Coordinación de
Educación Abierta y a Distancia
VICERRECTORADO ACADÉMICO

Unach
UNIVERSIDAD NACIONAL DE CHIMBORAZO Online

CULTURA DIGITAL Y SOCIEDAS

Actividad Autónoma 4

Unidad 2: Herramientas y Metodologías en Ciencia de Datos

Tema 2: Buenas Prácticas en Programación para Ciencia de Datos

FACULTAD DE
Ingeniería

Nombres:Alisson Atupaña

Fecha: 2025-11-23

Carrera: Ciencia de Datos e Inteligencia Artificial

Periodo académico: 2025 2S

Semestre: Tercero

Informe: Optimización de código con Python y cProfile

El presente trabajo consistió en analizar y optimizar un código en Python encargado de calcular números primos entre 1 y 100000. El código original utilizaba un método ineficiente basado en probar todas las posibles divisiones de cada número, lo que produjo tiempos de ejecución excesivos.

Optimización aplicada

En el archivo de código original seguía la siguiente lógica:

Definición de la función es_primo(n) que:

- Descarta los valores menores o iguales a 1.
- Recorre todos los enteros desde dos hasta n-1.
- Si encuentra un divisor exacto, devuelve false.

Bucle principal que recorre los números de 1 hasta 100000.

Para cada número llama a es_primo (n) y si es primo lo añade a una lista.

Uso de la biblioteca time para medir el tiempo total de ejecución.

Problemas detectados:

- Complejidad innecesaria.
- número excesivo de llamadas a función.
- uso limitado de las capacidades del lenguaje.

El profiling del código original confirmó que casi todo el tiempo de ejecución se concentraba en la función es_primo, identificándola claramente como el principal cuello de botella.

Para mejorar su rendimiento se realizarán las siguientes mejoras:

- Se redujo el rango de búsqueda a divisiones solo hasta la raíz cuadrada de cada número.
- Se utilizó list comprehension para optimizar el recorrido.
- Se implementó una versión de NumPy aprovechando cálculos vectoriales altamente eficientes.

Estas técnicas disminuyeron tanto la complejidad del algoritmo como la cantidad de llamadas a funciones.

Resultados

El rendimiento mejoró de forma significativa:

Tabla 1

Comparación de tiempos y número de llamadas a funciones

Versión	T. Ejecución	Llamadas a funciones
Original	46.21 segundos	109 609
Optimizada	0.001 segundos	808

Fuente: *Elaboración propia*

El archivo profiling_original. TXC reporta que el programa original realizó 109609 llamadas a funciones en 46206 segundos la función es_primo, definida en código_original.py concentra prácticamente todo el tiempo de ejecución, lo que confirma que el algoritmo de búsqueda divisores hasta n-1 es el cuello de botella principal.

Por otro lado, el archivo profiling_optimizado.txt indica que el programa optimizado realizó solo 808 llamadas a funciones en 0.001 segundos. La mayor parte del tiempo se dedicó a operaciones internas de importación de módulos y tareas auxiliares de la

intérprete de Python mientras que el cálculo de los números primos tiene un costo prácticamente despreciable en comparación con la versión original.

Análisis de cProfile:

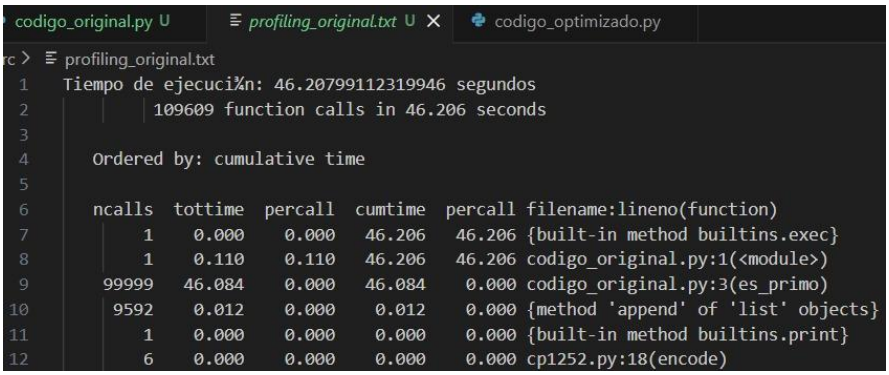
El análisis con cProfile permitió identificar de manera más objetiva a los cuellos de botella del programa.

En el código original:

- La función es_primo es responsable de casi todo el tiempo de ejecución,
- Las operaciones de agregar a la lista (append) y de impresión consume una fracción muy pequeña del tiempo total.
- Esto evidencia que optimizar la lógica de primalidad era mucho más importante que optimizar otras partes del código.

