Large-scale Logistic Regression and Linear Support Vector Machines Using Spark

Jhonatan Hernandez Ricardo Lopez Miguel Angulo

November 2025

`https://github.com/Jhon68h/taller_reproducibilidad`

1 Introducción

La regresión logística (LR) y las máquinas de soporte vectorial lineales (SVM) son métodos ampliamente utilizados para tareas de clasificación. Sin embargo, el manejo de conjuntos de datos de gran tamaño supera la capacidad de procesamiento de una sola máquina. Para abordar este problema, se han desarrollado soluciones distribuidas que aprovechan plataformas como Apache Spark.

Spark permite realizar cómputo en memoria, reduciendo el costo de operaciones iterativas comunes en el entrenamiento de modelos de aprendizaje automático. En el trabajo de Lin et al. (2014), se propone una implementación distribuida del método de Newton con región de confianza (TRON) para entrenar modelos de LR y SVM lineales sobre Spark, denominada *Spark LIBLINEAR*. Esta herramienta busca combinar la eficiencia de los métodos de segundo orden con la escalabilidad de la computación distribuida.

El objetivo de este proyecto es reproducir los resultados del artículo mencionado, evaluando el rendimiento y comportamiento del algoritmo en entornos distribuidos. Con ello se pretende analizar la eficiencia, escalabilidad y viabilidad práctica del método propuesto en sistemas de procesamiento de datos a gran escala.

2 Marco Teórico

2.1 Modelos lineales y formulación convexa

Dado un conjunto de entrenamiento $\{(\mathbf{x}_i, y_i)\}_{i=1}^{l}$ con $\mathbf{x}_i \in R^n$ y $y_i \in \{-1, 1\}$, la familia de clasificadores lineales (LR y SVM lineal) se entrena resolviendo

$$\min_{\mathbf{w}\in R^n} \; f(\mathbf{w}) \;\equiv\; \frac{1}{2}\,\mathbf{w}^\top \mathbf{w} \;+\; C\sum_{i=1}^{l} \xi(\mathbf{w}; \mathbf{x}_i, y_i), \tag{1}$$

donde $C > 0$ y la pérdida ξ es, entre otras, logística $\xi(\mathbf{w}; \mathbf{x}_i, y_i) = \log\bigl(1 + \exp(-y_i\, \mathbf{w}^\top \mathbf{x}_i)\bigr)$ (LR), o la pérdida cuadrática *hinge* para SVM L_2. Ambas son diferenciables (salvo L_1-hinge), lo que permite métodos de segundo orden.

2.2 Método de Newton con región de confianza (TRON)

TRON resuelve (1) mediante iteraciones de región de confianza. Dado $\mathbf{w}_t$, calcula un paso $\mathbf{d}_t$ resolviendo aproximadamente el subproblema cuadrático

$$\min_{\|\mathbf{d}\| \le \Delta_t} \; q_t(\mathbf{d}) \;\equiv\; \nabla f(\mathbf{w}_t)^\top \mathbf{d} + \tfrac{1}{2}\, \mathbf{d}^\top \nabla^2 f(\mathbf{w}_t)\, \mathbf{d}, \tag{2}$$

donde $\Delta_t > 0$ es el radio de confianza. El subproblema se resuelve con *conjugate gradients* (CG) sin formar el Hessiano explícito; sólo se requieren productos Hessiano–vector.

Hessiano en LR y producto Hessiano–vector. Para LR,

$$\nabla^2 f(\mathbf{w}) \;=\; I \;+\; C\, X^\top D X, \quad D = \operatorname{diag}\!\left(\frac{\exp(-y_i\, \mathbf{w}^\top \mathbf{x}_i)}{(1 + \exp(-y_i\, \mathbf{w}^\top \mathbf{x}_i))^2} \right),$$

de modo que cada iteración de CG usa

$$\nabla^2 f(\mathbf{w})\, \mathbf{v} \;=\; \mathbf{v} \;+\; C\, X^\top \bigl(D\, (X \mathbf{v}) \bigr), \tag{3}$$

evitando almacenar $\nabla^2 f$. Para SVM L_2 se emplea un Hessiano generalizado análogo en el producto.

