Universidad Politécnica Salesiana
Rayner Steven Palta Tenecela

Sistemas Expertos, prueba 2.

Enunciado:

- Diseñe y desarrolle un algoritmo Knn en Neo4j para:
  - **Fila A - 0**: Usemos el ejemplo de conjunto de datos de Kaggle.com. Elegír el conjunto de datos del automóvil para este ejemplo y permite predecir el tipo de carro o automóvil, para ello el siguiente link de datos https://github.com/yfujieda/tech-cookbook/blob/master/python/knn-example2/data/car_dataset.csv [1]
  - **Fila B - 1:** Este es un conjunto de datos de empleados en una empresa y el resultado es estudiar sobre la deserción de los empleados, para ello se debe descargar los datos del siguiente link: http://smalldatabrains.com/wp-content/uploads/2018/03/data.csv [2].
  - **Fila C – 2:** Predicción de "género" mediante el valor de "Compra" y el tipo de "Ocupación", para descargar los datos del siguiente link: https://www.kaggle.com/alllexander/blackfriday [3]

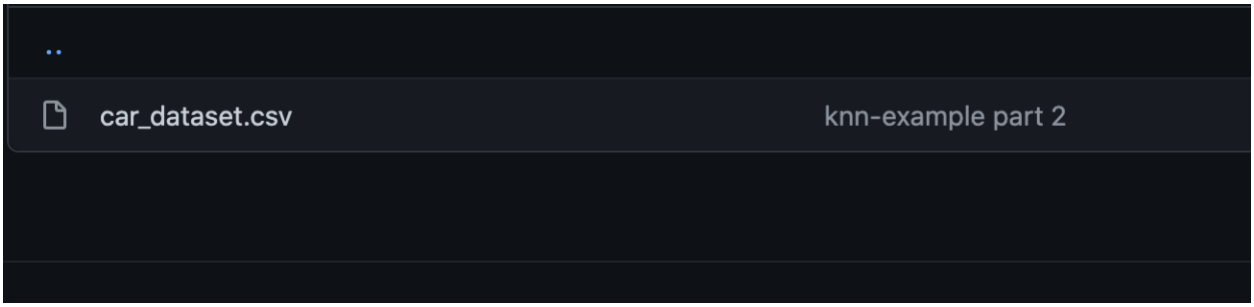
- Ingresar cada uno de los datos en un nodo y obtener el grado de similitud se recomienda utilizar la distancia Euclidiana o Person, una vez obtenido la similitud ingresar datos de prueba para validar (Máximo 3 datos).
- Generar otro entorno en donde solo ingrese el 70% de los datos y validar con el 30%.
- Agregar el grafico con los nodos conformados.

El proceso de programación desarrollado deberá considerar los siguientes aspectos:

- Se deberá tener un archivo que tenga todos los procesos o código de búsqueda y datos de Neo4j (https://neo4j.com/docs/labs/apoc/current/export/cypher/).

Proceso.

Descargar el csv a usarse desde la url dada.



Crear un archivo csv con todos los datos recopilados.

Una vez creado el archivo creamos nuestra base de datos en Neo4j. Es importante instalar los plugins de APOC y Science Library para poder desarrollar este problema.

Ya instalados los plugins debemos mover el archivo csv a la carpeta import de la base creada anteriormente.

Ahora se procede a cargar los datos para la creación de los nodos.

