Cuadratic Discriminant Analysis (QDA)

Implementación y Optimización Tensorizada

Analisis Matemático - FUIBA

Autores: Goñi Rodrigo, Corteggiano Tomas

Índice

Notación

A lo largo del presente informe, se adoptan las siguientes notaciones:

- k: cantidad de clases distintas.
- n: cantidad total de observaciones en el conjunto de datos.
- p: dimensión del espacio de características (features).

Es recomendable realizar un análisis exhaustivo de la dimensionalidad de cada tensor antes de implementar los métodos, utilizando funciones tales como `reshape`, `transpose`, `stack`, `diagonal`, y `flatten`, entre otras.

1. Tensorización de QDA

1.1. Comparación entre `QDA` y `TensorizedQDA`

1. **¿Sobre qué paraleliza `TensorizedQDA`? ¿Sobre las k clases, las n observaciones a predecir, o ambas?**

   La implementación `TensorizedQDA` realiza la paralelización exclusivamente sobre las k clases, y no sobre las n observaciones. Esto permite acelerar el cómputo simultáneo de los términos cuadráticos de todas las clases para una misma observación.

2. **Analizar los shapes de de `tensor_inv_covs` y `tensor_means`, y explicar paso a paso cómo es que `TensorizedQDA` llega a predecir lo mismo que `QDA`.**

   En este modelo, se parte de las siguientes estructuras:

   - `self.inv_covs`: lista de k matrices inversas de covarianza, cada una de forma $(p \times p)$.
   - `self.means`: lista de k vectores columna de medias, cada uno de forma $(p \times 1)$.

   Estas listas se convierten en tensores utilizando `np.stack`, lo que da lugar a:

   $$\texttt{tensor\_inv\_cov} \in \mathbb{R}^{k \times p \times p}, \quad \texttt{tensor\_means} \in \mathbb{R}^{k \times p \times 1}$$

   Dada una observación $x \in \mathbb{R}^{p \times 1}$, el modelo QDA calcula el logaritmo de la probabilidad a posteriori para cada clase k. El modelo `TensorizedQDA` permite realizar este cálculo en paralelo para todas las clases mediante operaciones vectorizadas. El procedimiento es el siguiente:

a) **Centrado de la observación:** se resta el vector de medias de cada clase al vector x:

```
unbiased_x = x - self.tensor_means
```

Donde:

- `x` tiene forma $(p, 1)$.
- `self.tensor_means` tiene forma $(k, p, 1)$.

El resultado `unbiased_x` es un tensor de forma $(k, p, 1)$, que contiene los vectores centrados por clase.

b) **Cálculo del producto interno cuadrático:** se utiliza multiplicación matricial sobre los ejes correspondientes:

```
inner_prod = unbiased_x.transpose(0, 2, 1) @ self.tensor_inv_cov @
↪  unbiased_x
```

Detalles:

- `unbiased_x.transpose(0, 2, 1)` cambia la forma a $(k, 1, p)$.
- `self.tensor_inv_cov` tiene forma (k, p, p).
- El primer producto da $(k, 1, p)$ @ $(k, p, p) \Rightarrow (k, 1, p)$.
- El segundo producto da $(k, 1, p)$ @ $(k, p, 1) \Rightarrow (k, 1, 1)$.

Así, `inner_prod` es un tensor de forma $(k, 1, 1)$, que contiene el valor cuadrático $x^\top \Sigma_k^{-1} x$ para cada clase.

c) **Cálculo de la log-verosimilitud condicional:**

```
return 0.5 * np.log(LA.det(self.tensor_inv_cov)) - 0.5 *
↪  inner_prod.flatten()
```

La función `LA.det(self.tensor_inv_cov)` calcula los determinantes de las k matrices de forma paralela, resultando en un vector de dimensión $(k,)$. Luego se aplica el logaritmo elemento a elemento. El producto interno `inner_prod.flatten()` también se convierte en un vector de forma $(k,)$.

