Balanceo de cargas por medio de algoritmos genéticos

Autor: Gabriel Melchor Campos

Ingeniero en computación Inteligente

Análisis

El primer paso es leer el dataset “Suicide_Detection.csv” para extraer los datos, esto se logró mediante la librería de pandas.

Después preprocesamos el texto de nuestro csv, de forma que le quitamos los signos de puntuación, los emojis, las stopwords y lemantizamos. De tal forma que quede limpio y listo para se utilizado para el entrenamiento del word2vec.

Out[8]:

	Unnamed: 0	text	class
0	2	ex wife threatening suiciderecently i left my ...	suicide
1	3	am i weird i don t get affected by compliments...	non-suicide
2	4	finally 2020 is almost over so i can never hea...	non-suicide
3	8	i need helpjust help me im crying so hard	suicide
4	9	i m so losthello my name is adam 16 and i ve b...	suicide
5	11	honetly idki dont know what im even doing here...	suicide
6	12	trigger warning excuse for self inflicted bur...	suicide
7	13	it ends tonight i can t do it anymore i quit	suicide
8	16	everyone wants to be edgy and it s making me s...	non-suicide
9	18	my life is over at 20 years oldhello all i am ...	suicide

Al analizar el texto de ‘text’ notamos que había muchos símbolos y etiquetas HTML que no queremos que se tomen en cuenta en la evaluación y que podrían alterar la precisión del modelo, por lo cual se les hizo una limpieza utilizando la librería re una función que nos retorna el texto ya depurado.

Elegimos utilizar Word2Vec con estos clasificadores debido a las siguientes razones:

1. Captura el contexto y la semántica: A diferencia de TF-IDF, que se basa únicamente en la frecuencia de las palabras, Word2Vec captura la información contextual y las relaciones semánticas entre las palabras. Esto puede mejorar la precisión de los clasificadores al considerar la estructura y el contenido del texto.
2. Representación de baja dimensión: Word2Vec genera representaciones vectoriales de palabras en un espacio de menor dimensión en comparación con TF-IDF, lo que puede facilitar el entrenamiento y la eficiencia de algunos algoritmos de clasificación.
3. Transferibilidad: Los modelos Word2Vec pre-entrenados pueden ser fácilmente transferidos a diferentes conjuntos de datos y tareas, lo que

permite aprovechar el conocimiento previo y reducir el tiempo de entrenamiento.

Al utilizar Word2Vec con los clasificadores KNN, Random Forest y AdaBoosting, nuestro equipo pudo abordar eficazmente el problema de clasificación del texto al capturar la información contextual y las relaciones semánticas entre las palabras. La combinación de Word2Vec con estos clasificadores nos permitió explorar diferentes enfoques y encontrar el método más adecuado para nuestro problema específico.

Así se vería un ejemplo del funcionamiento del word2vec:

```
Vectores para los primeros 5 datos en el conjunto de entrenamiento:
Dato 1 - Vector: [-0.16432947 -0.17327309 -0.61434263  0.44026953  0.05820833  0.784133
 -0.35785967 -0.68639666  0.1822196   0.9459795  -0.5712038  -0.25837627
  0.35807592  0.2498178   0.6153931  -0.29495803  0.28817937  0.58066434
 -0.44355023  0.3635445  -0.8662003  -0.4629706   0.54392666 -0.4141743
 -0.41987476 -0.5765817   1.0110871   0.89348525  0.56611687 -0.45833626
  0.20740475 -1.0060531   0.09150474  0.33041313  0.39930576 -1.0404369
 -0.5319017   0.69356436  0.00395907  0.42667395 -0.19017701 -0.1821837
 -0.22136064  0.2658463  -0.08173977  0.26098213  0.2808848  -0.21488668
 -0.48467308 -0.2611486   0.6347406  -0.3563019   0.7472623   0.34114966
  0.48987514  0.3460893   0.1331067   0.5580865   0.19045456 -0.6556156
  0.0277957  -0.3206683  -0.02884484 -0.9468036  -0.607926   -0.2696667
  0.05238795  0.27230403 -0.0828352   0.23058662  0.8026477  -0.48250702
 -0.12473752  0.09020808 -0.46853843  0.12047365 -0.19669238  0.84905964
  0.07071456 -0.276401   -0.4172953   0.09691642  0.53442067 -0.90329075
 -0.4105015   0.48968592 -0.7448421  -0.30370027 -0.5872885   0.41867283
 -0.0614573  -0.6708838  -1.1273277   0.31809303 -0.5581366  -0.01656103
  0.33612555  1.1873703  -0.93437904 -0.26279223]
```

