estar en el mismo directorio que el .ipynb

```
with open("configs.json", "r") as f:
    configs = json.load(f)["models"]

print(f"Configuraciones cargadas: {len(configs)} modelos\n")
```

```
Configuraciones cargadas: 6 modelos
```

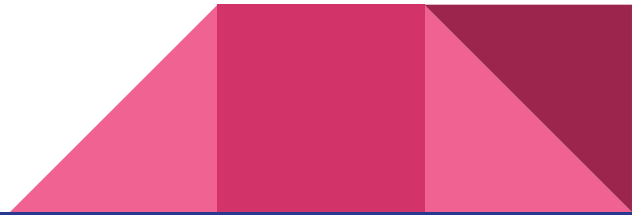
# Procedimiento

Se subdivide el dataset para poder entrenar las diferentes configuraciones.

Luego, se crean los modelos con los parámetros de la configuración.

Se inicializan los parámetros (pesos) para las funciones de regresión.

Finalmente, se entrenan todos en paralelo, asignándoles valores, y eliminando al peor en cada ronda, hasta que solo queden 2.



Subdividir parte del train para evaluacion periodica, y creacion de los modelos.

```
X_train_main, X_train_eval, y_train_main, y_train_eval = tts(
    X_train, y_train, test_size=0.1, random_state=42, stratify=y_train
)
classes = np.unique(y_train)


models_meta = []
for cfg in configs:
    loss = "log_loss" if cfg["type"] == "logistic" else "hinge"
    clf = SGDClassifier(
        loss=loss,
        penalty="l2",
        alpha=cfg["alpha"],
        learning_rate=cfg["learning_rate"],
        eta0=cfg["eta0"],
        random_state=42
    )
    models_meta.append({
        "name": cfg["name"],
        "type": cfg["type"],
        "cfg": cfg,
        "clf": clf,
        "epochs_done": 0,
        "max_epochs": cfg["max_epochs"],
        "batch_size": cfg["batch_size"],
        "alive": True,
        "last_eval_acc": None
    })
```

Inicializar pesos, y definir la funcion de entrenamiento por bloque.

```
for m in models_meta:
    init_batch = min(100, X_train_main.shape[0])
    m["clf"].partial_fit(X_train_main[:init_batch], y_train_main[:init_batch], classes=classes)


def train_chunk(model, X_main, y_main, epochs_chunk, batch_size, classes):
    n = X_main.shape[0]
    for ep in range(epochs_chunk):
        X_sh, y_sh = skshuffle(X_main, y_main, random_state=int(time.time() * 1000) % 2**32)
        for start in range(0, n, batch_size):
            end = min(start + batch_size, n)
            Xb, yb = X_sh[start:end], y_sh[start:end]
            model.partial_fit(Xb, yb, classes=classes)
    y_pred_eval = model.predict(X_train_eval)
    acc = accuracy_score(y_train_eval, y_pred_eval)
    return model, acc
```

Entrenamiento iterativo en paralelo con eliminacion del peor cada 5 epocas.

```
epochs_chunk = 5
max_rounds = max(m["max_epochs"] for m in models_meta) // epochs_chunk + 1
print(f"Iniciando entrenamiento paralelo ({len(models_meta)} configuraciones, {epochs_chunk} épocas por ronda)\n")

for round_i in range(max_rounds):
    alive = [m for m in models_meta if m["alive"] and m["epochs_done"] < m["max_epochs"]]
    if len(alive) <= 2:
        print("Menos de 3 modelos activos, deteniendo eliminaciones.")
        break

    print(f"--- Ronda {round_i+1} | modelos activos: {len(alive)} ---")
    futures = {}
    with ThreadPoolExecutor(max_workers=min(len(alive), 4)) as exe:
        for m in alive:
            futures[exe.submit(train_chunk, m["clf"], X_train_main, y_train_main, epochs_chunk, m["batch_size"], classes)] = m
        for fut in as_completed(futures):
            m = futures[fut]
            clf_updated, acc = fut.result()
            m["clf"] = clf_updated
            m["epochs_done"] += epochs_chunk
            m["last_eval_acc"] = acc
            print(f"{m['name']} ({m['type']}) -> acc_train_eval={acc:.4f}")

    # Eliminar el peor
    alive = [m for m in models_meta if m["alive"]]
    if len(alive) <= 2:
        break
    worst = min(alive, key=lambda x: x["last_eval_acc"] or -1.0)
    worst["alive"] = False
    print(f"Eliminado: {worst['name']} ({worst['last_eval_acc']:.4f})\n")

# Seleccionar ganadores
finalists = sorted([m for m in models_meta if m["alive"]], key=lambda x: x["last_eval_acc"] or 0.0, reverse=True)[:2]
print("\nFinalistas:")
for f in finalists:
    print(f"- {f['name']} ({f['type']}) acc_train_eval={f['last_eval_acc']:.4f}")
```