Criterio de aceptación y actualización de Δ_t. Calculado $\mathbf{d}_t$, se evalúa

$$\rho_t \;=\; \frac{f(\mathbf{w}_t + \mathbf{d}_t) - f(\mathbf{w}_t)}{q_t(\mathbf{d}_t)},$$

y se acepta $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{d}_t$ si $\rho_t > \eta$; en caso contrario, se rechaza. El radio Δ_t se actualiza por reglas estándar de región de confianza (contracción/expansión) en función de ρ_t.

2.3 Descomposición distribuida del objetivo y gradiente

Sea $X = [X_1; \ldots; X_p]$ una partición por filas en p partes (y $Y = \operatorname{diag}(y_1, \ldots, y_l)$). Definiendo operaciones componente a componente con $\sigma(\mathbf{v}) = [\, 1 + \exp(-v_j)\,]_j$, el objetivo, gradiente y HVP se descomponen como

$$f(\mathbf{w}) = \tfrac{1}{2}\,\mathbf{w}^\top \mathbf{w} \;+\; C \sum_{k=1}^{p} f_k(\mathbf{w}), \tag{4a}$$

$$f_k(\mathbf{w}) = \mathbf{1}_{l_k}^\top \log\big(\sigma(Y_k X_k \mathbf{w})\big), \tag{4b}$$

$$\nabla f(\mathbf{w}) = \mathbf{w} \;+\; C \sum_{k=1}^{p} \nabla f_k(\mathbf{w}), \tag{4c}$$

$$\nabla f_k(\mathbf{w}) = (Y_k X_k)^\top \Big(\sigma(Y_k X_k \mathbf{w})^{-1} - \mathbf{1}_{l_k}\Big), \tag{4d}$$

$$\nabla^2 f(\mathbf{w})\,\mathbf{v} = \mathbf{v} \;+\; C \sum_{k=1}^{p} \nabla^2 f_k(\mathbf{w})\,\mathbf{v}, \tag{4e}$$

$$\nabla^2 f_k(\mathbf{w})\,\mathbf{v} = X_k^\top \Big(D_k\,(X_k \mathbf{v})\Big), \tag{4f}$$

Donde:

- $\mathbf{1}_{l_k}$ es el vector de unos de dimensión l_k (número de ejemplos en el bloque k).
- $\sigma(\cdot) = 1 + \exp\big(-(\cdot)\big)$ se aplica componente a componente.
- $\log(\cdot)$ y $(\cdot)^{-1}$ se aplican componente a componente.
- $D_k = \operatorname{diag}\Big(\frac{\sigma(Y_k X_k \mathbf{w}) - \mathbf{1}_{l_k}}{\sigma(Y_k X_k \mathbf{w})^{\circ 2}}\Big)$, donde $^{\circ 2}$ indica potencia componente a componente.

donde $D_k = \operatorname{diag}\big((\sigma(Y_k X_k \mathbf{w}) - \mathbf{e}_k)/\sigma(Y_k X_k \mathbf{w})^2\big)$ y $\mathbf{e}_k$ es el vector de unos del bloque k. Esta forma habilita *map-reduce*: cada nodo computa $f_k, \nabla f_k, \nabla^2 f_k(\cdot)$ localmente y el maestro agrega.

2.4 Ejecución sobre Apache Spark

Spark expone RDDs (colecciones inmutables, particionadas y tolerantes a fallos) y operaciones paralelas (`transformations/actions`). Para TRON distribuido: el maestro difunde $\mathbf{w}_t$ (y vectores CG $\mathbf{v}$), los *workers* calculan $f_k, \nabla f_k$ y los términos de (4e), y se reduce al maestro para decidir $\mathbf{d}_t$, ρ_t y Δ_t.

Particionado y paralelismo. Si s es el número de *workers* y p el de particiones, conviene $p > s$ para explotar el paralelismo (Spark fija un $p_{\min}$ según el tamaño de datos si no se especifica).

Primitivas clave de implementación.

- `mapPartitions` en lugar de `map`: evita crear vectores intermedios por muestra y acumula sobre un único buffer denso por partición (menor sobrecosto y GC), tanto para el gradiente como para (3).
- **Broadcast de w (y v)**: se envía una sola vez por iteración TRON, reduciendo comunicaciones cuando $p \gg s$ y n es grande.
- `coalesce` antes de `reduce`: combina salidas locales a nivel de nodo; se pasa de comunicar p vectores de tamaño n a sólo s.
- **No cachear $\sigma(YX\mathbf{w})$ en RDDs**: RDDs son de sólo lectura; crear RDDs por iteración alarga *lineage* y añade copias/overhead. Es más eficiente recomputar que *cachear* y redistribuir este vector.