El equilibrio de carga consiste en dividir un programa en tareas más pequeñas que puedan ejecutarse simultáneamente y asignar a cada una de estas tareas un recurso, como un procesador.

El trabajo que aquí se propone (tomando como base el articulo denominado: Observations on Using Genetic Algorithms for Dynamic Load-Balancing.pdf) investiga como puede emplearse un algoritmo genético para resolver el problema de equilibrio de carga. Se desarrolla un algoritmo de equilibrio de carga mediante el cual las asignaciones de tareas optimas pueden evolucionar durante el funcionamiento del sistema.

Este algoritmo está diseñado esencialmente para repartir equitativamente la carga entre los procesadores y maximizar su utilización minimizando el tiempo total de ejecución de las tareas.

La información de carga está en constante actualización para que el mecanismo sea eficaz. Por tanto, aquí implementamos un AG para planificar la carga de 3600 twitts en 8 procesadores, donde mostramos una tabla en donde se ve la carga que tiene cada procesador.

A diferencia de la anterior entrega, esta vez nuestros cromosomas son de forma entera.

Suponiendo que estamos trabajando con 4 procesadores, podemos observar el análisis de nuestro algoritmo de balanceo de cargas.

Nuestros cromosomas son posibles soluciones de tipo entero, el cual representaran cargas junto el algoritmo de la ventana deslizante:

Cromosoma 1: [1, 2, 3, 4, 5, 6]

Cromosoma 2: [6, 5, 4, 3, 2, 1]

Cromosoma 3: [2, 4, 1, 5, 3, 6]

Cromosoma 4: [1, 3, 5, 2, 4, 6]

Cromosoma 5: [4, 3, 1, 2, 6, 5]

Cromosoma 6: [6, 1, 5, 2, 4, 3]

Con ayuda de nuestra tabla deslizante:

			Definimos 6 espacios para nuesta					
			ventana					
			Slide window					
			4	11	16	7	9	3
		Cromosoma	1	2	3	4	5	6
	P0	Espera						
	P1	Espera						
	P2	Espera						
	P3	Espera						

Se asignan las tareas por round robin, pasamos al maxspan, que es nuestra carga máxima y calculamos la utilización media.

$$P_X\,(utilización) = P_X\,(tiempo\ de\ finalización)/maxspan$$

Definimos 6 espacios para nuesta ventana
Slide window

		4	11	16	7	9	3		$P_X\,(utilización) = P_X\,(tiempo\ de\ finalización)/maxspan$
Cromosoma		1	2	3	4	5	6		utilizacion media
P0	4	9						13	0.8125
P1	11							11	0.6875
P2	16							16	1
P3	7	3						10	0.625
								Maxspan=	16

Por tanto, pasamos a nuestro cálculo del fitness.

$$fitness = \frac{1}{maxspan} \times Utilización\ de\ la\ energía \times \frac{\#\ colas\ aceptables}{\#\ procesadores}.$$

Definimos 6 espacios para nuesta ventana
Slide window

$P_X\,(utilización) = P_X\,(tiempo\ de\ finalización)/maxspan$

		4	11	16	7	9	3			
Cromosoma		1	2	3	4	5	6		utilizacion media	Fitness
P0	4	9						13	0.8125	0.05078125
P1	11							11	0.6875	0.04296875
P2	16							16	1	0.0625
P3	7	3						10	0.625	0.0390625
								Maxspan=	16	

$$fitness = \frac{1}{maxspan} \times Utilización\ de\ la\ energía \times \frac{\#\ colas\ aceptables}{\#\ procesadores}.$$

Selección, cruce y mutación

Selección por Ruleta:

- La selección por ruleta es un método de selección de cromosomas en un algoritmo genético basado en probabilidades.