# Resultados

```
Iniciando entrenamiento paralelo (6 configuraciones, 5 épocas por ronda)

--- Ronda 1 | modelos activos: 6 ---
log_B (logistic) -> acc_train_eval=0.7050
svm_A (svm) -> acc_train_eval=0.6987
log_A (logistic) -> acc_train_eval=0.7075
svm_B (svm) -> acc_train_eval=0.6963
log_C (logistic) -> acc_train_eval=0.6900
svm_C (svm) -> acc_train_eval=0.6837
Eliminado: svm_C (0.6837)

--- Ronda 2 | modelos activos: 5 ---
log_B (logistic) -> acc_train_eval=0.7000
svm_A (svm) -> acc_train_eval=0.6975
log_A (logistic) -> acc_train_eval=0.7087
svm_B (svm) -> acc_train_eval=0.7163
log_C (logistic) -> acc_train_eval=0.6913
Eliminado: log_C (0.6913)
```

```
--- Ronda 3 | modelos activos: 4 ---
log_B (logistic) -> acc_train_eval=0.7063
svm_B (svm) -> acc_train_eval=0.7087
log_A (logistic) -> acc_train_eval=0.7087
svm_A (svm) -> acc_train_eval=0.7137
Eliminado: log_B (0.7063)

--- Ronda 4 | modelos activos: 3 ---
log_A (logistic) -> acc_train_eval=0.6975
svm_A (svm) -> acc_train_eval=0.7087
svm_B (svm) -> acc_train_eval=0.7025
Eliminado: log_A (0.6975)

Menos de 3 modelos activos, deteniendo eliminaciones.

Finalistas:
- svm_A (svm) acc_train_eval=0.7087
- svm_B (svm) acc_train_eval=0.7025
```

Ganadores: SVM_A y SVM_B

# Evaluación de los ganadores

Teniendo los 2 finalistas, se evalúan utilizando el resto de los datos (20%), para obtener diferentes parámetros, como el accuracy, y el f1 score.

Evaluar finalistas, utilizando los datos restantes (20%)

```
results_summary = []
for f in finalists:
    clf = f["clf"]
    y_pred = clf.predict(X_test)
    acc_test = accuracy_score(y_test, y_pred)
    print(f"\n== {f['name']} ({f['type']}) ==")
    print(f"Accuracy test: {acc_test:.4f}")
    print(classification_report(y_test, y_pred, digits=4, zero_division=0))
    results_summary.append({
        "name": f["name"],
        "type": f["type"],
        "train_eval_acc": f["last_eval_acc"],
        "test_acc": acc_test
    })

results_summary
```

```
== svm_A (svm) ==
Accuracy test: 0.7040
              precision    recall  f1-score   support

           1     0.7246    0.6282    0.6730       737
           2     0.7126    0.8326    0.7680       968
           3     0.6203    0.7840    0.6926       125
           4     0.0000    0.0000    0.0000         9
           5     0.0000    0.0000    0.0000        32
           6     0.2222    0.0667    0.1026        60
           7     0.7400    0.5362    0.6218        69

    accuracy                         0.7040      2000
   macro avg     0.4314    0.4068    0.4083      2000
weighted avg     0.6829    0.7040    0.6875      2000


== svm_B (svm) ==
Accuracy test: 0.7145
              precision    recall  f1-score   support

           1     0.7064    0.6988    0.7026       737
           2     0.7400    0.8058    0.7715       968
           3     0.6102    0.8640    0.7152       125
           4     0.0000    0.0000    0.0000         9
           5     0.0000    0.0000    0.0000        32
           6     0.0000    0.0000    0.0000        60
           7     0.6842    0.3768    0.4860        69

    accuracy                         0.7145      2000
   macro avg     0.3916    0.3922    0.3822      2000
weighted avg     0.6802    0.7145    0.6938      2000
```

```
[{'name': 'svm_A',
  'type': 'svm',
  'train_eval_acc': 0.70875,
  'test_acc': 0.704},
 {'name': 'svm_B',
  'type': 'svm',
  'train_eval_acc': 0.7025,
  'test_acc': 0.7145}]
```

# Análisis de resultados

Los 2 mejores modelos de regresión fueron: smv_A y svm_B, ambos llegaron a un desempeño final muy similar de 70.4% y 71.4% de aciertos sobre el conjunto de prueba.

El modelo SVM_B logró un rendimiento ligeramente superior, especialmente la clase 2, que es la más representada en el conjunto de datos. Ambos modelos lograron identificar correctamente las clases más frecuentes (1, 2, 3), pero fallaron con las clases menos presentes (4, 5, 6), donde la cantidad de ejemplos disponibles fue demasiado baja para que se pudiera identificar algún patrón predictivo, las 3 mejores clases juntas, conforman alrededor de un 5% de los datos totales, lo que muestra un claro desbalance en el dataset utilizado.

El modelo SVM_A, aunque con una accuracy ligeramente menor, mostró en algunos casos, un puntaje f1 superior, lo que sugiere una convergencia más estable durante el entrenamiento para esas clases.

En resumen, los resultados fueron coherentes con respecto a los datos utilizados y los parámetros seleccionados, donde este fue incapaz de converger a una solución para todas las clases de la etiqueta, debido a el desbalance del dataset utilizado, lo que muestra la importancia de aplicar técnicas de balanceo de datos o ponderación por clase en casos futuros.

# Uso de inteligencia artificial

Es necesario mencionar que para esta tarea se utilizó chat gpt para la creación del código inicial, luego este fue debuggeando a mano.

Ningún área del análisis fue generado utilizando dicha herramienta.