Tolerancia a fallos y *lineage*. Spark reconstruye particiones perdidas reejecutando transformaciones registradas; la inmutabilidad de RDDs garantiza determinismo en la recomputación y soporta fallos de nodos sin intervención del usuario.

2.5 Algoritmo maestro–trabajador (resumen)

En cada iteración t del método TRON distribuido:

[label=(v)]

1. **Difusión.** El maestro difunde el vector $\mathbf{w}_t$ a todos los *workers*.
2. **Cómputo local.** Cada *worker* procesa sus particiones con `mapPartitions` y calcula $f_k(\mathbf{w}_t)$, $\nabla f_k(\mathbf{w}_t)$ y los términos locales para productos Hessiano–vector cuando se requieran (esto es, $X_k^\top\big(D_k(X_k\mathbf{v})\big)$ en LR).
3. **Agregación.** Se aplica `coalesce` para combinar salidas a nivel de nodo y luego se reduce al maestro:

$$f(\mathbf{w}_t) = \tfrac{1}{2}\,\mathbf{w}_t^\top\mathbf{w}_t + C\sum_k f_k(\mathbf{w}_t), \qquad \nabla f(\mathbf{w}_t) = \mathbf{w}_t + C\sum_k \nabla f_k(\mathbf{w}_t).$$

4. **Criterio de parada.** Si $\|\nabla f(\mathbf{w}_t)\| < \varepsilon$, detener.
5. **Paso de Newton truncado.** El maestro ejecuta CG para aproximar el minimizador de $q_t(\mathbf{d})$ con restricción $\|\mathbf{d}\| \le \Delta_t$, solicitando a los *workers* los productos Hessiano–vector según

$$\nabla^2 f(\mathbf{w}_t)\,\mathbf{v} \;=\; \mathbf{v} \;+\; C\sum_k X_k^\top\big(D_k(X_k\mathbf{v})\big),$$

y truncando si se alcanza la frontera de la región de confianza.

6. **Aceptación y actualización.** Calcular $\rho_t = \dfrac{f(\mathbf{w}_t + \mathbf{d}_t) - f(\mathbf{w}_t)}{q_t(\mathbf{d}_t)}$. Si $\rho_t > \eta$, aceptar $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{d}_t$; en caso contrario, $\mathbf{w}_{t+1} = \mathbf{w}_t$. Actualizar el radio Δ_{t+1} según las reglas estándar de región de confianza (contracción/expansión según ρ_t).

3 Implementación: algoritmos, fórmulas y flujo

1. **Función objetivo (LR / SVM lineal L2).**

$$f(\mathbf{w}) \;=\; \tfrac{1}{2}\,\|\mathbf{w}\|^2 \;+\; C\sum_{i=1}^{l}\xi\big(\mathbf{w};\mathbf{x}_i,y_i\big)$$

$$\xi\big(\mathbf{w};\mathbf{x}_i,y_i\big) \;=\; \begin{cases} \log\big(1+\exp(-y_i\,\mathbf{w}^\top\mathbf{x}_i)\big), & \text{LR},\\ \max\big(0,\,1-y_i\,\mathbf{w}^\top\mathbf{x}_i\big)^2, & \text{SVM L2}. \end{cases}$$

es sólo la suma de dos partes: (i) una regularización L2, $\frac{1}{2}\|\mathbf{w}\|^2$, que controla el tamaño de los pesos, y (ii) la suma de pérdidas por ejemplo, escalada por C. Según el modelo eliges *una* de las dos pérdidas: logística (LR) o hinge cuadrática (SVM L2).

```python
class LogisticLoss(BaseLoss):
    """
    Pérdida logística binaria (y ∈ {+1, -1}) con regularización L2.
    Definiciones (con m = Xw y t = -y * m):
      f(w) = 0.5 ||w||^2 + C * sum log(1 + exp(t))
      g(w) = w + C * X^T( -y * σ(t) )
      H(w) @ s = s + C * X^T( D * (X s) ), D_i = σ(t_i)*(1-σ(t_i))
    """

    def f(self, w: np.ndarray) -> float:
        m = self.backend.margin(w)                  # m = Xw
        t = -self.y * m                             # t = -y * m
        loss_sum = _softplus(t).sum()
        reg = 0.5 * float(w @ w)
        return reg + self.C * float(loss_sum)

    def grad(self, w: np.ndarray) -> np.ndarray:
        m = self.backend.margin(w)                  # m = Xw
        t = -self.y * m
        p = _expit(t)                               # σ(t)
        r = -self.y * p                             # tamaño l
        return w + self.C * self.backend.Xt_dot(r) # tamaño n

    def hess_vec(self, w: np.ndarray, s: np.ndarray) -> np.ndarray:
        m = self.backend.margin(w)                  # m = Xw
        t = -self.y * m
        p = _expit(t)                               # σ(t)
        D = p * (1.0 - p)                           # diag elementos (tamaño l)
        Xs = self.backend.X_dot(s)                  # tamaño l
        return s + self.C * self.backend.Xt_dot(D * Xs)
```