- Cada cromosoma en la población recibe una probabilidad proporcional a su aptitud.
- Se eligen cromosomas aleatoriamente según estas probabilidades. Los cromosomas más aptos tienen una mayor probabilidad de ser seleccionados.

Cruce PMX (Partial-Mapped Crossover):

- El cruce PMX implica tomar dos cromosomas padres y crear dos descendientes.
- Supongamos que tomamos dos padres, por ejemplo, "Cromosoma 1" y "Cromosoma 2", y seleccionamos puntos de corte aleatorios, digamos entre el tercer y el quinto espacio.
- Luego, copiamos la sección entre los puntos de corte del primer padre al primer descendiente y del segundo padre al segundo descendiente.

Mutación por Permutación:

- La mutación por permutación implica cambiar aleatoriamente la posición de uno o más elementos en un cromosoma.
- Supongamos que mutamos "Cromosoma 3". Podríamos intercambiar dos elementos aleatorios, por ejemplo, cambiar "2 4" a "4 2".

Ejemplo de lo anteriormente mencionado con nuestro algoritmo:

Tomemos de ejemplo los siguientes cromosomas:

Cromosoma 1: [1, 2, 3, 4, 5, 6]

Cromosoma 2: [6, 2, 3, 1, 4, 5]

Selección por ruleta: Supongamos que hemos calculado las probabilidades de selección y hemos generado un numero aleatorio que selecciona el cromosoma que tomamos de ejemplo (los más aptos).

Cruce por PMX: Supongamos que tomamos "Cromosoma 1" y "Cromosoma 2" como padres y seleccionamos puntos de corte entre el segundo y el cuarto espacio.

Cromosoma 1: [1, 2, 3, 4, 5, 6]

Cromosoma 2: [6, 2, 3, 1, 4, 5]

Lo cual puede verse asi:

Cromosoma 1: [?, ? , 3, 4, 5, ?]

Cromosoma 2: [?, ?, 3, 1, 4, ?]

Copia elementos no mapeados: Ahora, copia los elementos que no se han mapeado de los padres originales a los descendientes:

Para este caso seria 6 y 2 los no mapeados.

Cromosoma 1: [6, 2, 3, 4, 5, ?]

Cromosoma 2: [6, 2, 3, 1, 4, ?]

Rellenamos con los números faltantes respectivamente para cada cromosoma y obtenemos nuestros descendientes o hijos:

Hijo 1: [6, 2, 3, 4, 5, 1]

Hijo 2: [6, 2, 3, 1, 4, 5]

Mutación por permutación: Por ejemplo, mutemos el hijo 1, cambiando la cadena por medio de una permutación:

Hijo 1 (sin mutar): [6, 2, 3, 4, 5, 1]

Hijo 1 (mutado): [2, 3, 1, 5, 6,4]

Estas operaciones se aplican en un algoritmo genético para explorar diferentes soluciones y encontrar la mejor solución en un espacio de búsqueda.

Implementación

```
]: import pandas as pd
   import re
   import random
   from unicodedata import normalize
   from nltk.corpus import stopwords
   from nltk import word_tokenize
   from gensim.models import Word2Vec
   import numpy as np
   import time
   from multiprocessing import Pool
```

Incluimos las librerías para poder realizar el objetivo del proyecto etapa 2.

```
l7]: #Importamos pandas
     import pandas as pd
     #Leemos el dataset
     df = pd.read_csv('Suicide_Detection.csv')
     #Mostramos las primeras filas
     df.head(10)
```

l7]:

	Unnamed: 0	text	class
0	2	Ex Wife Threatening SuicideRecently I left my ...	suicide
1	3	Am I weird I don't get affected by compliments...	non-suicide
2	4	Finally 2020 is almost over... So I can never ...	non-suicide
3	8	i need helpjust help me im crying so hard	suicide
4	9	I'm so lostHello, my name is Adam (16) and I'v...	suicide
5	11	Honetly idkI dont know what im even doing here...	suicide
6	12	[Trigger warning] Excuse for self inflicted bu...	suicide
7	13	It ends tonight.I can't do it anymore. \nI quit.	suicide
8	16	Everyone wants to be "edgy" and it's making me...	non-suicide
9	18	My life is over at 20 years oldHello all. I am...	suicide

Leemos el dataset y mostramos las primeras 10 filas.

