

In [ ]:

In [ ]:

# Benchmark de RAM: BGE-m3 vs multilingual-e5-small

Medimos consumo real de memoria en cada fase:

1. **Baseline** — RAM antes de cargar nada
2. **Carga del modelo** — cuánta RAM consume tener el modelo en memoria
3. **Encoding** — pico de RAM durante encoding a distintos tamaños de corpus
4. **Scaling** — cómo escala la RAM con corpus de 100, 500, 1000, 5000 documentos

## 1. Setup

```
In [1]: import os, gc, time, tracemalloc
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import psutil

plt.style.use('seaborn-v0_8-whitegrid')
plt.rcParams['figure.figsize'] = (14, 6)

MODELS = {
    'multi-e5-small': {
        'id': 'intfloat/multilingual-e5-small',
        'q_prefix': 'query: ',
        'd_prefix': 'passage: ',
    },
    'BGE-m3': {
        'id': 'BAAI/bge-m3',
        'q_prefix': None,
        'd_prefix': None,
    },
}

def get_process_ram_mb():
    """RAM del proceso actual en MB (RSS)."""
    return psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024)

def get_system_ram():
    """RAM del sistema: total, usada, disponible en MB."""
    m = psutil.virtual_memory()
    return {'total_mb': m.total / (1024**2), 'used_mb': m.used / (1024**2),
```

```

sys_ram = get_system_ram()
print(f'Sistema: {sys_ram["total_mb"]:.0f} MB total, {sys_ram["available_mb"]:.0f} MB disponible')
print(f'Proceso actual: {get_process_ram_mb():.0f} MB')
print(f'Modelos a probar: {list(MODELS.keys())}')

```

Sistema: 31970 MB total, 21550 MB disponible (32.6% usado)

Proceso actual: 179 MB

Modelos a probar: ['multi-e5-small', 'BGE-m3']

## 2. Corpus sintético escalable

Generamos corpus de distintos tamaños a partir de frases semilla en español.

In [2]: **import** random

*# Frases semilla para generar corpus de cualquier tamaño*

```

SEEDS = [
    'Python es un lenguaje de programación de alto nivel conocido por su leg',
    'La fotosíntesis convierte la luz solar en energía química en las célula',
    'El pan de masa madre requiere un cultivo fermentado de harina y agua',
    'El interés compuesto hace crecer los ahorros exponencialmente a largo p',
    'La selva amazónica produce aproximadamente el 20 por ciento del oxígeno',
    'Las redes neuronales profundas aprenden representaciones jerárquicas de',
    'La criptografía asimétrica usa pares de claves pública y privada',
    'Los contenedores Docker encapsulan aplicaciones con todas sus dependenci',
    'La teoría de grafos modela relaciones entre entidades como nodos y aris',
    'Las bases de datos columnares optimizan consultas analíticas sobre gran',
    'El protocolo TCP garantiza la entrega ordenada y confiable de paquetes',
    'La transformada de Fourier descompone señales en sus frecuencias compor',
    'Los microservicios dividen una aplicación en servicios pequeños e indep',
    'El álgebra lineal es fundamental para machine learning y procesamiento',
    'La reacción de Maillard crea el dorado y sabor al cocinar proteínas a a',
    'Los índices invertidos permiten búsqueda full-text eficiente en documen',
    'Las corrientes oceánicas regulan el clima global y los patrones meteoro',
    'El garbage collector libera memoria de objetos que ya no son referencia',
    'La diversificación reduce el riesgo del portafolio distribuyendo invers',
    'Los embeddings representan texto como vectores densos en un espacio sem
]

```

```

def generate_corpus(n):
    """Genera corpus de n documentos variando las frases semilla."""
    random.seed(42)
    suffixes = [
        ' en sistemas modernos', ' según investigaciones recientes',
        ' de manera eficiente', ' con aplicaciones prácticas',
        ' en el contexto actual', ' para usuarios avanzados',
        ' combinando múltiples enfoques', ' optimizando recursos',
        ' a gran escala',
    ]
    corpus = []
    for i in range(n):
        base = SEEDS[i % len(SEEDS)]
        suffix = suffixes[i % len(suffixes)]
        corpus.append(f'{base}{suffix}')
    return corpus

```

```
CORPUS_SIZES = [100, 500, 1000, 5000]
corpora = {n: generate_corpus(n) for n in CORPUS_SIZES}
for n, c in corpora.items():
    print(f'Corpus {n:5d} docs → ejemplo: "{c[0][:70]}..."')
```

Corpus 100 docs → ejemplo: "Python es un lenguaje de programación de alto nivel conocido por su le..."