2. **Gradiente (descomposición por particiones).**

$$\nabla f(\mathbf{w}) \;=\; \mathbf{w} \;+\; C\sum_{k=1}^{p}\nabla f_k(\mathbf{w}),$$

donde cada bloque k (partición por filas X_k) calcula su contribución local $\nabla f_k(\mathbf{w})$ y luego el maestro suma todas las contribuciones.

el gradiente total es la suma de (i) la regularización $\mathbf{w}$ y (ii) los gradientes locales escalados por C. Cada *worker* procesa su partición X_k (en paralelo), computa $\nabla f_k(\mathbf{w})$ y se hace una *reducción* para obtener $\sum_k \nabla f_k(\mathbf{w})$.

```
if self.mode == "map":
    # Cada fila contribuye a z parcialmente
    def contrib_row(row, r_vec, n_feat):
        i, idx, val = row
        ri = r_vec[int(i)]
        z = np.zeros(n_feat, dtype=np.float64)
        for j, v in zip(idx, val):  # idx y val son listas
            z[j] += v * ri
        return z

    z = (
        rdd_local
        .map(lambda row: contrib_row(row, bc_r.value, n_features))
        .reduce(lambda a, b: a + b)
    )

else:
    # Acumular contribuciones por partición
    def contrib_part(it, r_vec, n_feat):
        z = np.zeros(n_feat, dtype=np.float64)
        for i, idx, val in it:
            ri = r_vec[int(i)]
            for j, v in zip(idx, val):
                z[j] += v * ri
        yield z

    rdd2 = rdd_local
    if self.num_slaves and self.num_slaves > 0:
        try:
            rdd2 = rdd2.coalesce(self.num_slaves)
        except Exception:
            pass

    z = (
        rdd2
        .mapPartitions(lambda it: contrib_part(it, bc_r.value, n_features))
        .reduce(lambda a, b: a + b)
    )

bc_r.unpersist()
return np.asarray(z, dtype=np.float64)
```

3. **Producto Hessiano–vector (HVP) para LR.**

$$\nabla^2 f(\mathbf{w})\,\mathbf{v} \;=\; \mathbf{v} \;+\; C\,X^\top\big(D\,(X\mathbf{v})\big), \qquad D = \operatorname{diag}\left(\frac{e^{-y_i\,\mathbf{w}^\top \mathbf{x}_i}}{\left(1+e^{-y_i\,\mathbf{w}^\top \mathbf{x}_i}\right)^2}\right).$$

no se forma el Hessiano; sólo se hacen tres pasos:
(i) proyectar $\mathbf{v}$ al espacio de ejemplos ($z = X\mathbf{v}$), (ii) escalar componente a componente con D (derivada de la sigmoide), y (iii) volver al espacio de parámetros ($X^\top(Dz)$), para finalmente sumar $\mathbf{v}$. Esto permite usar CG de forma eficiente en problemas grandes.

```
def hess_vec(self, w: np.ndarray, s: np.ndarray) -> np.ndarray:
    m = self.backend.margin(w)                  # m = Xw
    t = -self.y * m
    p = _expit(t)                               # σ(t)
    D = p * (1.0 - p)                           # diag elementos (tamaño l)
    Xs = self.backend.X_dot(s)                  # tamaño l
    return s + self.C * self.backend.Xt_dot(D * Xs)
```

4. **Conjugate Gradients (CG) truncado para el subproblema de TRON.**

$$\min_{\|\mathbf{d}\| \le \Delta_t} \; q_t(\mathbf{d}) \;=\; \nabla f(\mathbf{w}_t)^\top \mathbf{d} \;+\; \tfrac{1}{2}\, \mathbf{d}^\top \nabla^2 f(\mathbf{w}_t)\, \mathbf{d}.$$

El subproblema se resuelve con CG sin formar el Hessiano, usando llamadas repetidas al HVP.