Limpiamos los datos, pero eso es mas de la etapa 1, no daremos mucho detalle.

Limpieza de datos

In [18]:
```
#Funcion auxiliar de limpieza
import re
def Limpiador(text):
    # Remover tags de html
    text = re.sub('<[^<]*>','',text)

    # Almacenar temporalmente los emoticons
    emoticons = ''.join(re.findall('[:;=]-+[\)\(pPD]+',text))

    # Elimine los caracteres que no son palabras y combinar los emoticones
    text = re.sub('\W+',' ',text.lower()) + emoticons.replace('-','')

    return text
```

In [19]:
```
#Funcion para remover emojis
def RemoverEmoji(text):
    emoji_pattern = re.compile("["
                           u"\U0001F600-\U0001F64F"  # emoticons
                           u"\U0001F300-\U0001F5FF"  # symbols & pictographs
                           u"\U0001F680-\U0001F6FF"  # transport & map symbols
                           u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                           u"\U00002702-\U000027B0"
                           u"\U000024C2-\U0001F251"
                           "]+", flags=re.UNICODE)
    return emoji_pattern.sub(r'', text)
```

In [20]:
```
#Aplicamos los limpiadores
df['text'] = df['text'].apply(Limpiador)
```

In [21]:
```
#Remover Emojis
df['text']=df['text'].apply(lambda x: RemoverEmoji(x))
```

In [22]:
```
#Normalizar a utf-8, remover acentos
from unicodedata import normalize
```

En este bloque de código aplicamos el balanceo de las cargas, la slide window, distribución de la carga y todo lo que conlleva el algoritmo genético, explicado anteriormente en la parte de análisis.