Corpus 500 docs → ejemplo: "Python es un lenguaje de programación de alto nivel conocido por su le..."

Corpus 1000 docs → ejemplo: "Python es un lenguaje de programación de alto nivel conocido por su le..."

Corpus 5000 docs → ejemplo: "Python es un lenguaje de programación de alto nivel conocido por su le..."

### 3. Benchmark de RAM por modelo

Para cada modelo medimos:

- **RAM al cargar** — delta RSS después de instanciar el modelo
- **RAM pico encoding** — tracemalloc captura el pico durante encode
- **Tiempo de encoding** — por cada tamaño de corpus
- **RAM de embeddings** — cuánto pesan los arrays numpy resultantes

```
In [3]: from sentence_transformers import SentenceTransformer

def benchmark_model(name, config, corpora):
    """Benchmark completo de RAM para un modelo."""
    gc.collect()
    ram_before = get_process_ram_mb()

    # Cargar modelo
    t0 = time.perf_counter()
    model = SentenceTransformer(config['id'], trust_remote_code=True)
    load_time = time.perf_counter() - t0
    ram_after_load = get_process_ram_mb()
    ram_model = ram_after_load - ram_before

    # Parámetros del modelo
    n_params = sum(p.nelement() for p in model[0].auto_model.parameters()) /
    dim = model.get_sentence_embedding_dimension()

    print(f'\n{"=" * 70}')
    print(f'{name} ({n_params:.1f}M params, dim={dim})')
    print(f'{"=" * 70}')
    print(f'RAM baseline: {ram_before:.0f} MB')
    print(f'RAM después de carga: {ram_after_load:.0f} MB (modelo = +{ram_model:.0f} MB)')
    print(f'Tiempo de carga: {load_time:.1f}s')

    results = []
    prefix = config['d_prefix']

    for corpus_size, corpus in corpora.items():
        gc.collect()
```

```

texts = [f'{prefix}{doc}' for doc in corpus] if prefix else corpus

# Medir pico de RAM con tracemalloc
tracemalloc.start()
ram_pre_encode = get_process_ram_mb()
t0 = time.perf_counter()

embeddings = model.encode(texts, normalize_embeddings=True,
                           show_progress_bar=False, batch_size=32)

encode_time = time.perf_counter() - t0
ram_post_encode = get_process_ram_mb()
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

# Tamaño de los embeddings en MB
emb_size_mb = embeddings.nbytes / (1024 * 1024)

result = {
    'model': name,
    'corpus_size': corpus_size,
    'dim': dim,
    'ram_model_mb': round(ram_model),
    'ram_pre_encode_mb': round(ram_pre_encode),
    'ram_post_encode_mb': round(ram_post_encode),
    'ram_delta_encode_mb': round(ram_post_encode - ram_pre_encode),
    'tracemalloc_peak_mb': round(peak / (1024 * 1024), 1),
    'emb_size_mb': round(emb_size_mb, 2),
    'encode_time_s': round(encode_time, 3),
    'docs_per_sec': round(corpus_size / encode_time, 1),
}
results.append(result)

print(f' {corpus_size:5d} docs | encode={encode_time:6.2f}s ({result[
    f'RAM delta=+{result["ram_delta_encode_mb"]:4d} MB | peak(trac
    f'embeddings={emb_size_mb:.2f} MB')

# Limpiar modelo
del model
gc.collect()
ram_after_cleanup = get_process_ram_mb()
print(f' RAM después de cleanup: {ram_after_cleanup:.0f} MB (liberados

return results

# Ejecutar benchmarks secuencialmente
all_results = []
for name, config in MODELS.items():
    all_results.extend(benchmark_model(name, config, corpora))

```

Warning: You are sending unauthenticated requests to the HF Hub. Please set a HF\_TOKEN to enable higher rate limits and faster downloads.