CG busca una buena dirección $\mathbf{d}$ dentro de la región de confianza de radio Δ_t minimizando el modelo cuadrático q_t. Se *trunca* cuando: (i) el residuo cae bajo una tolerancia (precisión suficiente), y/o (ii) la norma del paso propuesto alcanza la frontera $\|\bar{\mathbf{d}}\| = \Delta_t$. Esto controla el costo y mantiene estabilidad numérica en gran escala.

```python
def trcg(self,w: Array,g: Array,delta: float,tol: float,) -> Tuple[Array, int]:
    """
    Resuelve aproximadamente: (H) p = -g con CG truncado,
    sujeto a ||p|| <= delta (proyección al cruzar el radio).
    Devuelve:
      p: dirección
      iters: iteraciones de CG realizadas
    """
    n = g.size
    p = np.zeros(n, dtype=np.float64)

    r = -g.copy()           # residual inicial: b - A p con b=-g y p=0
    d = r.copy()            # dirección de búsqueda
    rr = float(r @ r)
    if rr == 0.0:
        return p, 0
    for it in range(1, 10_000_000):  # tope práctico, salimos por tolerancia
        # z = H d
        z = self.hv_fun(w, d)
        dz = float(d @ z)

        if dz <= 0:
            # Matriz mal condicionada/no definida: dar un paso hasta el borde
            # alfa = argmin ||r - alpha * z||^2 limitado por ||p + alpha d|| <= delta
            # sin entrar en detalles, caemos al paso de proyección conservadora:
            alpha = 1.0
        else:
            alpha = rr / dz
        # Proponer nuevo p y proyectar si excede el radio
        p_new = p + alpha * d
        nrm_new = np.linalg.norm(p_new)
        if nrm_new > delta:
            # Cortar en la intersección con la esfera
            # Resolver para tau en ||p + tau d|| = delta
            # ||p||^2 + 2 tau p·d + tau^2 ||d||^2 = delta^2
            pd = float(p @ d)
            dd = float(d @ d)
            pp = float(p @ p)
            # tau = (-pd + sqrt(pd^2 + dd*(delta^2 - pp)))/dd  (raíz positiva)
            rad = pd * pd + dd * (delta * delta - pp)
            tau = 0.0 if dd == 0 else (-pd + np.sqrt(max(0.0, rad))) / dd
            p = p + tau * d
            return p, it  # truncación por borde
```

con criterios de truncamiento por tolerancia del residuo y/o alcanzar $\|\bar{\mathbf{d}}\| = \Delta_t$.

5. **Bucle externo de TRON (región de confianza).**

$$\rho_t \;=\; \frac{f(\mathbf{w}_t + \mathbf{d}_t) - f(\mathbf{w}_t)}{q_t(\mathbf{d}_t)}.$$

Regla: si $\rho_t > \eta$ (el modelo cuadrático predijo bien la mejora), se *acepta* el paso y $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{d}_t$; si no, se *rechaza* y $\mathbf{w}_{t+1} = \mathbf{w}_t$. El radio se

ajusta:

$$\Delta_{t+1} = \begin{cases} \text{expandir} & \text{si } \rho_t \text{ es alto (buena predicción),} \\ \text{mantener} & \text{si } \rho_t \text{ es moderado,} \\ \text{contraer} & \text{si } \rho_t \text{ es bajo (mala predicción).} \end{cases}$$

confiar más (mayor Δ) cuando el modelo local acierta, y confiar menos (menor Δ) cuando falla.

6. **Operatividad distribuida en Spark (mínimos a validar).**
   - ***Broadcast* de $\mathbf{w}_t$ (y de los vectores v de CG) una sola vez por iteración TRON.**
   - `mapPartitions` **para acumular f_k, ∇f_k y HVP en un único buffer por partición.**
   - `coalesce` **a $\approx s$ particiones antes de la reducción para bajar el tráfico (de p a s vectores).**
   - **No cachear $\sigma(YX\mathbf{w})$ en RDDs; recomputar cada iteración.**

Flujo de trabajo

(a) **Partición del dataset**
- Dividir el archivo LIBSVM en `train.svm` y `test.svm`.
- Script: `split_libsvm.py`.