[31]:
```
%%time
def cargas(sumaFinal, tamaño_ventana):
    numeros_random = []
    suma = 0
    while suma < sumaFinal:
        numeroA = random.randint(1, 30)  # Modificado para que sea de 1 a 30
        if suma + numeroA > sumaFinal:
            numeros_random.append(sumaFinal - suma)
            break
        else:
            numeros_random.append(numeroA)
            suma = sum(numeros_random)
    if len(numeros_random) % tamaño_ventana != 0:
        resto = len(numeros_random) % tamaño_ventana
        if resto != 0:
            elementos_faltantes = tamaño_ventana - resto
            numeros_random.extend([0] * elementos_faltantes)
    return numeros_random

def generar_poblacion(num_individuos, tamaño_ventana):
    poblacion = []
    for _ in range(num_individuos):
        individuo = [random.uniform(1, tamaño_ventana) for _ in range(tamaño_ventana)]
        poblacion.append(individuo)
    return poblacion

def calcular_fitness(asignacion_procesos, maxspan):
    utilizacion_procesadores = []
    for procesos in asignacion_procesos.values():
        carga_total = sum(procesos)
        utilizacion = carga_total / maxspan
        utilizacion_procesadores.append(utilizacion)

    apu = sum(utilizacion_procesadores) / len(utilizacion_procesadores)
    fitness = (1 / maxspan) * apu
    return fitness
```

```python
def seleccion_por_ruleta(poblacion, fitness_poblacion):
    total_fitness = sum(fitness_poblacion)
    probabilidad_seleccion = [fit / total_fitness for fit in fitness_poblacion]
    papa1 = random.choices(poblacion, weights=probabilidad_seleccion)[0]
    papa2 = random.choices(poblacion, weights=probabilidad_seleccion)[0]
    return papa1, papa2

def mutacion(individuo, tasa_mut):
    if random.random() < tasa_mut:
        idx1, idx2 = random.sample(range(len(individuo)), 2)
        individuo[idx1], individuo[idx2] = individuo[idx2], individuo[idx1]
    return individuo

def cruzamiento(papa1, papa2, tasa_cruz, tasa_mut):
    if random.random() < tasa_cruz:
        punto_de_cruce = random.randint(1, len(papa1) - 1)

        hijo1 = papa1[:punto_de_cruce] + [x for x in papa2 if x not in papa1[:punto_de_cruce]]
        hijo2 = papa2[:punto_de_cruce] + [x for x in papa1 if x not in papa2[:punto_de_cruce]]
    else:
        hijo1 = papa1
        hijo2 = papa2

    return mutacion(hijo1, tasa_mut), mutacion(hijo2, tasa_mut)

def conversion(individuo, carga, tamaño_ventana, num_procesadores):
    asignacion_procesos = {i + 1: [] for i in range(num_procesadores)}
    indices = list(map(int, individuo))  # No se usa split() porque `individuo` es una lista
    x = 0
    while x < len(carga):
        for i, indice in enumerate(indices):
            clave = (i % num_procesadores) + 1
            asignacion_procesos[clave].append(carga[(indice - 1) + x])
        x = x + tamaño_ventana
    maxspan = max(sum(procesos) for procesos in asignacion_procesos.values())
    fitness = calcular_fitness(asignacion_procesos, maxspan)
    return asignacion_procesos, fitness
```

```python
def slide_window(procesos, size_w):
    ventanas = []
    for i in range(0, len(procesos), size_w):
        ventana = procesos[i:i + size_w]
        if len(ventana) < size_w:
            ventana += [0] * (size_w - len(ventana))
        ventanas.append(ventana)
    return ventanas

def calcular_estadisticas_procesadores(asignacion_procesos):
    colas_procesadores = [procesos for procesos in asignacion_procesos.values()]
    media_procesadores = sum(map(sum, colas_procesadores)) / len(colas_procesadores)
    max_cola = max(sum(procesos) for procesos in asignacion_procesos.values())
    return colas_procesadores, media_procesadores, max_cola

def imprimir_colas_procesadores(colas_procesadores):
    for i, cola in enumerate(colas_procesadores):
        print(f"Procesador {i} = {sum(cola)}")

def cargar_datos_csv(nombre_archivo):
    df = pd.read_csv(nombre_archivo)
    num_registros = len(df)
    return df, num_registros
```

Este bloque de código parece estar diseñado para distribuir y entrenar un modelo Word2Vec en paralelo utilizando un algoritmo genético para asignar tareas a diferentes procesadores de manera eficiente.

[34]:
```python
def distribuir_y_entrenar_word2vec(num_procesadores, df, tamaño_ventana, num_individuos, num_generaciones, tasa_mut, tasa_cruz):
    tiempos_entrenamiento = []

    for procesadores in range(2, num_procesadores + 1):
        mejor_asignacion_global = None
        mejor_carga_global = float('-inf')
        num_registros = len(df)

        carga = cargas(num_registros, tamaño_ventana)
        poblacion = generar_poblacion(num_individuos, tamaño_ventana)

        for generacion in range(num_generaciones):
            fitness_poblacion = []
            nueva_poblacion = []

            for individuo in poblacion:
                asignacion, _ = conversion(individuo, carga, tamaño_ventana, procesadores)
                carga_procesadores = [sum(procesos) for procesos in asignacion.values()]
                max_carga_procesador = max(carga_procesadores)

                if max_carga_procesador > mejor_carga_global:
                    mejor_carga_global = max_carga_procesador
                    mejor_asignacion_global = asignacion.copy()

                fitness = calcular_fitness(asignacion, max_carga_procesador)
                fitness_poblacion.append(fitness)
                nueva_poblacion.append(mutacion(individuo, tasa_mut))

            poblacion = nueva_poblacion

        tiempo_inicio = time.time()
        word2vec_model = Word2Vec(sentences=df['tokens'], vector_size=100, window=5, min_count=1, workers=procesadores)
        tiempo_fin = time.time()

        tiempo_transcurrido = tiempo_fin - tiempo_inicio
        print(f"Tiempo de entrenamiento de Word2Vec con {procesadores} procesadores: {tiempo_transcurrido} segundos")

        tiempos_entrenamiento.append((procesadores, tiempo_transcurrido))

    return tiempos_entrenamiento

# Obtener los tiempos de entrenamiento
tiempos_entrenamiento = distribuir_y_entrenar_word2vec(num_procesadores, df, tamaño_ventana, num_individuos, num_generaciones, ta

# Imprimir los resultados
for procesadores, tiempo in tiempos_entrenamiento:
    print(f"{procesadores} procesadores: {tiempo} s")
```

