

Reporte tarea 8 - Programación y algoritmos

Erick Salvador Alvarez Valencia, Centro de Investigación en Matemáticas, A.C.

Index Terms—Clasificador bayesiano, Componentes conexos.

I. INTRODUCCIÓN

En el presente reporte se analizará la implementación de dos programas realizados para la octava tarea, el primer programa es un clasificador de textos, el cual en base a un pequeño entrenamiento decidirá si el texto de prueba es spam o no. Esto se puede lograr mediante un algoritmo llamado "Naive Bayes."^{el} cual usa el teorema de Bayes para obtener las probabilidades deseadas.

El segundo programa es un analizador de imágenes binarias, dada una imagen con varios elementos conexos, el programa determinará cuáles elementos conexos hay, y cuál es el de mayor tamaño. Para determinar lo anterior, se usa un algoritmo de búsqueda por anchura, el cual es un algoritmo de búsqueda sin información que se encarga de recorrer todos los nodos conexos de un grafo o un árbol asociados.

En el presente reporte se verá una pequeña descripción de los programas, así como los algoritmos usados para la resolución, posteriormente unas pruebas de las ejecuciones y para culminar algunas conclusiones.

mds

Octubre 23, 2017

II. PROGRAMA 1: CLASIFICADOR DE TEXTOS

II-A. Descripción

En el programa 1 se presentó el problema de los clasificadores de email, los servidores modernos de correo tienen la tarea de decidir si un correo es Spam o no, y de ser así usar el mismo de entrenamiento para posteriores correos.

Queremos en base a conjuntos de entrenamiento determinar si un texto es o no spam, y para ello usaremos el algoritmo de "Naive Bayes."^{el} cual se encarga de definir las probabilidades de lo anterior usando el teorema de Bayes, el cual establece: $P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}$ De lo anterior queremos saber la probabilidad de que un correo sea spam conociendo su contenido: $P(X|Y)$. Esto se logra conociendo la probabilidad de que un conjunto de palabras se encuentren en un correo spam $P(Y|X)$, la probabilidad de que un correo cualquiera sea spam: $P(X)$ y que el conjunto de palabras aparezcan en un correo cualquiera: $P(Y)$.

De lo anterior podemos obtener la siguiente expresión: **(spam|conjunto de palabras) = P(conjunto de palabras|spam) * P(spam) / P(conjunto de palabras).**

Para aplicar lo anterior necesitamos un conjunto de entrenamiento y uno de prueba, en el conjunto de entrenamiento existirán dos archivos los cuales serán el que contiene correos spam y el que contiene los que no son, partiendo de ahí debemos leer palabra por palabra y generar dos tablas, las

cuales contendrán la palabra y la frecuencia de aparición de las mismas, una vez que se tengan creadas ambas tablas podremos crear una que contiene la intersección de estas mismas, dicha tabla contendrá tres columnas: Una conteniendo el string, la otra conteniendo las frecuencias en los textos spam y la tercera conteniendo las frecuencias en los textos libres de spam. A continuación veremos la forma de construcción de estas tablas:

Algorithm 1 Creación de tabla en base a un texto.

```
1: procedure CREATETABLE(text)
2:   dict ← Vector que contendrá objetos de tipo WORD
3:   while Hayan palabras disponibles do
4:     str ← Siguiente palabra en archivo.
5:     str ← toLower(str).
6:     str ← Validate(str).
7:     ind ← inDict(dict, str).
8:     if ind != -1 then
9:       AddToDict(dict, str).
10:    else
11:      Dict[ind].freq += 1
12:  return dict
```

En el Algoritmo 1. podemos ver que creamos un vector que contiene objetos de tipo **WORD**, los cuales contienen internamente un string representando a la palabra y un entero que representa su frecuencia. También podemos ver que se usan funciones para el tratamiento de las palabras, las cuales aplican reglas sencillas reglas para filtrar los caracteres inválidos y un método para convertir las palabras a minúsculas, de esta forma será más sencillo clasificar las palabras. Ahora, con este algoritmo creamos las tablas de palabras spam y no spam, una vez hecho esto necesita generar una intersección de ambos conjuntos para obtener una tabla, la cual contenga palabras únicas con sus frecuencias en la parte de spam y no spam. La tabla anterior la podemos generar con el siguiente algoritmo:

Algorithm 2 Intersección de dos tablas.

```
1: procedure INTERSECTTABLE(table1, table2)
2:   interTable ← Vector que contendrá objetos de tipo REG
3:   for i ← 1 to table1.size() do
4:     aux ← table1[i].str
5:     ind ← inDict(table2, aux).
6:     if ind != -1 then
7:       AddToDict(interTable, str, table[1].freq, table[2].freq).
8:   return interTable
```

En el Algoritmo 2. definimos un vector que contiene elementos de tipo **REG**, los cuales a parte de contener el string que los asocia, ahora contienen dos enteros que representan las frecuencias de cada conjunto de datos.