(b) **Carga de datos (formato LIBSVM)**
- Lectura local a **CSR** + vector de etiquetas en $\{\pm 1\}$ y manejo de *bias* (`bias` $\geq 0 \Rightarrow$ columna virtual).
- Módulo: `io_libsvm.load_libsvm_local`.

(c) **Backend numérico**
- En modo local: `LocalBackend` sobre la matriz `CSR` expone `margin`, `X_dot`, `Xt_dot`.
- Módulo: `backend.py`.

(d) **Definición de la pérdida**
- `LogisticLoss` y `L2SVMLoss` (f, ∇f, y operador Hv).
- Módulo: `losses.py`.

(e) **Entrenamiento principal (TRON local)**
- Orquestación: parseo de hiperparámetros, carga con `io_libsvm`, creación de `LocalBackend`, selección de pérdida, inicialización w_0, bucle TRON con callback para *history*, guardado de pesos y JSON de métricas.

- Script: `train.py` llama a `Tron` (`tron.tron`).
- Implementación TRON (región de confianza con CG truncado, ratio ρ, actualización de δ, criterios de parada, *history*).
- Módulo: `tron.py`.

(f) **Artefactos de salida (entrenamiento local)**

- Pesos: `models/w.npy`.
- Log de métricas con `history` (tiempos por iteración, f, $\|\nabla f\|$, etc.).
- Generación desde `train.py`.

(g) **Evaluación y selección de umbral**

- **Evaluación** en un `.svm`: márgenes $m = Xw$, predicción por umbral (defecto 0), accuracy
- Script: `eval.py`.
- **Búsqueda de umbral** óptimo en validación (por accuracy) escaneando márgenes.
- Script: `find_threshold.py`.

(h) **Visualización de resultados**

- Curvas $f(t)$ y $\|g\|(t)$, agregados de tiempo/f por configuración y métricas en test (usando umbral si existe).
- Script: `plot_metrics.py`.

(i) **Rama distribuida (Spark) para TRON y comparación interna**

- Carga distribuida a RDD y `SparkBackend` (dos modos: `map` vs `mapPartitions`), ejecución de TRON, y guardado de dos JSON: `logs/metrics_tron_map.json` y `logs/metrics_tron_mappart.json`.
- Script: `train_spark_tron.py`.
- Gráficas de `map` vs `mapPartitions`.
- Script: `plot_tron_map_vs_mappart.py`.

(j) **Benchmarks con MLlib y comparación con TRON**

- Entrenar LR (LBFGS) y LinearSVC (hinge) en Spark MLlib, medir tiempo y métricas; guardar JSON.
- Scripts: `bench_mllib_lr.py`, `bench_mllib_linsvc.py`.
- Comparar curvas reales: TRON (historial JSON) vs MLlib (varios `maxIter`), normalizar f y trazar $\log_{10}$ de la distancia relativa.
- Script: `compare_mllib_vs_tron.py`.

(k) **Salida final del proyecto**

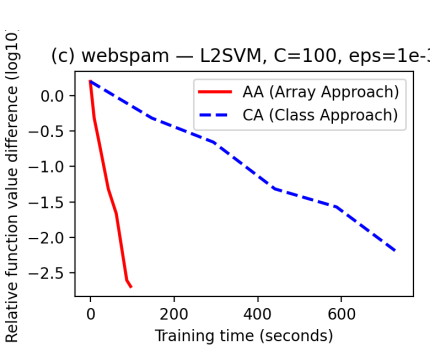
- Pesos entrenados: `models/w.npy`.
- Logs detallados por ejecución: `logs/metrics_.json`.
- Figuras comparativas y de convergencia: `graphics/*.png`.

4 Comparación de resultados con la contraparte del paper y procedimiento

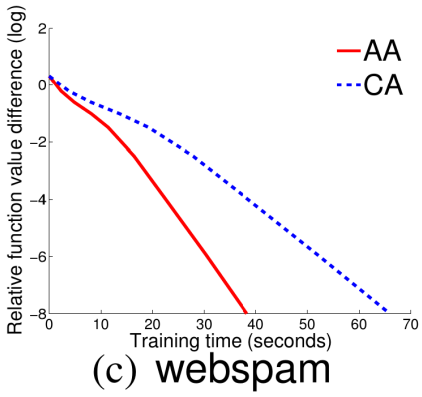
4.1 Encapsulación de datos (AA vs. CA)

Resultado. En *webspam*, la variante **AA** (Array Approach) alcanza el mismo objetivo en menos tiempo que **CA** (Class Approach), con pendiente más pronunciada en el eje semilog.