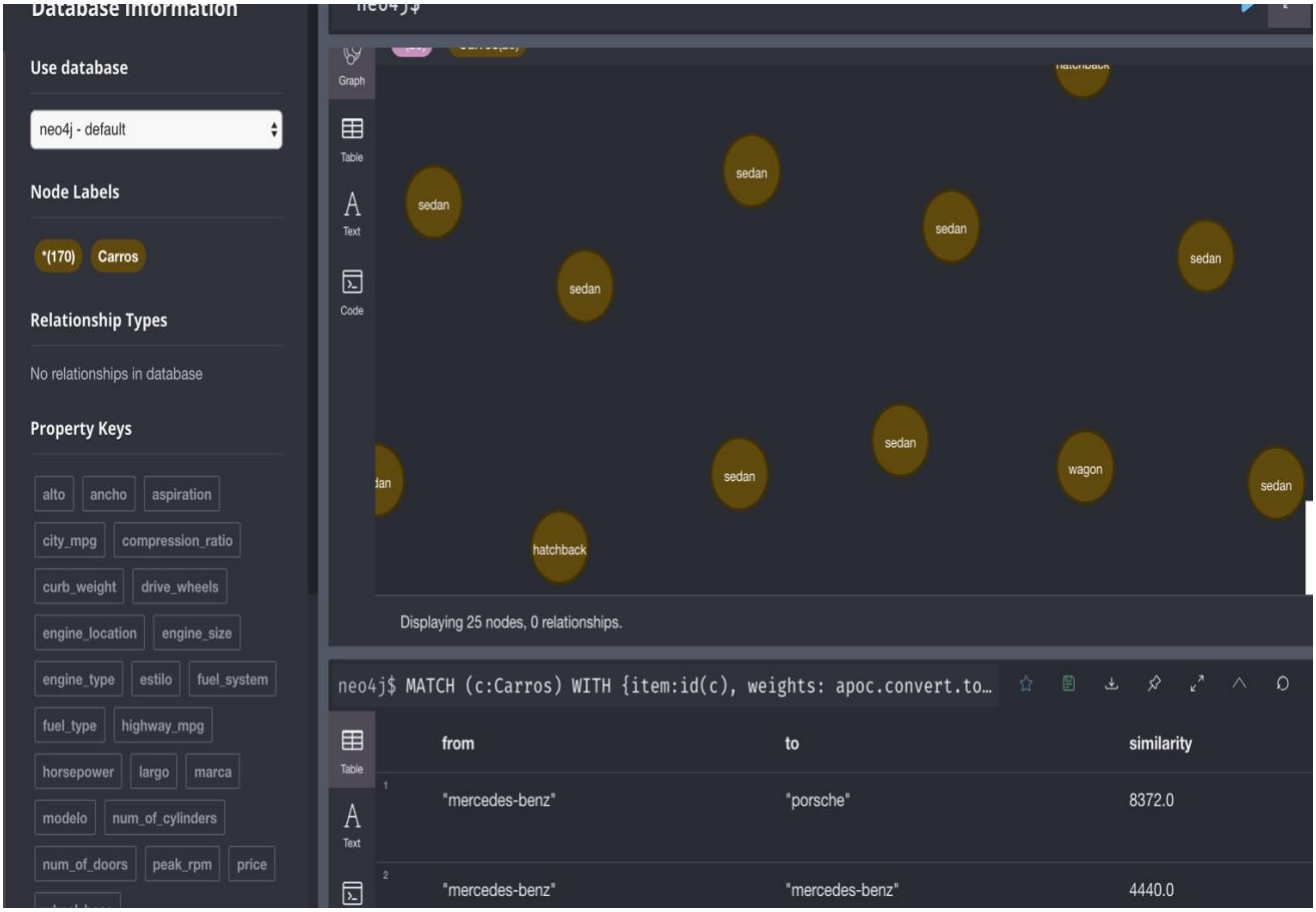
Una vez termine de correr el script los nodos se crean y se verán de la siguiente manera.

Ahora es necesario diseniar el algoritmo de similitud de distancia de Eucliades.

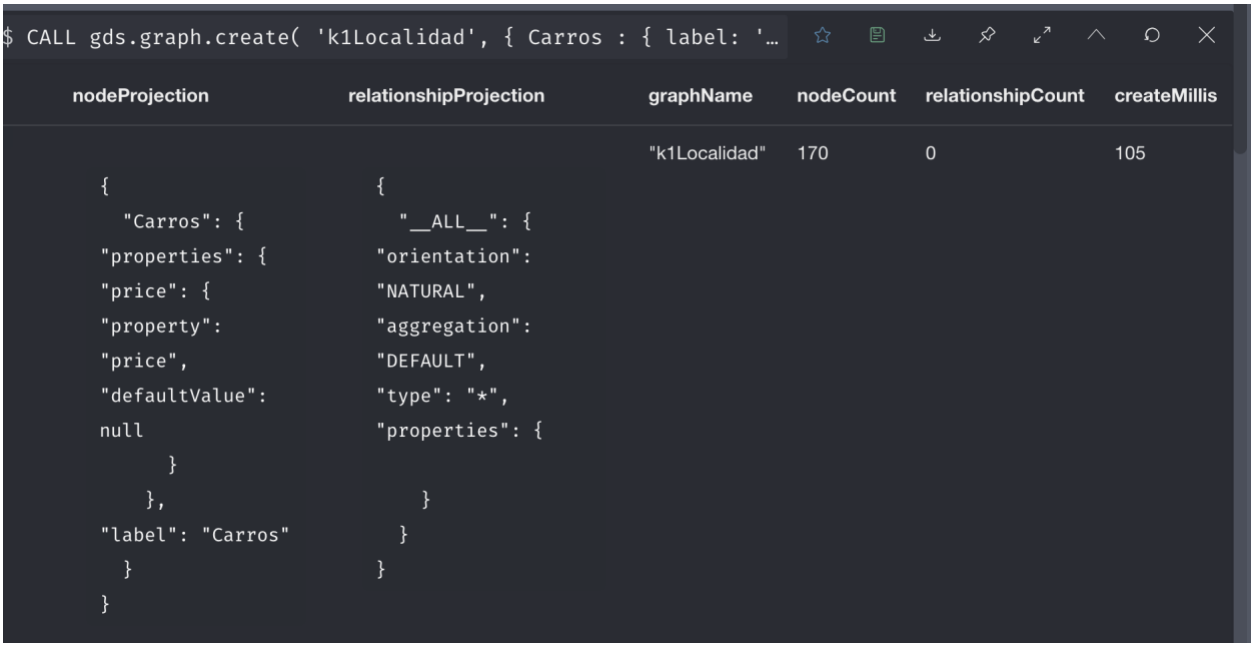
```
MATCH (c:Carros)
 WITH {item:id(c), weights: apoc.convert.toIntList(c.price)} AS userData
 WITH collect(userData) AS valorPrecio
 CALL gds.alpha.similarity.euclidean.stream({
  data: valorPrecio,
  skipValue: null
 })
 YIELD item1, item2,count1, count2, similarity
 RETURN gds.util.asNode(item1).marca AS from, gds.util.asNode(item2).marca AS
to, similarity
 ORDER BY similarity DESC
```

