Comparación de Métodos

- `QDA`
- `TensorizedQDA`
- `fasterQDA` (matriz $n \times n$)

Método	`_predict_one`	`predict`
`QDA`	`for` por clase	`for` por obs.
`TensorizedQDA`	una pasada por obs.	`for` por obs.
`fasterQDA`	una pasada	una pasada

Table 1: Resumen de las diferencias entre `QDA`, `TensorizedQDA` y `fasterQDA`.

Comparación de Dimensiones

X_unbiased $(\mathbf{x}-\mu_j)$: Vector de observaciones sin promedios.

Cov_inv $(\boldsymbol{\Sigma}_{\mathbf{j}}^{-\mathbf{1}})$: Matriz de covarianza invertida.

Prod_int: $(\mathbf{x}-\mu_j)^T\boldsymbol{\Sigma}_{\mathbf{j}}^{-\mathbf{1}}(\mathbf{x}-\mu_j)$.

Log_vero: $\frac{1}{2}\log\left|\Sigma_j^{-1}\right| - \frac{1}{2}(\mathbf{x}-\mu_j)^T\boldsymbol{\Sigma}_{\mathbf{j}}^{-\mathbf{1}}(\mathbf{x}-\mu_j)$.

Método	X_unbiased	Cov_inv	Prod_int	Log_vero
`QDA`	4×1	4×4	1	1
`TensorizedQDA`	$3\times 4\times 1$	$3\times 4\times 4$	3×1	3×1
`fasterQDA`	$3\times 4\times n$	$3\times 4\times 4$	$3\times n\times n$	$3\times n$

Table 2: Dimensiones de las variables principales por método.

- `np.diagonal()` convierte `Prod_int` de $3\times n\times n$ a $3\times n$ en `Log_vero`.

Comparación de Tiempos: QDA, TensorizedQDA y FasterQDA

- Set Test, split de 0.3 ($n = 45$).

Método	Tiempo (s)	Desviación estándar (s)
`QDA`	0.00242	0.00055
`TensorizedQDA`	0.00062	0.00051
`fasterQDA`	0.00010	0.00030

Table 3: Tiempos de ejecución de los métodos QDA, TensorizedQDA y FasterQDA.

- `TensorizedQDA` speedup x4 sobre `QDA`
- `fasterQDA` speedup x6 sobre `TensorizedQDA`

Comparación de Tiempos: fasterQDA vs. FasterQDA

- `fasterQDA` (matriz $n \times n$) vs. `FasterQDA`
- Usando $\text{diag}(A \cdot B) = \sum_{\text{col}}(A \odot B^T) = \text{np.sum}(A \odot B^T, \text{axis} = 1)$ es posible acelerar el método `fasterQDA`, evitando pasar por la matrices de $n \times n$.
- Set Test, split de 0.3 ($n = 45$).

Método	Tiempo (s)	Desviación estándar (s)
`fasterQDA`	0.00010	0.00030
`FasterQDA`	0.00008	0.00027

Table 4: Tiempos de ejecución de los fasterQDA y FasterQDA.

- No se ve un speedup considerable!

Diferencias en el código: fasterQDA vs. FasterQDA

- Diferencias en los códigos:

fasterQDA

```
unbiased_X_transposed =  unbiased_X.transpose(0,2,1) # Forma (classes, n, p)
inner_prod_mat = unbiased_X_transposed @ self.tensor_inv_cov @ unbiased_X # Forma
     (classes, n, n)
```

Costo matemático: $classes \times (n \times p^2 + n^2 \times p)$

FasterQDA

```
unbiased_X_transposed =  unbiased_X.transpose(0,2,1) # Forma (classes, n, p)
A = unbiased_X_transposed @ self.tensor_inv_cov # Forma (classes, n, p)
inner_prod = (A * unbiased_X_transposed).sum(axis=-1)  # Forma (classes, n)
```

Costo matemático: $classes \times (n \times p^2 + n \times p)$

Relación en costos matemáticos entre fasterQDA y FasterQDA

- Relación de costos matemático `fasterQDA` y `FasterQDA` para el producto interno:

$$\frac{classes \times (n \times p^2 + n^2 \times p)}{classes \times (n \times p^2 + n \times p)}$$

- Aumentando el n obtengo un speedup mayor.
- Full Set ($n = 150$).

Método	Tiempo (s)	Desviación estándar (s)
`fasterQDA`	0.00019	0.00039
`FasterQDA`	0.00011	0.00032

Table 5: Tiempos de ejecución de los fasterQDA y FasterQDA.

- Complejidad en tiempo no sólo depende de la complejidad matemática del producto interno.

Speedup de FasterQDA

- Al evitar la matriz $n \times n$, `FasterQDA` lograría un **speedup teórico** de:

$$\frac{classes \times (n \times p^2 + n^2 \times p)}{classes \times (n \times p^2 + n \times p)} = \frac{n}{p} \quad \text{para} \quad n \gg p$$

- En la práctica, las **operaciones adicionales reducen** el speedup, aunque sigue siendo notable.

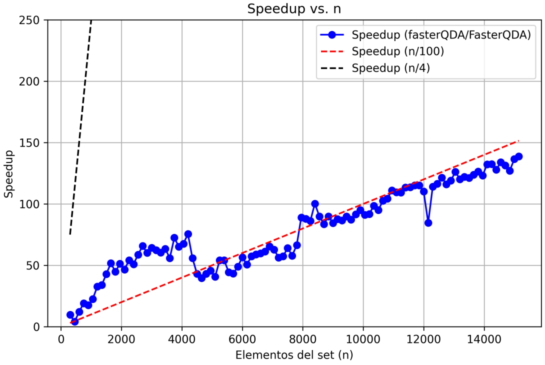

Figure 1: Gráfico de Speedup vs. n

Conclusiones

- Se compararon las diferentes implementaciones de QDA y los tiempos de ejecución.
- Se analizó el costo matemático de las implementaciones **de una sola pasada** y su relación con el tiempo de ejecución.
- Se analizó brevemente el speedup al evitar pasar por matrices $n \times n$.