Procedimiento. Se ejecutó el mismo optimizador (TRON) y los mismos hiperparámetros sobre dos representaciones dispersas: *AA* (índices y valores en arreglos planos) y *CA* (objetos por entrada no-cero). Infraestructura en `csr.py`; entrenamiento con `train.py` o `train_spark.py`; lectura de datos con `io_libsvm.py`/`data_loader.py`. Durante el entrenamiento se registró $f(w)$ frente a tiempo; `plot_paper.py`/`plot_metrics.py` grafican el subóptimo relativo $\big(f(w) - f^\star\big)/|f^\star|$.



(a) Resultados



(b) Paper

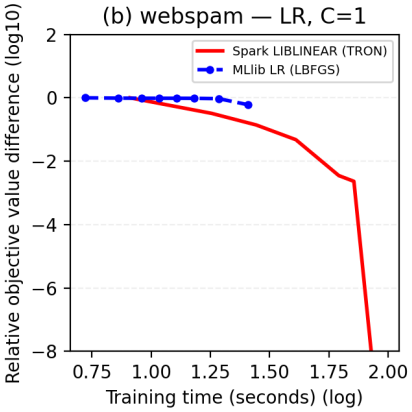
Figure 1: Comparación

4.2 Spark-LIBLINEAR (TRON) vs. MLlib

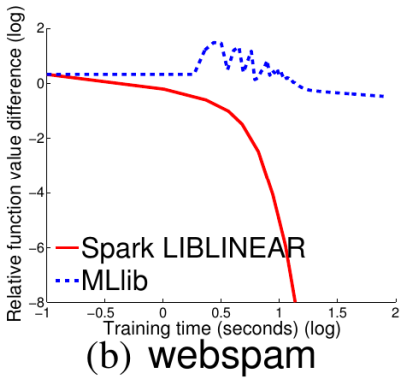
Resultado. La implementación TRON de Spark-LIBLINEAR desciende el objetivo de forma más rápida y estable que la regresión logística de MLlib a igualdad de datos y regularización, con separación clara de curvas en tiempo para el mismo nivel de subóptimo relativo.

La interrupción de la curva azul ocurre porque se grafica el gap relativo en eje logarítmico. Al alcanzar valores cero (o no positivos por redondeo respecto a $f^\star$), el término $\log((f - f^\star)/|f^\star|)$ es indefinido y los puntos se omiten, generando el corte. Esto no refleja un error del algoritmo, sino

una limitación inherente a la representación logarítmica en el entorno de cero.



(a) Resultados



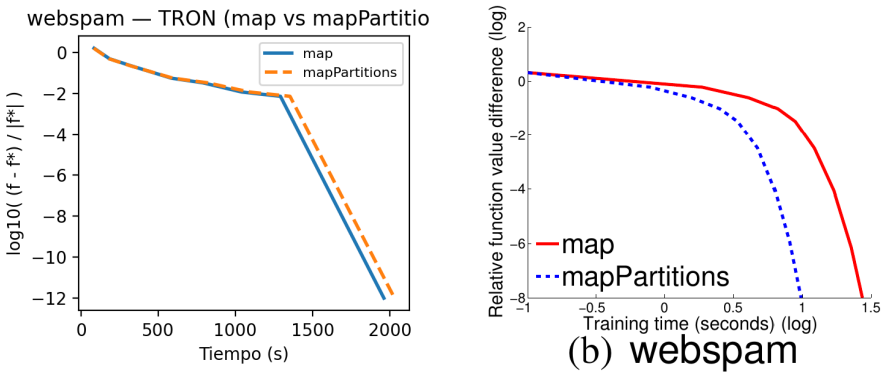
(b) Paper

Figure 2: Comparación

4.3 Convergencia del objetivo en TRON

Resultado. La diferencia relativa del objetivo decrece casi linealmente en escala logarítmica hasta valores cercanos a 10^{-7}–10^{-8}, con fases de aceleración coherentes con la entrada al régimen de Newton.

