TAREA

IRIS KMeans

Algoritmos Avanzados

Alumnos	Quijada Castillo Juan Diego
	Perez Gonzalez Sebastian
Maestra	Torres Peralta Raquel
Fecha	21 de Febrero de 2026

1. Análisis de los datos y selección de atributos

Para el problema de encontrar los atributos, mejor usamos un enfoque automatizado, en el cual, en vez de seleccionar las features manualmente, el algoritmo divide las 4 características del dataset (longitud y ancho del sépalo, longitud y ancho del pétalo) en todos los pares posibles. El algoritmo de K-Means se ejecuta para cada par de features y evalúa su rendimiento. La representación de datos final que se selecciona como "ganadora" es aquella que maximiza la métrica de F1-Score, lo que garantiza que las muestras sean lo más diferenciables posibles sin sacrificar la eficacia del modelo.

Para hacer esto, se implementó una función que utiliza la librería itertools para generar dinámicamente todas las combinaciones posibles de dos dimensiones:

```
        def dividir_lista_en_pares(lista):
            """
            Genera todas las combinaciones posibles de 2 features (columnas)
            de la lista de puntos.
            """
            if not lista:
                return [], []

            num_elementos = len(lista[0])
            # Obtiene todas las combinaciones de 2 índices entre las columnas
        disponibles
            indices = list(itertools.combinations(range(num_elementos), 2))

            # Guarda cada punto del dataset a 2 dimensiones
            pares = [[[fila[i], fila[j]] for fila in lista] for i, j in indices]

            return pares, indices
```

2. Aplicación del algoritmo K-Means

El algoritmo de K-Means fue implementado desde cero utilizando Python Vanilla para la lógica matemática, sin depender de librerías externas, aunque se usa la librería interna Itertools para el algoritmo. Para mandar los datos a la página, se usa Flask, una librería externa. En esta página se hace uso de HTML, TailwindCSS y ChartJS para la interfaz gráfica y visualización de los datos.

El algoritmo inicializa los centroides de forma aleatoria (implementando la lógica de K-Means++ para una mejor distribución) basándose en los puntos existentes. Luego, asigna cada punto al centroide más cercano utilizando la distancia euclidiana al cuadrado, y recalcula la posición de los centroides iterativamente hasta que estos ya no cambian de posición, o se alcanza el número máximo de iteraciones.

Este proceso iterativo es lo mas importante del algoritmo y se ejecuta para cada par de características generadas:

```
# Inicializa centroides con K-Means++
clusters = inicializar_centroides(puntos_plot_par, k)
lista_asignaciones = []

# Itera hasta que no cambien los centroides o hasta alcanzar el límite de
iteraciones
for veces in itertools.count(1):
    # Asigna cada punto al centroide más cercano
    lista_asignaciones = sacar_distancias(puntos_plot_par, clusters)
    lista_dividida = dividir_en_valor(lista_asignaciones)

    # Recalcula la posición de los centroides
    centros_dict = calcular_centros(puntos_plot_par, lista_dividida, k)
    clusters_new = [centros_dict[i] for i in sorted(centros_dict.keys())]

    if veces >= max_veces:
        break  # Se alcanzó el límite de iteraciones
    elif clusters_new != clusters:
        clusters = clusters_new  # Los centroides cambiaron, continuar
    else:
        break  # Los centroides no cambiaron, fin del algoritmo
```

3. Correspondencia entre clusters y clases

Debido a que K-Means es un algoritmo de aprendizaje no supervisado, este asigna etiquetas numéricas arbitrarias a los clusteres (Ejemplo: 0, 1, 2), entonces para encontrar la correspondencia entre las clases reales (Setosa, Versicolor, Virginica) y los clusteres creados por el algoritmo, se implementó una función que evalúa todas las permutaciones posibles de los clusteres generados contra las clases reales, seleccionando la permutación que produce más número de coincidencias exactas, traduciendo las clases numéricas a sus clases correspondientes.

La búsqueda de la correspondencia óptima se realiza en este bloque de código:

```
# Prueba todas las permutaciones posibles de etiquetas para encontrar el
mejor mapeo
for permutacion in itertools.permutations(clases_unicas):
    mapeo = dict(zip(clusters_encontrados, permutacion))
    coincidencias = sum(
        1
        for asignado, real in zip(asignaciones, clases_reales)
        if mapeo.get(asignado) == real
    )
    if coincidencias > max_coincidencias:
        max_coincidencias = coincidencias
        mejor_mapeo = mapeo

# Traduce las asignaciones del cluster al nombre de clase real según el
mejor mapeo
asignaciones_traducidas = [mejor_mapeo.get(a, -1) for a in asignaciones]
```

4. Cálculo de aciertos y métricas

El código calcula el rendimiento global y por clase, por medio de comparar las predicciones (traducidas anteriormente) contra las clases reales del dataset. Para cada clase, determina los True Positive (TP), False Positives (FP) y los Falsos Negativos (FN), y usando estos valores, el código calcula la precisión, Recall y F1-Score. Además, calcula la cantidad de aciertos totales, el porcentaje de muestras clasificadas correctamente sobre el total del dataset.

