

Comparativa de Métodos: Implementaciones

Método	<code>_predict_one</code>	<code>predict</code>
QDA	Ciclo for por clase	Ciclo for por observación
TensorizedQDA	Sin ciclos	Ciclo for por observación
fasterQDA	Todo tensorizado	Todo tensorizado

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

Comparativa de Dimensiones: QDA, TensorizedQDA y fasterQDA

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

Cov_inv (Σ_j^{-1}): Matriz de covarianza invertida.

Prod_int: $(\mathbf{x} - \mu_j)^T \Sigma_j^{-1} (\mathbf{x} - \mu_j)$.

log_vero: $\frac{1}{2} \log \left| \Sigma_j^{-1} \right| - \frac{1}{2} (\mathbf{x} - \mu_j)^T \Sigma_j^{-1} (\mathbf{x} - \mu_j)$.

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.

- Uso de `np.diagonal()` para convertir Prod_int de $3 \times n \times n$ a $3 \times n$ en log_vero.

Comparativa de Tiempos: QDA, TensorizedQDA y fasterQDA

- Se utiliza el set de test con un split de 0.3. En este caso, $n = 40$

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.

Comparativa de Tiempos: fasterQDA 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 $n \times n$.
- Se utiliza el set de test con un split de 0.3. En este caso, $n = 45$

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.

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

Comparando, puede verse que la relación en costos entre fasterQDA y FasterQDA es:

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

- Se utiliza el set completo. En este caso, $n = 150$

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.

Speedup de FasterQDA

- En resumen, al evitar la matriz $n \times n$, FasterQDA logra una aceleración significativa, con un speedup teórico de:

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

- Sin embargo, en la práctica, las operaciones adicionales reducen esta aceleración, aunque sigue siendo notable.

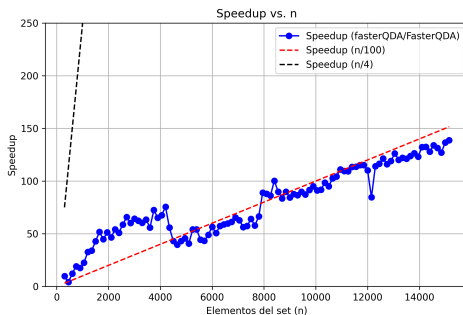


Figure 1: Gráfico de Speedup vs. n