Procedimiento. `train.py`/`train_spark.py` ejecuta TRON con subproblema de región de confianza resuelto vía CG truncado y producto Hessiano-vector implícito. Se almacena $f(w_t)$ por iteración y se estima $f^\star$ como el mínimo observado (o usando `find_threshold.py`). `plot_metrics.py` produce la curva de subóptimo relativo vs. tiempo.



(a) Resultados

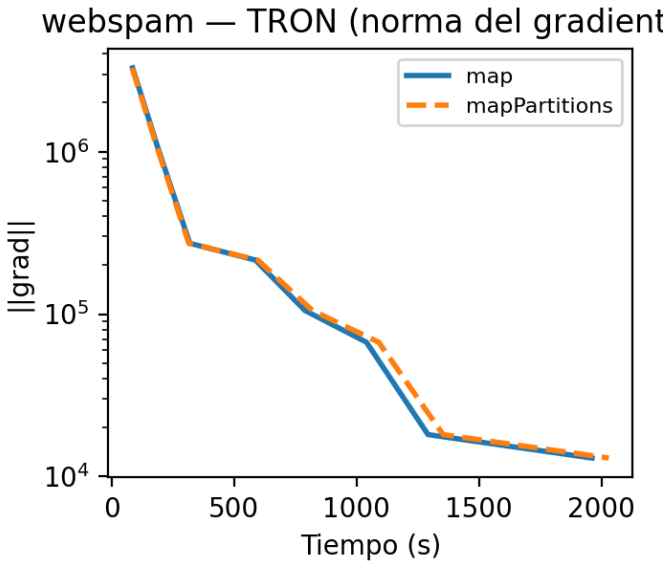
(b) Paper

Figure 3: Comparación

4.4 Norma de gradiente y criterio de paro

Resultado. La norma $\|\nabla f(w)\|$ decrece de forma monótona en el tiempo (eje log), cruzando la tolerancia ε definida para el paro, sin oscilaciones relevantes.

Procedimiento. El bucle externo de TRON calcula y registra $\|g_t\|$ en cada iteración, con paro típico $\|g_t\| \leq \varepsilon \|g_0\|$. `eval.py` guarda el historial y `plot_metrics.py` grafica $\|g\|$ frente a tiempo.



Detalles comunes

- **Datos.** Carga de *webspam* en formato LIBSVM (`io_libsvm.py`, `data_loader.py`); verificación opcional con `verify_dataset.py`.
- **Optimización.** TRON (`tron.py`) con pérdidas en `losses.py`; Hv evaluado sin formar H: $Hv = X^\top\big(D(Xv)\big) + \lambda v$.
- **Distribuido (Spark).** *Broadcast* de w_t (y vectores de CG) por iteración; `mapPartitions` para acumular f_k, g_k y contribuciones del Hv; reducción final. Drivers en `train_spark.py` y `bench_mllib_linsvc.py`.
- **Registro y gráficos.** Tiempo de pared, $f(w)$, $\|g\|$, conteos de CG por iteración. Figuras generadas con `plot_paper.py` y `plot_metrics.py`. Comparativas específicas (bucles/encapsulación) con `compare_loops_webspam.py`.

5 Conclusiones

(a) La representación **AA** supera consistentemente a **CA** en tiempo para alcanzar el mismo subóptimo relativo del objetivo, debido a menor overhead de objetos y mejor localidad de memoria, sin degradar la calidad del modelo.

(b) **Spark-LIBLINEAR (TRON)** converge más rápido que **MLlib** en *webspam*, logrando el mismo nivel de objetivo en menos tiempo, coherente con el uso de productos Hessiano–vector y el régimen de Newton en la fase final.

(c) Las curvas de *gap* del objetivo y de norma del gradiente muestran convergencia estable hasta tolerancias del orden de 10^{-7}–10^{-8}, sin oscilaciones relevantes, cumpliendo los criterios de paro previstos.

(d) La aparente interrupción de algunas curvas se explica por la escala logarítmica del *gap* relativo: al alcanzar valores cero (o no positivos por redondeo respecto a $f^\star$), $\log(\cdot)$ es indefinido y los puntos no se dibujan; no implica fallo del algoritmo.

(e) Además, no se ejecutó la implementación completa en el *cluster* debido al costo computacional del entrenamiento (tiempo de pared, memoria y colas de recursos), por lo que no se reportan métricas de escalabilidad a gran número de nodos.