Tercer Parcial PP: Regresión Lineal

1. Diseño PI-Calculus con Concurrencia

El **π-calculus** modela sistemas concurrentes mediante paso de mensajes en canales. Para regresión lineal, diseñamos procesos paralelos que se comunican:

- Nivel 1: Dividir dataset en N particiones, calcular gradientes en paralelo
- Nivel 2: Pipeline de épocas (época i predice mientras época i-1 actualiza)

Tercer Parcial PP: Regresión Lineal

1. Diseño PI-Calculus con Concurrencia

El **π-calculus** modela sistemas concurrentes mediante paso de mensajes. Para regresión lineal diseñamos procesos paralelos:

Procesos: Master (coordina) | Predictor (calcula y_{pred}) | GradientCalc (gradientes) | ParamUpdater (actualiza)

Canales: params_ch, pred_ch, gradient_ch, update_ch

Paralelización:

- Dividir X,y en N particiones → calcular gradientes parciales en paralelo → agregar
- Pipeline: epoch_i (predict) | epoch_i (calcgrad) | epoch_{i-1} (update)

Ventajas: Escalable, modular, dinámico, tolerancia a fallos.

---

1. Diseño con Paradigma de Concurrencia y Cálculo PI

1.1 Fundamentos del π-calculus

El **π-calculus** es un modelo formal de computación concurrente basado en el paso de mensajes a través de canales. Permite describir sistemas distribuidos con topología dinámica, donde los nombres de canales pueden comunicarse entre procesos[1].

Características fundamentales:

- **Concurrencia (P | Q)**: Dos procesos ejecutándose en paralelo
- **Comunicación de entrada c(x).P**: Proceso espera mensaje en canal c
- **Comunicación de salida** $\overline{c}\langle y\rangle.P$: Envía valor y en canal c
- **Replicación (!P)**: Crea copias ilimitadas de proceso P
- **Restricción (ν x)P**: Crea nuevo canal local x
- **Nil (0)**: Proceso terminado

1.2 Arquitectura del Sistema Concurrente

Para implementar regresión lineal con π-calculus, proponemos los siguientes procesos paralelos:

Proceso	Responsabilidad
Master	Coordina épocas y flujo de entrenamiento
Predictor	Calcula $y_{pred} = w \cdot X + b$
GradientCalc	Computa gradientes dw y db
ParamUpdater	Actualiza parámetros usando gradientes
ErrorMonitor	Calcula MSE y reporta progreso

Table 1: Procesos principales del diseño concurrente

1.3 Canales de Comunicación

Los canales permiten sincronización y comunicación entre procesos:

- **params_channel**: Transmite parámetros actuales (w, b)
- **pred_channel**: Envía predicciones calculadas
- **gradient_channel**: Comunica gradientes calculados
- **update_channel**: Notifica parámetros actualizados
- **control_channel**: Coordina sincronización entre épocas
- **error_channel**: Reporta MSE para monitoreo

1.4 Especificación Formal en Notación PI

La arquitectura se especifica formalmente como:

```
Master = (ν params_ch, pred_ch, grad_ch, upd_ch).
params_ch⟨w_init, b_init⟩ |
!Epoch(params_ch, pred_ch, grad_ch, upd_ch, epoch_count)

Epoch(params, pred_ch, grad_ch, upd_ch, count) =
if count < MAX_EPOCHS then
params(w, b).
Predictor(pred_ch) | GradientCalc(grad_ch) |
ParamUpdater(upd_ch) |
pred_ch⟨w, b, X, y⟩.
pred_ch(y_pred).
grad_ch⟨y_pred, y⟩.
grad_ch(dw, db).
upd_ch⟨w, b, dw, db, lr⟩.
upd_ch(w_new, b_new).
params⟨w_new, b_new⟩ |
Epoch(params, count+1)
else
params(w_final, b_final).
output⟨w_final, b_final⟩

Predictor(pred_ch) =
pred_ch(w, b, X, y).
let y_pred = w * X + b in
pred_ch⟨y_pred⟩

GradientCalc(grad_ch) =
grad_ch(y_pred, y).
let error = y_pred - y in
let dw = (2/m) * sum(error * X) in
let db = (2/m) * sum(error) in
grad_ch⟨dw, db⟩

ParamUpdater(upd_ch) =
upd_ch(w, b, dw, db, lr).
let w_new = w - lr * dw in
let b_new = b - lr * db in
upd_ch⟨w_new, b_new⟩
```

1.5 Estrategia de Paralelización

Nivel 1 - Paralelización de Datos

Dividir el dataset X, y en N particiones y calcular gradientes parciales en paralelo:

```
(ν partition1_ch, ..., partitionN_ch).
(Partition1(partition1_ch) |
Partition2(partition2_ch) |
...
PartitionN(partitionN_ch)) |
GradientAggregator(partition1_ch, ..., partitionN_ch)
```

