Centro de Investigación en Matemáticas (CIMAT). Unidad Monterrey

Temas Selectos de Cómputo
Tarea 1

Marcelo Alberto Sanchez Zaragoza

29 de septiembre de 2021

1. Problema

Sea T un documento de texto. Se desea encontrar la palabra más frecuente y la menos frecuente. Defina:

a) La función *Map*.

b) La función *Reduce*.

Solución

En el siguiente cuadro mostramos la implementación que se realizo para encontrar los valores que nos piden utilizando MRJob.

```
%%file word_count_example.py
from mrjob.job import MRJob, MRStep

#### Ejercicio 1 ####
class WordCountMR(MRJob):

def mapper(self, _, line):
punctuation_marks = '.,;'
for mark in punctuation_marks:
line = line.replace(mark, '')
for word in line.split():
yield word.lower(), 1

def reducer(self, word, value): # la clave es la palabra
yield None, (sum(value), word) # no regresamos una llave, solo la suma

def reducer_find_min_word(self, _, word_count_2):
p = list(word_count_2) # van a ser una lista con todas las tuplas
yield min(p), max(p)

def steps(self):
return [ MRStep(mapper=self.mapper,
reducer=self.reducer),
MRStep(reducer=self.reducer_find_min_word)]

if __name__ == '__main__':
WordCountMR.run()

```

Se incluye una sección de código donde lo que realizamos es quitar algún carácter que pueda ser tomado en cuenta como palabra y nos de resultados equivocados.

a) La función *mapper*: La función mapper recibe como entrada cada una de las lineas del texto limpio, en esta sección lo que realizamos fue contar cada una de las palabras que contiene la linea y regresar como llave la palabra en minusculas y como valor 1, ya que eso nos va a ayudara al final.

b) La función *reducer*: La función reducer recibe como llave la palabra y como valor el número 1, una vez que tiene estos valores lo que

realiza es la suma de todo los elementos con esa misma llave, al final no regresa una llave pero si un valor que contiene la suma total de cada palabra que se repitio en el texto ingresado.

c) La función $reducer\_find\_min\_max\_word$: La función recibe el total de cada una de las palabras pero como no recibe una una llave lo que se reliza es pasar a una lista todas las sumas, al final lo unico que regreamos es una llave con valor 1 y una lista con la palabra más frecuenta y la menos frecuente.

2. Problema

Sea X una matrix de mxn. Modele bajo el paradigma de MapReduce una función que calcule el valor máximo de cada columna en X.

Solución

En el siguiente cuadro mostramos la implementación que se realizo para encontrar los valores que nos piden utilizando Spark.

```
1  def kobe(matrix_rdd):
2    col1 = matrix_rdd.flatMap(lambda row:[ (col, row[col]) for col in range(len(row)) ])
3  return col1.reduceByKey(lambda x, y: x if x > y else y).sortByKey().collect()
4
```

a) La función Map: Nos ayuda a encontrar la matriz transpuesta, recibe una fila y nos va a regresar una llave que sea el indice de columna y valor que va a tiene ese elemento.

b) La función $Reduce$: La función reduce recibe la llave que corresponde a cada columna y el valor de cada elemento en la fila pero solo va tomando los que tienen la misma llave, teniendo los datos ya se seleccionan el mayor correspondiente a cada columna.

3. Problema

Sea X una matrix de mxn, donde m es el número de ejemplos en el conjunto de datos, cada uno descrito por n características, y y un vector binario con las etiquetas de clase de cada ejemplo. Se desea construir el clasificador de Naive Bayes bajo el paradigma MapReduce, defina:

a) La función *Map*.

b) La función *Reduce*.

Solución

Como primer paso contamos cuantos datos contiene nuestra matriz X.

1.- Con el primer mapper1 solicitamos que nos regrese la llave para la clase a la que pertenece y como valor asignamos 1.

```
1: Def mapper1(lista_1)
```

2.- Lo que sigue es contar cada uno de ellos con alguna función contabilizamos la ocurrencia por categoria pero lo realizamos utilizando la llave que tiene cada una de ellas.

3.- En el reducer1 realizamos una suma total de la ocurrencia por llave, la cual tiene la categoria.

```
1: Def reducer1(lista_1)
```

4.- En el siguiente mapper2 necesitamos encontrar el número de ocurrencia de las características por categoria. En este caso ingresan valores en forma de tabla y regresamos una tupla donde la llave es igual a la característica y clase a la que pertenece y al valor le asignamos 1.

1: Def **mapper2**(lista_1)

5.- En el reducer2 lo que entra son las tuplas que resultaron del mapper2(caracteristica, clase y valor = 1) y se desea que nos regrese la frecuencia por caterística por categoria. Finalmente encontramos la ocurrencia por carcterística y por categoria.