De la misma forma podemos notar la existencia de la función **inDict** la cual busca en una tabla el registro que contiene el string que se pasó por argumento, si encuentra dicho registro, regresará su índice, de caso contrario regresará menos uno.

Una vez que se tiene la tabla intersectada podemos clasificar textos de prueba, para ello se hace un proceso muy parecido a los descritos anteriormente, generamos una tabla con las palabras únicas del texto de prueba y una vez teniendo esta tabla y dependiendo si se quiere saber si el texto es spam o no definimos una función de clasificación la cual recorrerá cada palabra única y verá si existe en la tabla intersectada, de ser así obtenemos la frecuencia relativa dividiendo la frecuencia de la palabra entre el total de palabras de esa columna, por cada palabra obtenemos su frecuencia relativa y ahí aplicamos el concepto de Naive Bayes: $P(spam|a_1, a_2, \dots, a_n) = P(spam|a_1) * P(spam|a_2) * \dots * P(spam|a_n)$ Podemos ver que la probabilidad condicional de que un texto sea spam dado un conjunto de palabras es el producto de cada las probabilidades marginales.

Para lo anterior iremos multiplicando las frecuencias relativas para obtener la probabilidad de que un texto sea o no sea spam. A continuación se brindará el algoritmo de la función clasificación:

Algorithm 3 Clasificación de texto de prueba.

```

1: procedure CLASIFY(text, table, isSpam)
2:   cntSpam  $\leftarrow$  0.
3:   cntNoSpam  $\leftarrow$  0.
4:   for i  $\leftarrow$  1 to table.size() do
5:     cntSpam  $\leftarrow$  cntSpam + table[i].freq_S
6:     cntNoSpam  $\leftarrow$  cntNoSpam + table[i].freq_NS
7:   perc  $\leftarrow$  1.
8:   for i  $\leftarrow$  1 to text.size() do
9:     word  $\leftarrow$  text[i].str
10:    if ind  $\neq$  -1 then
11:      if isSpam then
12:        perc = perc * table[i].freq_S / cntSpam.
13:      else
14:        perc = perc * table[i].freq_NS / cntNoSpam.
15:   return perc

```

En el Algoritmo 3. podemos notar que hay un parámetro llamado **isSpam** el cual decide si se calculará el porcentaje de spam o no spam, una vez que realizamos lo anterior ya podemos definir si el texto es spam o no.

II-B. Ejecución y pruebas realizadas

A continuación se mostrarán dos resultados de ejecuciones realizadas al programa, una con conjuntos de entrenamiento de pocas palabras y la otra con conjuntos de entrenamiento más grandes.

```

-082017-04090203704:~/Documents/Maestria/GIT/Semestre 1/Programacion y algoritmos/prog2017_08_Alvarez_ES/015 ./main spam2.txt nospam1.txt test1.txt
Tamaño del conjunto de entrenamiento: 3 palabras.
Tamaño del conjunto de prueba: 4 palabras.
Resultado: El texto ingresado es spam.
Porcentaje de spam: 1.125%
Porcentaje de no spam: 1.64089%
-082017-04090203704:~/Documents/Maestria/GIT/Semestre 1/Programacion y algoritmos/prog2017_08_Alvarez_ES/015

```

(a) Figura 1. Ejecución del programa con un conjunto de datos pequeño.

```

-082017-04090203704:~/Documents/Maestria/GIT/Semestre 1/Programacion y algoritmos/prog2017_08_Alvarez_ES/015 ./main spam.txt nospam.txt test.txt
Tamaño del conjunto de entrenamiento: 55 palabras.
Tamaño del conjunto de prueba: 105 palabras.
Resultado: El texto ingresado no es spam.
Porcentaje de spam: 5.88171e-31%
Porcentaje de no spam: 4.96438e-30%
-082017-04090203704:~/Documents/Maestria/GIT/Semestre 1/Programacion y algoritmos/prog2017_08_Alvarez_ES/015

```

(b) Figura 2. Ejecución del programa con un conjunto de datos grande.

En la Figura 1. podemos observar que hay tres palabras en el entrenamiento y la probabilidad de ser Spam es de 1.84 % pero en la Figura 2. donde hay más datos las probabilidades se vuelven muy pequeñas, esto puede presentar un problema.

II-C. Conclusiones

Para este problema se analizó un algoritmo de clasificación que nos ayudó a determinar si un texto es considerado como spam, y pese a que funcionó de manera deseada no se recomienda utilizar este método ya que con archivos lo suficientemente grandes se generarán tablas igualmente grandes, posteriormente al multiplicar las frecuencias relativas para obtener las probabilidades podremos generar errores numéricos por redondeo y provocar que la probabilidad se convierta a cero.