El resultado final es un vector con la log-verosimilitud condicional para cada clase k:

$$\log p(x \mid y = k) = \frac{1}{2} \log \det(\Sigma_k^{-1}) - \frac{1}{2} \cdot x_k^\top \Sigma_k^{-1} x_k$$

d) **Predicción final:** se suma a cada log-verosimilitud el logaritmo de la probabilidad a priori de la clase correspondiente, y se selecciona aquella con mayor valor:

```
prediction = np.argmax(self.log_a_priori +
↪  self._predict_log_conditionals(x))
```

Así se obtiene la clase más probable dada la observación x.

1.2. Optimización: `FasterQDA` y `EfficientQDA`

3. **Implementación de `FasterQDA`**

   Se implementa una nueva clase `FasterQDA` que hereda de `TensorizedQDA`, con mejoras sustanciales en cómputo por uso anticipado de operaciones matriciales optimizadas.

4. **Identificación de la matriz $n \times n$**

   En este caso, se considera un conjunto de n observaciones organizadas en una matriz:

   $$X \in \mathbb{R}^{p \times n}$$

   donde cada columna representa una observación distinta con p características.

   Las estadísticas por clase se mantienen:

   - $\mu_k \in \mathbb{R}^{k \times p \times 1}$: pila de vectores de medias por clase.
   - $\Sigma_k^{-1} \in \mathbb{R}^{k \times p \times p}$: pila de matrices inversas de covarianza por clase.

   Para adaptar el cálculo a múltiples observaciones, primero se centra cada columna de X respecto a las medias por clase. Esto se realiza expandiendo X como un tensor de forma $(1 \times p \times n)$ y difundiendo las medias:

   $$X_{\text{centered}} = X - \mu_k \in \mathbb{R}^{k \times p \times n}$$

   Luego, se calcula el producto cuadrático generalizado para todas las clases y todas las observaciones:

   $$Q = (X - \mu_k)^\top \Sigma_k^{-1} (X - \mu_k) \in \mathbb{R}^{k \times n \times n}$$

   Este resultado representa, para cada clase k, una matriz cuadrada $(n \times n)$ donde cada entrada $Q_{k,i,j}$ representa la forma cuadrática entre las observaciones x_i y x_j respecto a la clase k.

   Nota: si se desea únicamente obtener los valores diagonales, es decir, los términos $x_i^\top \Sigma_k^{-1} x_i$ para cada clase y cada observación (usualmente lo que se necesita en clasificación), se extrae la diagonal de este tensor:

   $$\texttt{diag}(Q) \in \mathbb{R}^{k \times n}$$

Estos valores diagonales corresponden a los términos cuadráticos requeridos para calcular las log-verosimilitudes condicionales para cada observación en cada clase.

Finalmente, se calculan las log-verosimilitudes condicionales:

$$\log p(x^{(i)} \mid y = k) = \frac{1}{2} \log \det(\Sigma_k^{-1}) - \frac{1}{2} Q_{k,i,i}$$

y se suman a los logaritmos de las probabilidades a priori para obtener las probabilidades posteriores y realizar la predicción para cada observación.

$$\hat{y}^{(i)} = \arg\max_k \left[\log \pi_k + \log p(x^{(i)} \mid y = k) \right]$$

5. **Demostración de la identidad:**

$$diag(A \cdot B) = \sum_{cols} A \odot B^T = np.sum(A \odot B^T, axis = 1)$$

Sea $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{m \times n}$. La componente i-ésima de la diagonal de AB se puede escribir como:

$$[AB]_{ii} = \sum_{k=1}^{m} A_{ik} B_{ki}$$

Entonces, el vector de la diagonal de AB se define como:

$$\text{diag}(AB) = \left[\sum_{k=1}^{m} A_{1k} B_{k1},\ \sum_{k=1}^{m} A_{2k} B_{k2},\ \ldots,\ \sum_{k=1}^{m} A_{nk} B_{kn} \right]$$

Por otro lado, como $B^T \in \mathbb{R}^{n \times m}$, entonces $(A \odot B^T)_{ij} = A_{ij} B_{ji}$. Luego, sumando sobre las columnas para cada fila, se tiene:

$$\sum_{\text{cols}} A \odot B^T = \left[\sum_{k=1}^{m} A_{1k} B_{k1},\ \sum_{k=1}^{m} A_{2k} B_{k2},\ \ldots,\ \sum_{k=1}^{m} A_{nk} B_{kn} \right]$$

Esto equivale a aplicar la función `np.sum(A  B.T, axis=1)` en NumPy, que computa la suma sobre las columnas (eje 1) del producto elemento a elemento entre A y B^T, dando como resultado el vector con la diagonal de AB.