1: Def **reducer2**(lista_1)

6.- El siguiente paso es obtener una lista con las probabilidades por categoria ya que se esta dividiendo la ocurrencia entre el total por clase.

7.- Como ultimo paso lo que resta es multiplicar las probabilidades de las características por clase y dividirlas entre la probabilidad de la clase.

4. Problema

Sea X una matrix de mxn, donde m es el número de ejemplos en el conjunto de datos, cada uno descrito por n características, y y un vector binario con las etiquetas de clase de cada ejemplo. Se desea seleccionar el conjunto de ejemplos en X de menor tamaño posible que minimice el error en el vecino más cercano.
Formule el problema bajo el paradigma MapReduce, definiendo:

a) La función Map.

b) La función $Reduce$.

Solución

Se muestra a continuación el Pseudocódigo:

Los datos que se ingresan para la implementación deben tener la clase hasta la ultima posición.
Def **nearesNeighborSearch**(train_rdd, test_set, k = 3):

1: Def **mapper**(train_subset)

2: Convertimos nuestro conjunto de datos en una lista.

3: Iteramos sobre cada uno de los puntos de prueba.

4: Calculamos la distancia de ese punto de prueba a cada uno de los ejemplos que se encuentran en el subconjunto de entrenamiento.

5: Ordenamos por distancias y nos quedamos con los primeros k elementos de dicho orden.

6: Regresa como llave el indice de ese punto de prueba y como valor una tupla con las distancias de nuestro punto de prueba a cada unos de los elementos del conjunto de entrenamiento y el respectivo elemento de entrenamiento.

En el paso 1 del **mapper** recibimos el conjunto de datos que se encuentra en cada subconjunto.

1: Def **reducer**(X, Y)

2: Unimos las dos tuplas X + Y.

3: Ordenamos los valores que tenemos y definimos una lista vacia.

4: Dado el valor del número de vecinos más cercanos tomamos para cada punto de prueba los k-vecinos más cercanos

5: Regresamos la lista con el indice del punto de prueba, la distancia a cada vecino y cada uno de los vecinos.

En el paso 1 del **reducer** recibimos dos elementos con la misma llave, dos listas del vecino más cercano correspondiente al mismo ejemplo de prueba.

En el siguiente mapper la intención es obtener una lista donde los elementos que va a contener son los indices de cada ejemplo de prueba y un arreglo con las etiquetas de cada vecino, en este caso regresa la lista ordenada de mayor a menor.

1: Def **mapper2**(lista_1)

La intención de hacer esta lista es observar que puntos tienen más vecinos cercanos y así poder reducir la dimensión de los datos.

En el siguiente mapper dicha lista es nos va a ayudar a obtener el error como primer paso y posteriormente comenzar a con la reducción del conjunto de datos, esto con el fin de encontrar un punto donde dicho error aumente y este sea un punto de paro, en este caso lo que regresamos solo es un valor, este valor es la posición de la lista_2 hasta donde se observo que el error no cambio.

1: Def **mapper3**(lista_2)

Con dicho indice que ingresa al mapper lo que unico que se va a relizar es la disminución a nuestros datos, ya que ese indice nos va a determinar hasta donde podemos reducir los datos sin aumentar el error.

1: Def **reducer3**(indice_final))

La intención de realizar lo antes descrito es encontrar primero las distancias de todos los elementos de la matriz X con la misma matriz X, cabe recalcar que dichas distancias estan ordenadas de menor a mayor, donde el primer elemento es el mismo x_i y posteriormente comienzan sus vecinos más cercanos.
Como resutado de dicho algoritmo se espera que podamos tener acceso a las clases de cada de unos de los vecinos más cercanos para cada elemento de la matriz X, una vez que se puede acceder a dicho elementos lo que

nos resta es resumir en una lista o vector las clases de cada de uno de los vecinos más cercanos incluyendo la clase del elemento x_i.
Ya que tenemos esta lista con los vecinos más cercanos se realiza un conteo de cuantos vecinos coinciden con la clase del elemento x_i, es decir, contamos hasta que punto la clase del elemento x_i se mantiene sin cambio a lo largo de la lista o vector, dichos resultados se guardan en otra lista con el indice y el número de vecinos más cercanos.
Lo que sigue es encontrar el error que tenemos al tomar el primer vecino más cercano, ya que que este punto sera el comienzo para determinar que tanto reduce los elementos de la matriz X.
Ya que encontramos dicho error vamos a recorrer la ultima lista que obtuvimos y en cada paso vamos a ir eliminando cada uno de los elementos de la matriz X y obteniendo un error_2 y compararlo con el del principio. Este criterio nos ayudara a observar si nuestro error aumenta es señal que ese punto no debe ser eleminado pero si no cambia quiere decir que se puede trabajar sin ese elemento.