Nivel 2 - Pipeline de Épocas

Mientras una época calcula gradientes, otra puede actualizar parámetros en paralelo:

Epoch_i(Predict) | Epoch_i(CalcGrad) | Epoch_{i-1}(Update)

Barrera de Sincronización

Implementar barreras para coordinar múltiples procesos:

Barrier(sync_ch, n_workers) =
let count = ref 0 in
!sync_ch(worker_id).
count := !count + 1 |
if !count == n_workers then
broadcast⟨continue⟩ |
count := 0

1.6 Ventajas del Diseño Concurrente

- **Escalabilidad**: Cada proceso ejecutable en núcleos o máquinas diferentes
- **Modularidad**: Separación clara de responsabilidades entre procesos
- **Dinamismo**: Canales creados dinámicamente para cada época
- **Tolerancia a fallos**: Procesos pueden replicarse usando $!P$
- **Composicionalidad**: Procesos se pueden combinar de formas nuevas

1.7 Complejidad y Consideraciones

- **Tiempo total**: O(épocas × (n + log (particiones))) con paralelización óptima
- **Comunicación**: Overhead de paso de mensajes puede dominar con datasets muy pequeños
- **Synchronization**: Requerida al final de cada época para agregación de gradientes
- **Ideal para**: Regresión con datasets masivos distribuidos en múltiples nodos

---

2. Diseño Orientado a Aspectos (AOP)

Conceptos clave: Concern, Cross-cutting concern, Join point, Pointcut, Advice, Aspect

Preocupaciones transversales: Logging, Performance Monitoring, Validation, Error Handling, Caching, Profiling

Código Core (sin aspectos):
class LinearRegression:
def predict(self, X): return self.w * X + self.b
def compute_gradients(self, X, y, error):
m = len(X)
dw = (2/m) * np.dot(error, X)
db = (2/m) * np.sum(error)
return dw, db
def train(self, X, y):

for epoch in range(self.epochs):
y_pred = self.predict(X)
error = y_pred - y
dw, db = self.compute_gradients(X, y, error)
self.update_parameters(dw, db)

Aspectos aplicados:

2.4 Diseño de Aspectos

- **Logging:** Before train() → log "Iniciando". After each 200 epochs → log MSE, w, b
- **Performance:** Around train() → medir tiempo. Around predict() → cache resultados
- **Validation:** After compute_gradients() → verificar no hay NaN. After update → validar convergencia
- **Convergence:** Monitorear MSE improvement. Si < threshold 10 épocas → early stop
- **Caching:** Cache de predicciones con hash(X, w, b) como key

Join Points: Method execution (train, predict), field access (w, b), exception handling, loop iterations

Weaving: Compilador AspectJ/Spring inserta advices en compile-time. Resultado: código limpio + debug/monitoring automático

Ventajas: Separación de concerns, reutilización, mantenibilidad, debugging integrado

---

3. Implementación Rust

Código compilado:
use std::time::Instant;
fn main() {
let x: Vec<f64> = vec![1.0, 2.0, 3.0, 4.0, 5.0];
let y: Vec<f64> = vec![2.0, 4.0, 6.0, 8.0, 10.0];
let mut w = 0.0; let mut b = 0.0;
let lr = 0.01; let epochs = 1000;
let m = x.len() as f64;
let start = Instant::now();

```
for epoch in 0..epochs {
    let y_pred: Vec<f64> = x.iter().map(|&xi| w * xi + b).collect();
    let error: Vec<f64> = y_pred.iter().zip(y.iter())
        .map(|(&p, &a)| p - a).collect();
    let dw = (2.0/m) * x.iter().zip(error.iter())
        .map(|(&xi, &ei)| ei * xi).sum::<f64>();
    let db = (2.0/m) * error.iter().sum::<f64>();
    w -= lr * dw; b -= lr * db;
}
let duration = start.elapsed();
```

```
println!("Tiempo: {:?}", duration);
```

}

Compilación: `rustc -O linear_regression.rs` o `cargo run --release`

Desempeño:

- Python: **21.11 ms** (promedio 100 ejecuciones)
- Rust: **2-4 ms** (esperado, 5-10x más rápido)
- Razones: Código nativo, sin GC, zero-cost abstractions, optimizaciones LLVM, sin GIL

Comparación:

- **Python:** Interpretado, GC automático, baja latencia de desarrollo
- **Rust:** Compilado, ownership system, máxima performance, threading real

Cuándo usar cada uno:

- Python: Prototipado rápido, datasets pequeños, ecosistema ML establecido
- Rust: Producción, datasets masivos, crítico latencia, embedded systems

---

Resumen Comparativo

Paradigma	Mejor Para	Overhead	Escalabilidad
PI-Calculus	Sistemas distribuidos, paralelismo masivo	Comunicación entre procesos	Excelente
AOP	Mantenibilidad, debugging, separación concerns	Mínimo (compile-time)	No afecta
Rust	Producción, baja latencia, performance crítica	Ninguno	Lineal con datos