Una posible mejora que se podría aplicar al algoritmo es el usar una estructura tipo diccionario como un mapa, el cual permite la búsqueda de palabras bastante rápida.

III. ANALIZADOR DE COMPONENTES CONEXAS

III-A. Descripción

Para este problema se dió la tarea de analizar una imagen binaria (formato PGM) la cual contenía diferentes componentes conexas, posteriormente determinar cuantas eran, y encontrar la de mayor tamaño. A continuación se muestra una imagen de ejemplo:



(c) Figura 3. Imagen con componentes conexas.

En la Figura 3. podemos determinar que los elementos en color blanco (las que tienen forma de gusano) son considerados como componentes conexas, en total se tiene 16 de ellas. Para encontrarlas todas se usó un famoso algoritmo de búsqueda en grafos llamado BFS (Breadth First Search) el cual

recorre todo el grafo o árbol donde es ejecutada, permitiendo así determinar cuál es la componente conexa. Este algoritmo utiliza una cola para ir almacenando los nodos vecinos al actual y así irlos visitando y posteriormente marcarlos. Para este caso se usó una matriz auxiliar donde se iban marcando los nodos ya visitados y una vez terminada la búsqueda ir guardando la componente con mayor tamaño.

La idea principal fue hacer un barrido a la matriz y una vez encontrado un pixel diferente de cero, se aplicaba el algoritmo de BFS empezando por ese pixel, el algoritmo devolvía el tamaño de la componente y una matriz donde se encontraba dicha componente ya marcada y fácil de identificar, posteriormente se borraba de la matriz original la componente y se continuaba con el barrido, de esa manera ya no se volvía a encontrar la misma componente y se podía tener la certeza que si se encontraba otro pixel diferente de cero, este pertenecía a una componente distinta, y de esta manera se contaron todas las componentes.

A continuación se mostrará el algoritmo de BFS usado en el programa:

Algorithm 4 Búsqueda en anchura.

```

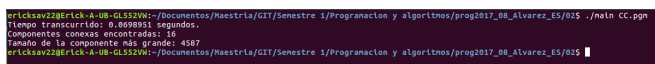
1: procedure BFS(img, width, height, ii, jj, visited)
2:   cnt  $\leftarrow$  1.
3:   q  $\leftarrow$  Nueva cola.
4:   visited[ii][jj] = True.
5:   q.push(ii, jj).
6:   while Cola no esté vacía do
7:     i  $\leftarrow$  q.i
8:     j  $\leftarrow$  q.j
9:     q.pop().
10:    neigh gets GetNeighbors(i, j).
11:    for i  $\leftarrow$  1 to neigh.size() do
12:      if not visited[neigh.i][neigh.j] then
13:        cnt  $\leftarrow$  cnt + 1.
14:        visited[neigh.i][neigh.j] = True
15:        q.push(neigh.i, neigh.j).
16:  return cnt

```

En el Algoritmo 4. podemos ver que para cada nodo se verifican sus vecinos y, en caso de no estar marcados, se visitan, y así hasta que la cola está vacía, lo que indica que toda la componente fue visitada.

III-B. Ejecución y pruebas realizadas

A continuación se mostrará el ejemplo de ejecución del algoritmo con la imagen mostrada en la Figura 1.



```

ericksav22@rick-A-U8-G152706: /documentos/maestría/GIT/Semestre 1/Programacion y algoritmos/prog2017_08_Alvarez_E5/015 $ ./main.cc.png
Tiempo transcurrido: 0.0698951 segundos.
Componentes conexas encontradas: 16
Tamaño de la componente más grande: 4587
ericksav22@rick-A-U8-G152706: /documentos/maestría/GIT/Semestre 1/Programacion y algoritmos/prog2017_08_Alvarez_E5/015 $

```

(d) Figura 4. Resultados mostrados por el analizador.



(e) Figura 5. Componente conexa de mayor tamaño.

En la Figura 4. podemos ver que el analizador nos muestra el tiempo tardado en ejecutarse, el cual es menor a 1 segundo, posteriormente nos dice que hay 16 componentes conexas en la imagen y que la de mayor tamaño abarca 4587 pixeles. En la Figura 5. se nos muestra la componente de mayor tamaño.

III-C. Conclusiones

Para este programa se pudo ver que el algoritmo propuesto trabaja apropiadamente, con él se pudo encontrar toda la información necesaria en la imagen en un tiempo pequeño. Una posible mejora podría ser el uso de la matriz original para marcar a los vecinos de cada nodo de la componente, y a la vez que se va realizando la búsqueda podríamos ir guardando cuáles son los nodos frontera para que una vez se termine el algoritmo y en caso de terminarse que la componente actual es la mayor hasta el momento, solo se recorra una parte de la matriz para copiar la componente.