from	to	similarity
"mercedes-benz"	"porsche"	8372.0
"mercedes-benz"	"mercedes-benz"	4440.0
"bmw"	"porsche"	4287.0
"bmw"	"mercedes-benz"	4085.0
"mercedes-benz"	"bmw"	4085.0

Parte 2.
Creación de una nueva base con el 70 % de los datos.



Ahora se procede a crear el mapa de datos que será usado en el algoritmo con estos datos para comprobar el grado de similitud que existen en nuestro precios con relacion a la marca de los vehículos.



Una vez creado el mapa debemos comprobar la cantidad de memoria que tomará ejecutar este algoritmo con los datos especificados.

```
CALL gds.beta.knn.write.estimate('k1Localidad', {
  nodeWeightProperty: 'price',
  writeRelationshipType: 'Carros',
  writeProperty: 'score',
  topK: 1
})
YIELD nodeCount as nodos, bytesMin , bytesMax, requiredMemory as
memoriaRequerida
```

```
j$ CALL gds.beta.knn.write.estimate('k1Localidad', { nodeWeigh…
```

	nodos	bytesMin	bytesMax	memoriaRequerida
1	170	13744	46384	"[13744 Bytes ... 45 KiB]"

Con la cantidad de memoria conocida procedemos a crear nuestro algoritmo knn.

```
CALL gds.beta.knn.write.estimate('k1Localidad', {
  nodeWeightProperty: 'price',
  writeRelationshipType: 'Carros',
  writeProperty: 'score',
  topK: 1
})
YIELD nodeCount as nodos, bytesMin , bytesMax, requiredMemory as
memoriaRequerida
```

```
j$ CALL gds.beta.knn.write.estimate('k1Localidad', { nodeWeigh…
```

	nodos	bytesMin	bytesMax	memoriaRequerida
1	170	13744	46384	"[13744 Bytes ... 45 KiB]"

```
CALL gds.beta.knn.stream('k1Localidad', {topK: 2, randomSeed: 42,
nodeWeightProperty: 'horsepower'})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).marca AS Marca1, gds.util.asNode(node2).modelo
AS Modelo, similarity AS Similitud
ORDER BY Similitud ASCENDING, Marca1, Modelo
```

4j$ CALL gds.beta.knn.stream('k1Localidad', {topK: 2, randomSee…

	Marca1	Modelo	Similitud
5	"jaguar"	"porsche 129"	0.017857142857142856
6	"jaguar"	"porsche 130"	0.037037037037037035
7	"porsche"	"jaguar 50"	0.037037037037037035
8	"volkswagen"	"mazda 52"	0.058823529411764705
9	"volkswagen"	"toyota 156"	0.09090909090909091
10	"volvo"	"porsche 126"	0.1

```
CALL gds.beta.knn.stream('k1Localidad', {topK: 2, randomSeed: 42,
nodeWeightProperty: 'price'})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).marca AS Marca1, gds.util.asNode(node2).marca
AS Marca2, similarity AS Similitud
ORDER BY Similitud ASCENDING, Marca1, Marca2

```

neo4j$ CALL gds.beta.knn.stream('k1Localidad', {topK: 2, randomSee…

Table
Text
Code

	Marca1	Marca2	Similitud
1	"audi"	"toyota"	0.0
2	"isuzu"	"chevrolet"	0.0
3	"isuzu"	"subaru"	0.0
4	"make"	"mercury"	0.0
5	"porsche"	"mercedes-benz"	0.0
6	"mercedes-benz"	"mercedes-benz"	0.00022517451024544022

```
CALL gds.beta.knn.stream('k1Localidad', {topK: 5,sampleRate: 1 , randomSeed:
-1, nodeWeightProperty: 'horsepower'})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).marca AS Modelo, gds.util.asNode(node2).marca
AS Modelo2, similarity AS Similitud
ORDER BY Similitud ASCENDING, Modelo, Modelo2
```

j$ CALL gds.beta.knn.stream('k1Localidad', {topK: 5,sampleRate…

	Modelo	Modelo2	Similitud
	"mazda"	"volkswagen"	0.5
250	"mazda"	"volvo"	0.5
251	"mercedes-benz"	"alfa-romero"	0.5
252	"mercedes-benz"	"alfa-romero"	0.5
253	"mercedes-benz"	"toyota"	0.5
254	"mercedes-benz"	"toyota"	0.5
255	"mercedes-benz"	"toyota"	0.5