Loading weights: 0%| | 0/199 [00:00<?, ?it/s]

## BertModel LOAD REPORT from: intfloat/multilingual-e5-small

Key	Status	
-----+-----+--		
embeddings.position_ids	UNEXPECTED	

### Notes:

- UNEXPECTED: can be ignored when loading from different task/architecture; not ok if you expect identical arch.

=====

multi-e5-small (117.7M params, dim=384)

=====

RAM baseline: 881 MB

RAM después de carga: 1565 MB (modelo = +684 MB)

Tiempo de carga: 5.5s

100 docs	encode= 0.28s ( 356.2 docs/s)	RAM delta=+ 191 MB	peak(t racemalloc)= 0.2 MB	embeddings=0.15 MB
500 docs	encode= 0.28s ( 1791.3 docs/s)	RAM delta=+ 2 MB	peak(t racemalloc)= 0.9 MB	embeddings=0.73 MB
1000 docs	encode= 0.55s ( 1807.4 docs/s)	RAM delta=+ 2 MB	peak(t racemalloc)= 1.8 MB	embeddings=1.46 MB
5000 docs	encode= 2.61s ( 1915.6 docs/s)	RAM delta=+ 15 MB	peak(t racemalloc)= 9.1 MB	embeddings=7.32 MB

RAM después de cleanup: 1757 MB (liberados ~-192 MB)

Loading weights: 0%| | 0/391 [00:00<?, ?it/s]

=====

BGE-m3 (567.8M params, dim=1024)

=====

RAM baseline: 1756 MB

RAM después de carga: 1866 MB (modelo = +110 MB)

Tiempo de carga: 10.0s

100 docs	encode= 0.52s ( 192.5 docs/s)	RAM delta=+ 0 MB	peak(t racemalloc)= 0.5 MB	embeddings=0.39 MB
500 docs	encode= 1.00s ( 501.7 docs/s)	RAM delta=+ 0 MB	peak(t racemalloc)= 2.2 MB	embeddings=1.95 MB
1000 docs	encode= 1.87s ( 535.7 docs/s)	RAM delta=+ 0 MB	peak(t racemalloc)= 4.3 MB	embeddings=3.91 MB
5000 docs	encode= 9.01s ( 554.9 docs/s)	RAM delta=+ 24 MB	peak(t racemalloc)= 21.3 MB	embeddings=19.53 MB

RAM después de cleanup: 1833 MB (liberados ~33 MB)

## 4. Tabla resumen

```
In [4]: df = pd.DataFrame(all_results)
display_cols = ['model', 'corpus_size', 'dim', 'ram_model_mb', 'ram_delta_er
               'tracemalloc_peak_mb', 'emb_size_mb', 'encode_time_s', 'docs
df_display = df[display_cols].copy()
df_display.columns = ['Modelo', 'Docs', 'Dim', 'RAM modelo (MB)', 'RAM delta
                    'Peak tracemalloc (MB)', 'Embeddings (MB)', 'Tiempo (
df_display.style.background_gradient(subset=['RAM modelo (MB)', 'RAM delta e
```

Out[4]:

	Modelo	Docs	Dim	RAM modelo (MB)	RAM delta encode (MB)	Peak tracemalloc (MB)	Embeddings (MB)	Tiempo (s)
0	multi-e5-small	100	384	684	191	0.200000	0.150000	0.281000
1	multi-e5-small	500	384	684	2	0.900000	0.730000	0.279000
2	multi-e5-small	1000	384	684	2	1.800000	1.460000	0.553000
3	multi-e5-small	5000	384	684	15	9.100000	7.320000	2.610000
4	BGE-m3	100	1024	110	0	0.500000	0.390000	0.520000
5	BGE-m3	500	1024	110	0	2.200000	1.950000	0.997000
6	BGE-m3	1000	1024	110	0	4.300000	3.910000	1.867000
7	BGE-m3	5000	1024	110	24	21.300000	19.530000	9.011000