Evaluación

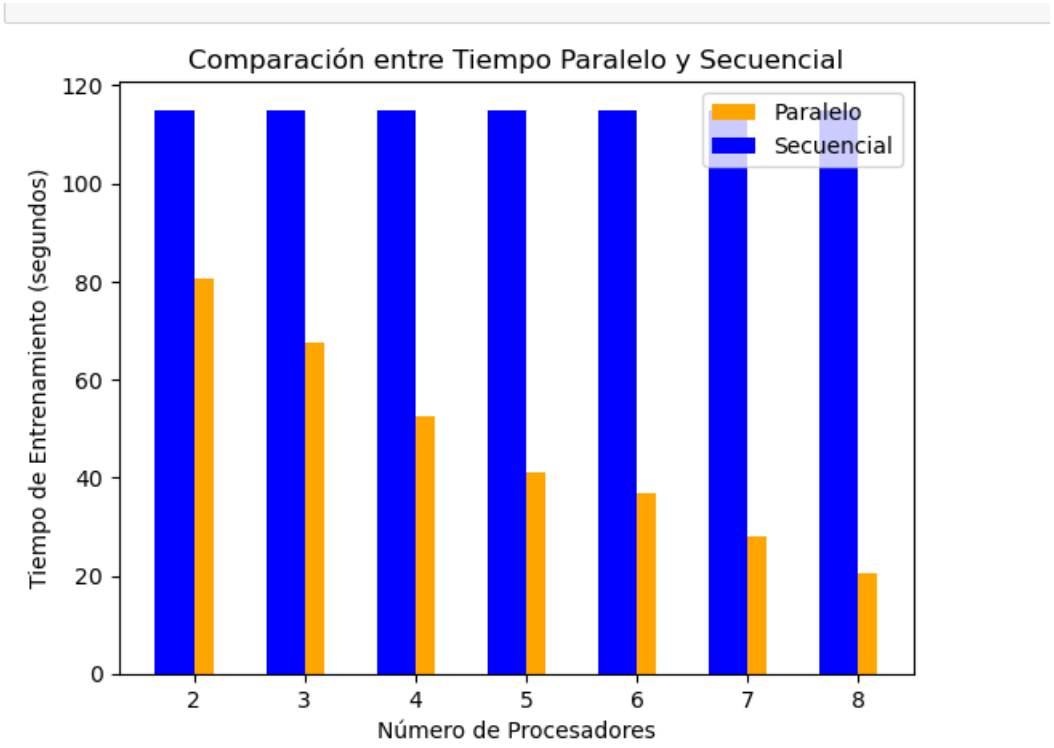
Word2vec sin balanceo de cargas y AG:

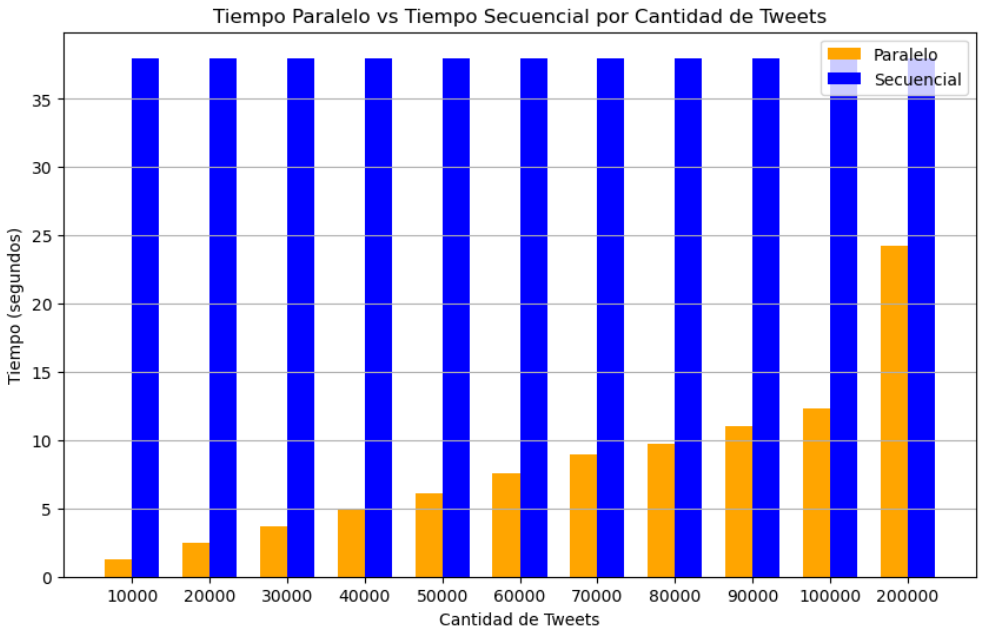
```
                                epochs=100, seed=42)
CPU times: total: 1h 8min 10s
Wall time: 37min 53s
```

Word2Vec con balanceo de cargas y AG:

```
Tiempo de entrenamiento de Word2Vec con 2 procesadores: 84.96683311462402 segundos
Tiempo de entrenamiento de Word2Vec con 3 procesadores: 68.050042390823364 segundos
Tiempo de entrenamiento de Word2Vec con 4 procesadores: 58.85554122924805 segundos
Tiempo de entrenamiento de Word2Vec con 5 procesadores: 43.41162467002869 segundos
Tiempo de entrenamiento de Word2Vec con 6 procesadores: 32.14200282096863 segundos
Tiempo de entrenamiento de Word2Vec con 7 procesadores: 24.61114430427551 segundos
Tiempo de entrenamiento de Word2Vec con 8 procesadores: 23.874298095703125 segundos
```

Graficas de comparación:





Nuestros modelos de machine learning con buenos resultados:

Tabla Comparativa de Modelos:

Modelo	Precisión	Sensibilidad	Especificidad	FPR
Random Forest	0.974551	0.912098	0.927427	0.072573
k-NN	0.982516	0.968653	0.803238	0.196762
AdaBoost	0.950516	0.914044	0.899987	0.100013

Conclusión

En este proyecto, se presenta una implementación de un algoritmo genético (AG) que aborda el desafío de distribuir una lista de procesos en varios procesadores de manera eficiente. A través del AG, el programa busca encontrar una solución óptima que cumpla con las restricciones de carga máxima y mínima en cada procesador. Durante el proceso de entrenamiento, el AG genera y evalúa poblaciones de cromosomas, realiza operaciones de cruce y mutación, y selecciona la élite basada en la aptitud. La solución final proporciona la distribución óptima de procesos en cada procesador.

Este código demuestra la aplicabilidad de los algoritmos genéticos en la resolución de problemas de optimización, como la distribución de cargas. A través de la definición de una estructura de clases y métodos, se implementa un enfoque sistemático y modular para abordar el problema. El código se beneficia de la aleatoriedad en la selección de cromosomas y operaciones de cruce, lo que contribuye a explorar diferentes soluciones potenciales.

La ejecución del AG proporciona información detallada sobre la distribución de cargas en cada paso del proceso de entrenamiento, lo que facilita la evaluación y comprensión de los resultados. Además, se mide el tiempo de ejecución para tener una idea de la eficiencia del algoritmo en función del tamaño del problema.

Referencias

Zomaya, A. (2001). Observations on Using Genetic Algorithms for Dynamic Load- Balancing. ResearchGate. https://www.researchgate.net/publication/3300581_Observations_on_Using_Genet ic_Algorithms_for_Dynamic_Load-Balancing QuestionPro. (2023).