6. **Optimización en `EfficientQDA`** Ver clase EfficientQDA en QDA.py especificamente en

```
    diag_inner_prod = np.sum(inv_cov_x * unbiased_x, axis=1)  # (k, n)
```

7. **Benchmark comparativo entre modelos:**

Modelo	Tiempo (ms)	Accuracy	Speedup	Memoria Relativa
QDA	3.481492	0.982407	1.000000	1.000000
TensorizedQDA	1.511678	0.982593	2.303065	0.627546
FasterQDA	0.124683	0.985741	27.922748	0.068564
EfficientQDA	0.087479	0.983333	39.798031	0.098019

Cuadro 1: Comparativa de rendimiento entre variantes de QDA

A partir de la tabla, se destacan las siguientes observaciones:

- **Eficiencia computacional:** `FasterQDA` y `EfficientQDA` presentan una mejora drástica en el tiempo de inferencia respecto al QDA clásico. En particular, `EfficientQDA` es aproximadamente 31 veces más rápido, con una latencia de apenas 0.11 ms.
- **Precisión:** Todas las variantes de QDA logran una precisión muy similar, superior al 98 %. De hecho, `EfficientQDA` alcanza la mayor precisión (0.9861), lo cual indica que la optimización no compromete la calidad del modelo.
- **Uso de memoria:** Las versiones `FasterQDA` y `EfficientQDA` también reducen significativamente el consumo de memoria relativa, siendo las más ligeras.

Conclusión: Las variantes optimizadas (`FasterQDA` y `EfficientQDA`) no sólo conservan la precisión del modelo clásico, sino que lo superan en términos de velocidad y eficiencia de memoria. Este comportamiento es consistente con lo esperado: al aprovechar operaciones vectorizadas y evitar bucles innecesarios, se logra una ejecución más rápida sin sacrificar precision ni aumentar el consumo de memoria.

2. Factorización de Cholesky

2.1. QDA con Descomposición Cholesky

8. **Si una matriz A tiene fact. de Cholesky $A = LL^T$, expresar A^{-1} en términos de L. ¿Cómo podría esto ser útil en la forma cuadrática de QDA?**

Siendo $A = LL^T$, la inversa de A se puede expresar como:

$$A^{-1} = L^{-T}L^{-1}$$

Retomando la ecuación de la log-verosimilitud condicional:

$$\log f_j(x) = -\frac{1}{2}\log|\Sigma_j| - \frac{1}{2}(x-\mu_j)^T\Sigma_j^{-1}(x-\mu_j) + C$$

Si

$$\Sigma_j = L_j L_j^T$$

entonces su inversa es:

$$\Sigma_j^{-1} = L_j^{-T} L_j^{-1}$$

Definiendo el término:

$$\delta(x) = (x - \mu_j)^T \Sigma_j^{-1} (x - \mu_j)$$

y reemplazando:

$$\delta(x) = (x - \mu_j)^T L_j^{-T} L_j^{-1} (x - \mu_j)$$

Dado que:

$$(L_j^{-1}(x - \mu_j))^T = (x - \mu_j)^T L_j^{-T}$$

Si definimos $z = L_j^{-1}(x - \mu_j)$, se tiene:

$$\delta(x) = z^T z = \|z\|^2 = \|L_j^{-1}(x - \mu_j)\|^2$$

Esto es computacionalmente más eficiente, ya que invertir L_j, una matriz triangular inferior, es más rápido que invertir Σ_j directamente.