## 5. Visualizaciones

```
In [5]: fig, axes = plt.subplots(1, 3, figsize=(18, 6))
        colors = {'multi-e5-small': '#3498db', 'BGE-m3': '#e74c3c'}

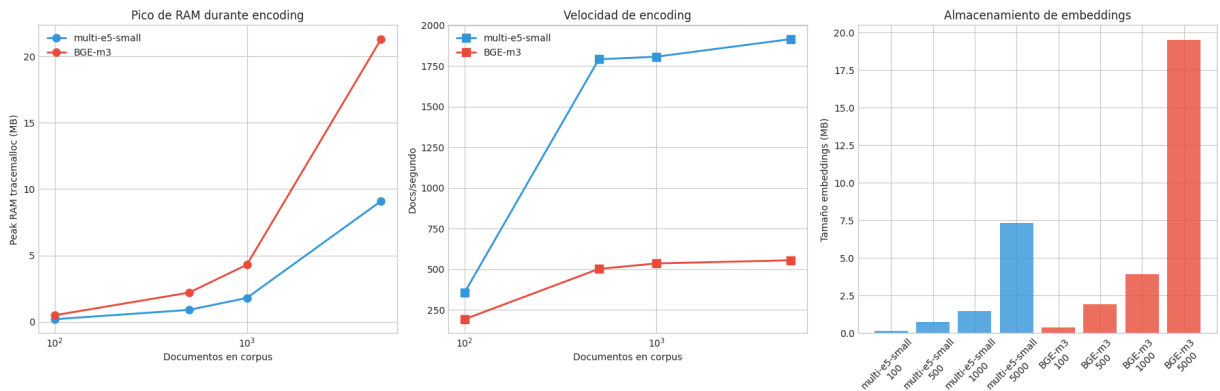
        # Gráfico 1: RAM pico vs tamaño de corpus
        ax = axes[0]
        for name in MODELS:
            subset = df[df['model'] == name]
            ax.plot(subset['corpus_size'], subset['tracemalloc_peak_mb'], 'o-',
                    color=colors[name], label=name, linewidth=2, markersize=8)
        ax.set_xlabel('Documentos en corpus')
        ax.set_ylabel('Peak RAM tracemalloc (MB)')
        ax.set_title('Pico de RAM durante encoding')
        ax.legend()
        ax.set_xscale('log')

        # Gráfico 2: Throughput (docs/s) vs tamaño de corpus
        ax = axes[1]
        for name in MODELS:
            subset = df[df['model'] == name]
            ax.plot(subset['corpus_size'], subset['docs_per_sec'], 's-',
                    color=colors[name], label=name, linewidth=2, markersize=8)
        ax.set_xlabel('Documentos en corpus')
        ax.set_ylabel('Docs/segundo')
        ax.set_title('Velocidad de encoding')
        ax.legend()
        ax.set_xscale('log')
```

# Gráfico 3: Tamaño de embeddings vs corpus

```
ax = axes[2]
for name in MODELS:
    subset = df[df['model'] == name]
    ax.bar([f'{name}\n{n}' for n in subset['corpus_size']],
           subset['emb_size_mb'], color=colors[name], alpha=0.8)
ax.set_ylabel('Tamaño embeddings (MB)')
ax.set_title('Almacenamiento de embeddings')
ax.tick_params(axis='x', rotation=45)

plt.tight_layout()
plt.show()
```