Esto se ve en el cálculo dentro de la función de evaluación:

```
        # Calcula métricas por clase: Precision, Recall y F1-Score
        metricas_por_clase = {}
        for clase in clases_unicas:
            TP = sum(1 for a, r in zip(asignaciones_traducidas, clases_reales) if a
        == clase and r == clase)
            FP = sum(1 for a, r in zip(asignaciones_traducidas, clases_reales) if a
        == clase and r != clase)
            FN = sum(1 for a, r in zip(asignaciones_traducidas, clases_reales) if a
        != clase and r == clase)

            # Calcula métricas evitando divisiones por cero
            precision_clase = TP / (TP + FP) if (TP + FP) > 0 else 0
            recall_clase = TP / (TP + FN) if (TP + FN) > 0 else 0
            f1_clase = (
                2 * (precision_clase * recall_clase) / (precision_clase +
        recall_clase)
                if (precision_clase + recall_clase) > 0
                else 0
            )
```

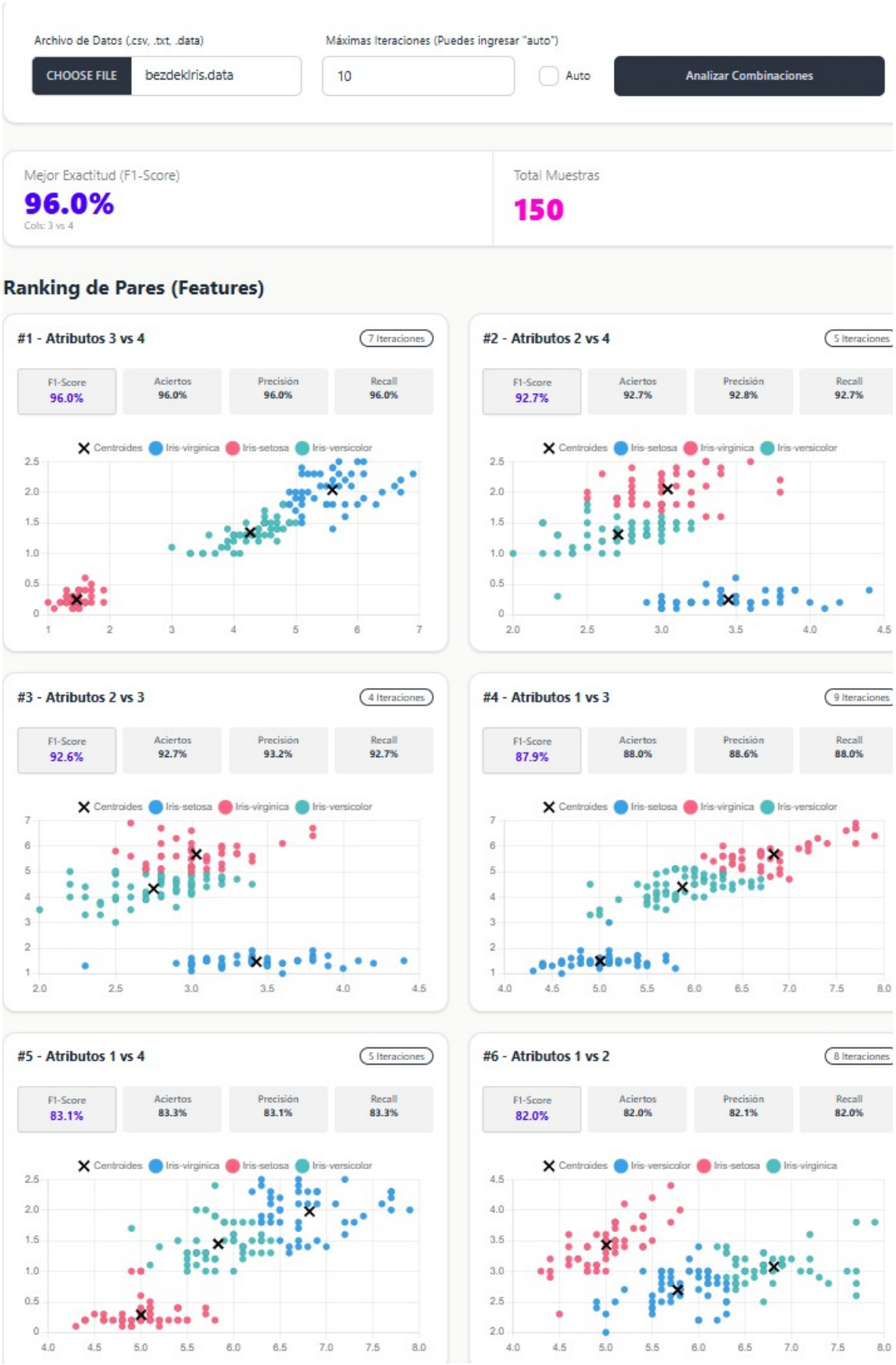
5. Gráfica de los clusters y las predicciones