Por otro lado, reemplazando la factorización en el término del log-determinante:

$$-\frac{1}{2} \log |\Sigma_j| = -\frac{1}{2} \log |L_j L_j^T|$$

Usando la propiedad del determinante de un producto:

$$|L_j L_j^T| = |L_j||L_j^T|$$

Como $|L_j^T| = |L_j|$ (la transpuesta no cambia el determinante) se tiene:

$$|L_j L_j^T| = |L_j|^2$$

Finalmente, dado que L_j es una matriz triangular inferior, su determinante es simplemente el producto de los elementos en su diagonal. Esto también es computacionalmente eficiente.

9. **Explicar las diferencias entre `QDA_Chol1` y `QDA` y cómo `QDA_Chol1` llega, paso a paso, hasta las predicciones.**

   La diferencia más notable entre `QDA_Chol1` y `QDA` se encuentra en la etapa de *training*. En `QDA` se calcula la matriz de covarianza para cada clase Σ_j, mientras que en `QDA_Chol1`, aplicando la factorización de Cholesky y considerando el resultado del punto 8, se logra un algoritmo en donde se computa la inversa de una matriz triangular inferior, siendo este cómputo más rápido. Esto se muestra en el siguiente bloque de código:

```
    self.L_invs = [
        LA.inv(cholesky(np.cov(X[:, y.flatten() == idx], bias=True),
        ↪  lower=True))
        for idx in range(len(self.log_a_priori))
    ]
```

La función `cholesky` obtiene, en este caso con el parámetro `lower=True`, la matriz inferior resultante de la factorización. Luego, para cada clase, se calcula su inversa.

El resultado del punto 8 también simplifica la etapa de predicción, usando:

$$\delta(x) = z^T z = \|z\|^2 = \|L_j^{-1}(x - \mu_j)\|^2$$

Entonces el código correspondiente se simplifica como:

```
    y = L_inv @ unbiased_x
    return np.log(L_inv.diagonal().prod()) - 0.5 * (y**2).sum()
```

10. **¿Cuáles son las diferencias entre `QDA_Chol1`, `QDA_Chol2` y `QDA_Chol3`?**

    La principal diferencia entre `QDA_Chol1`, `QDA_Chol2` y `QDA_Chol3` yace en la forma en que se calcula y y, por lo tanto, en cómo se trata a L_j.

    En `QDA_Chol1` y `QDA_Chol3`, en la etapa de *training* se calcula la inversa de la matriz triangular inferior L_j. En el primer caso, se hace de la forma clásica:

```
    LA.inv(cholesky(np.cov(X[:, y.flatten() == idx], bias=True),
    ↪  lower=True))
```

    y en el segundo caso se utiliza el algoritmo `DTRTRI`, que computa la inversa de una matriz triangular superior o inferior:

```
    dtrtri(cholesky(np.cov(X[:, y.flatten() == idx], bias=True),
    ↪  lower=True), lower=1)[0]
```

    En el caso de `QDA_Chol2` simplemente se calcula la matriz triangular inferior L_j. Esto implica una diferencia con las otras dos variantes, ya que al momento de predecir es necesario calcular y y, por lo tanto, obtener la inversa de L_j.

    Sin embargo, calcular $L_j^{-1}(x - \mu_j)$ equivale a resolver el sistema lineal

$$L_j y = (x - \mu_j)$$

    por lo que es posible usar la función `solve_triangular`, un método que resuelve la ecuación $ax = b$ para x, asumiendo que a es una matriz triangular:

```
    y = solve_triangular(L, unbiased_x, lower=True)
```

11. **Comparar la performance de las 7 variantes de QDA implementadas**

Modelo	Tiempo (ms)	Accuracy	Speedup	Memoria Relativa
QDA	3.481492	0.982407	1.000000	1.000000
TensorizedQDA	1.511678	0.982593	2.303065	0.627546
FasterQDA	0.124683	0.985741	27.922748	0.068564
EfficientQDA	0.087479	0.983333	39.798031	0.098019
QDA_Chol1	1.917189	0.986111	1.815935	0.974185
QDA_Chol2	5.746567	0.982222	0.605839	0.988842
QDA_Chol3	1.950756	0.984444	1.784689	0.988842

Cuadro 2: Comparativa de rendimiento entre variantes de QDA sin tensorización

Observando los resultados de la tabla y comparando únicamente los modelos sin tensorización —es decir, `QDA`, `QDA_Chol1`, `QDA_Chol2` y `QDA_Chol3`—, notamos que el uso de la factorización de Cholesky permite un *speedup* de aproximadamente $2\times$ en el caso de `QDA_Chol1` y `QDA_Chol3`. Esto es esperable, dado que reduce la complejidad computacional al evitar el cálculo directo de la inversa de la matriz de covarianza, utilizando en su lugar una matriz triangular inferior obtenida mediante la descomposición de Cholesky. Esta matriz se calcula una única vez durante la etapa de entrenamiento.