## 6. Comparación directa: coste por documento

```
In [6]: # Comparación lado a lado para el corpus más grande
biggest = df[df['corpus_size'] == max(CORPUS_SIZES)]

fig, ax = plt.subplots(figsize=(10, 5))
metrics = ['ram_model_mb', 'tracemalloc_peak_mb', 'emb_size_mb']
labels = ['RAM modelo\n(MB)', 'Peak RAM encode\n(MB)', 'Embeddings\n(MB)']
x = np.arange(len(labels))
width = 0.35

for i, (_, row) in enumerate(biggest.iterrows()):
    vals = [row[m] for m in metrics]
    offset = -width/2 + i * width
    bars = ax.bar(x + offset, vals, width, label=row['model'], color=list(cc))
    for bar, val in zip(bars, vals):
        ax.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 5,
                f'{val:.0f}', ha='center', va='bottom', fontsize=10, fontwei

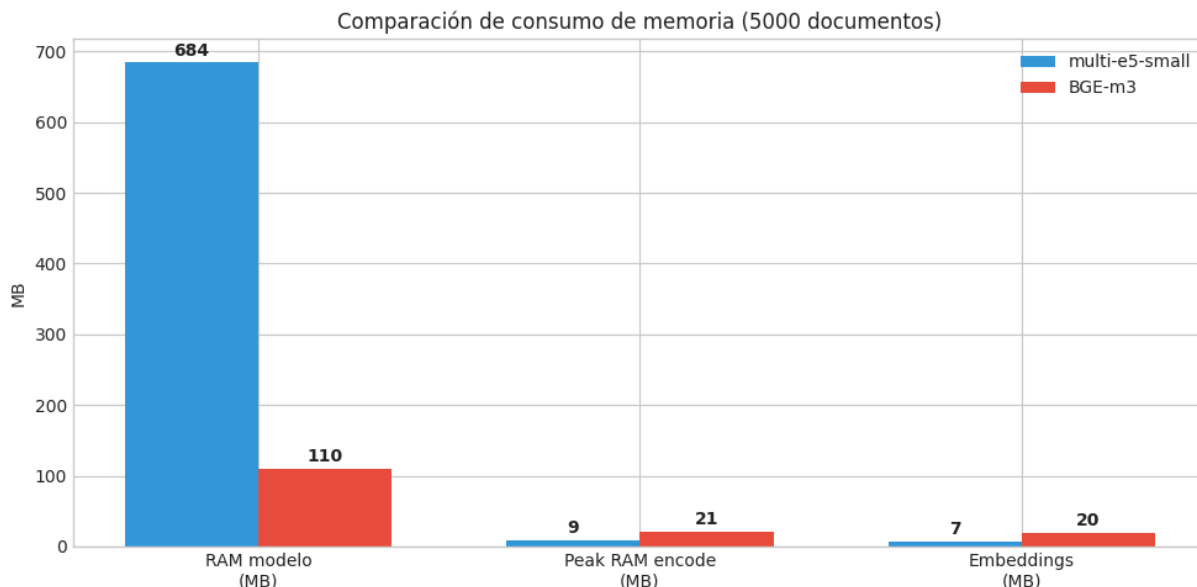
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.set_ylabel('MB')
ax.set_title(f'Comparación de consumo de memoria ({max(CORPUS_SIZES)} docume
ax.legend()
plt.tight_layout()
plt.show()

# Resumen numérico
print(f'\n{"Métrica":30s} | {"multi-e5-small":>15s} | {"BGE-m3":>15s} | {"Ra
```

```

print('-' * 78)
for _, row_e5 in biggest[biggest['model'] == 'multi-e5-small'].iterrows():
    for _, row_bge in biggest[biggest['model'] == 'BGE-m3'].iterrows():
        for metric, label in [('ram_model_mb', 'RAM modelo (MB)'),
                               ('tracemalloc_peak_mb', 'Peak RAM encode (MB)'),
                               ('emb_size_mb', f'Embeddings {max(CORPUS_SIZE)}, (MB)'),
                               ('encode_time_s', f'Encode {max(CORPUS_SIZES)} docs (s)'),
                               ('docs_per_sec', 'Throughput (docs/s)')]:
            v_e5 = row_e5[metric]
            v_bge = row_bge[metric]
            ratio = v_bge / v_e5 if v_e5 > 0 else float('inf')
            print(f'{label:30s} | {v_e5:>15.1f} | {v_bge:>15.1f} | {ratio:>7.1f}')

```



Métrica	multi-e5-small	BGE-m3	Ratio
RAM modelo (MB)	684.0	110.0	0.2
Peak RAM encode (MB)	9.1	21.3	2.3
Embeddings 5000 docs (MB)	7.3	19.5	2.7
Encode 5000 docs (s)	2.6	9.0	3.5
Throughput (docs/s)	1915.6	554.9	0.3