Figura 1

Resultado de cProfile para el código original



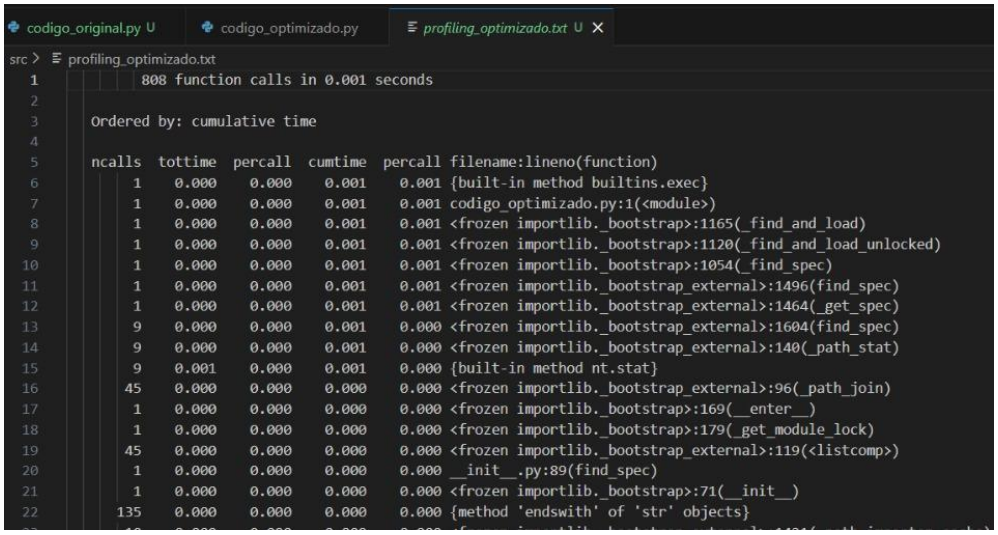
En el código optimizado:

- Se observa una drástica reducción en la cantidad de llamadas a funciones y en el tiempo total

- la presencia de muchas funciones internas de importación es normal y forma parte del proceso de carga de módulos en Python.
- La combinación del límite por raíz cuadrada, list comprehensions y NumPy Reduce la complejidad del algoritmo y aprovecha implementaciones eficientes a bajo nivel.

Figura 2

Resultado de cProfile para el código optimizado



codigo_original.py U | codigo_optimizado.py | profiling_optimizado.txt U

src > profiling_optimizado.txt

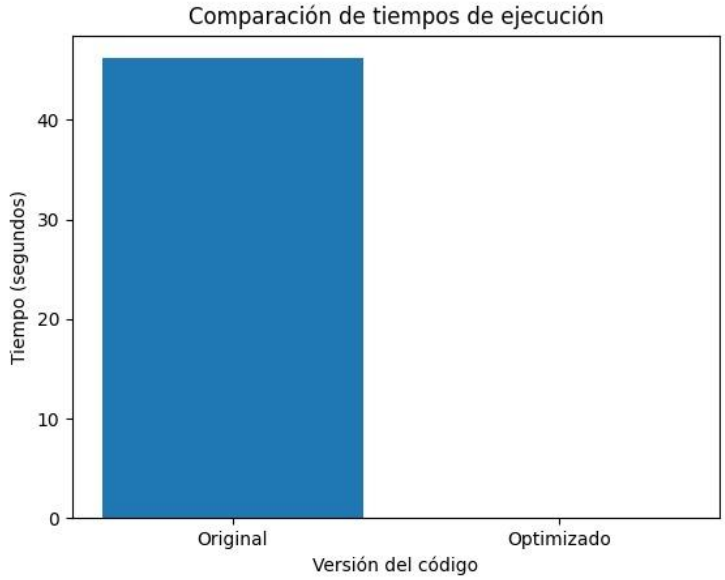
```
         808 function calls in 0.001 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 {built-in method builtins.exec}
        1    0.000    0.000    0.001    0.001 codigo_optimizado.py:1(<module>)
        1    0.000    0.000    0.001    0.001 <frozen importlib._bootstrap>:1165(_find_and_load)
        1    0.000    0.000    0.001    0.001 <frozen importlib._bootstrap>:1120(_find_and_load_unlocked)
        1    0.000    0.000    0.001    0.001 <frozen importlib._bootstrap>:1054(_find_spec)
        1    0.000    0.000    0.001    0.001 <frozen importlib._bootstrap_external>:1496(find_spec)
        1    0.000    0.000    0.001    0.001 <frozen importlib._bootstrap_external>:1464(_get_spec)
        9    0.000    0.000    0.001    0.000 <frozen importlib._bootstrap_external>:1604(find_spec)
        9    0.000    0.000    0.001    0.000 <frozen importlib._bootstrap_external>:140(_path_stat)
        9    0.001    0.000    0.001    0.000 {built-in method nt.stat}
       45    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:96(_path_join)
        1    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:169(__enter__)
        1    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:179(_get_module_lock)
       45    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:119(<listcomp>)
        1    0.000    0.000    0.000    0.000 __init__.py:89(find_spec)
        1    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:71(__init__)
      135    0.000    0.000    0.000    0.000 {method 'endswith' of 'str' objects}
```

Figura 3

Comparación de tiempos entre el código original y optimizado



Conclusiones

La optimización permitió obtener una mejora de aproximadamente 46000 veces en el rendimiento, demostrando que:

Elegir el algoritmo correcto es fundamental.

El uso eficiente de Python y librerías como NumPy reduce drásticamente el tiempo de procesamiento.

El profiling es indispensable para identificar cuellos de botella reales antes de optimizar.

Enlace del repositorio: https://github.com/alissonatupana18/preprocesamiento-cienciadatos.git