Sin embargo, en el caso de `QDA_Chol2`, observamos una ligera degradación en el rendimiento con respecto al modelo base (`QDA`). Esto se debe a que `QDA_Chol2` sólo calcula la matriz triangular inferior de Cholesky durante el entrenamiento, y luego, en cada inferencia, debe resolver el sistema lineal

$$Ax = b$$

utilizando `solve_triangular`. Este enfoque resulta más costoso computacionalmente en tiempo de inferencia que calcular una única inversa en la etapa de entrenamiento, al menos en este caso específico.

Finalmente, puede observarse que la mejora obtenida solo por factorizar las matrices de covarianza es comparable a la optimización lograda mediante el tensorizado por clases implementado en la clase `TensorizedQDA`.

2.2. Optimización

12. **Implementar el modelo `TensorizedChol` paralelizando sobre clases/observaciones según corresponda.**

Permite cálculo vectorizado de todas las k factorizaciones y predicciones.

13. **Implementación de `EfficientChol`:**

    Combina vectorización y uso de diagonales de productos, análogo a `EfficientQDA`.

14. **Comparar la performance de las 9 variantes de QDA**

Modelo	Tiempo (ms)	Accuracy	Speedup	Memoria Relativa
QDA	3.481492	0.982407	1.000000	1.000000
TensorizedQDA	1.511678	0.982593	2.303065	0.627546
FasterQDA	0.124683	0.985741	27.922748	0.068564
EfficientQDA	0.087479	0.983333	39.798031	0.098019
QDA_Chol1	1.917189	0.986111	1.815935	0.974185
QDA_Chol2	5.746567	0.982222	0.605839	0.988842
QDA_Chol3	1.950756	0.984444	1.784689	0.988842
TensorizedChol	1.408362	0.986667	2.472014	0.590344
EfficientChol	0.075791	0.985556	45.935429	0.097863

Cuadro 3: Comparativa de rendimiento entre las 9 variantes de QDA

Análisis cualitativo Al analizar la tabla se observa que los modelos más eficientes en términos de tiempo de inferencia y reducción de memoria son `EfficientChol` y `EfficientQDA`, seguidos por `FasterQDA`. Estas variantes combinan técnicas avanzadas de optimización, como el uso de la factorización de Cholesky junto con implementaciones eficientes y posiblemente paralelización o tensorización, lo que les permite lograr aceleraciones de hasta casi 46 veces respecto al modelo base `QDA`.

Los modelos basados únicamente en la factorización de Cholesky sin optimizaciones adicionales (`QDA_Chol1` y `QDA_Chol3`) presentan mejoras moderadas en velocidad (alrededor de 1.8 a 1.9 veces más rápidos que `QDA`) y mantienen una precisión ligeramente mejorada. Sin embargo, `QDA_Chol2` se muestra como una excepción, ya que su rendimiento es incluso peor que el modelo base, probablemente debido a que realiza la resolución de sistemas lineales en cada inferencia, lo que resulta costoso computacionalmente.

Las variantes con tensorización (`TensorizedQDA` y `TensorizedChol`) también muestran mejoras importantes, logrando aceleraciones alrededor de 2 a 2.5 veces y buena reducción de memoria, demostrando que aprovechar la estructura de los datos y las operaciones vectorizadas aporta beneficios significativos.

En cuanto a la precisión, todos los modelos mantienen valores muy altos y comparables, con ligeras mejoras en algunas variantes optimizadas, lo que indica que las optimizaciones no comprometen la calidad del clasificador.