En lugar de utilizar MatLab, se usa ChartJS para generar gráficas interactivas en el navegador. Con ChartJS se generan gráficas de dispersión (Scatter Plot) para cada par de features del dataset. En la gráfica se puede observar los puntos de cada par de features coloreados según el cluster en donde quedaron más pegados a un centroide, así como marcadores distintivos (Cruces negras) que representan la posición final de los centroides calculados.

En el Frontend, esta visualización se configura estructurando dinámicamente los datasets y pasándolos a la instancia de Chart.js:

```
        // Crea la gráfica de dispersión con Chart.js
        new Chart(ctx, {
            type: 'scatter',
            data: {
                datasets: puntos // Incluye los puntos clasificados y los
        centroides formateados
            },
            options: {
```

```
                responsive: true,
                scales: {
                    x: {
                        type: 'linear',
                        position: 'bottom'
                    }
                },
                plugins: {
                    legend: {
                        position: 'top',
                        labels: {
                            usePointStyle: true,
                        }
                    }
                }
            }
        });
```


Archivo de Datos (.csv, .txt, .data)
CHOOSE FILE
bezdekIris.data
Máximas Iteraciones (Puedes ingresar "auto")
10
Auto
Analizar Combinaciones
Mejor Exactitud (F1-Score)
96.0%
Cols: 3 vs 4
Total Muestras
150
Ranking de Pares (Features)
#1 - Atributos 3 vs 4
7 Iteraciones
F1-Score 96.0%
Aciertos 96.0%
Precisión 96.0%
Recall 96.0%
Centroides Iris-virginica Iris-setosa Iris-versicolor
#2 - Atributos 2 vs 4
5 Iteraciones
F1-Score 92.7%
Aciertos 92.7%
Precisión 92.8%
Recall 92.7%
Centroides Iris-setosa Iris-virginica Iris-versicolor
#3 - Atributos 2 vs 3
4 Iteraciones
F1-Score 92.6%
Aciertos 92.7%
Precisión 93.2%
Recall 92.7%
Centroides Iris-setosa Iris-virginica Iris-versicolor
#4 - Atributos 1 vs 3
9 Iteraciones
F1-Score 87.9%
Aciertos 88.0%
Precisión 88.6%
Recall 88.0%
Centroides Iris-setosa Iris-virginica Iris-versicolor
#5 - Atributos 1 vs 4
5 Iteraciones
F1-Score 83.1%
Aciertos 83.3%
Precisión 83.1%
Recall 83.3%
Centroides Iris-virginica Iris-setosa Iris-versicolor
#6 - Atributos 1 vs 2
8 Iteraciones
F1-Score 82.0%
Aciertos 82.0%
Precisión 82.1%
Recall 82.0%
Centroides Iris-versicolor Iris-setosa Iris-virginica

6. Conclusión

¿Era lo esperado? ¿Por qué se obtuvo este resultado?

Sí, los resultados obtenidos son los que esperábamos para el dataset de Iris utilizando K-Means, esto ya que se puede ver que la clase Iris Setosa es linealmente separable de las otras clases, por lo que al final el algoritmo siempre logra agruparla con una precisión cercana a 100%, pero como Iris Versicolor e Iris Virginica tiene una pequeña superposición en sus medidas, provoca que K-Means tenga un pequeño margen de error al intentar separar estos grupos. El resultado conseguido por el mejor resultado demuestra que usar las medidas relacionadas al pétalo (longitud y ancho) da una separación más clara que las medidas del sépalo, lo cual en parte justifica el evaluar todas las combinaciones de atributos, ya que no conocíamos eso de antemano.