Aprendizaje Automático

Proyecto Final Grupo 11, Curso 2018-2019

CLASIFICACIÓN DE COMENTARIOS SPAM EN YOUTUBE

Juan Gómez-Martinho González

Pedro Morell Prats

Tabla de contenido

Propósito del proyecto

El mundo digital está actualmente lleno de clasificadores de spam. Pretendemos tener una experiencia libre de contenido no deseado en Internet, y eso implica la existencia de multitud de herramientas que filtran mareas de contenido bruto; intentando que a nuestras pantallas llegue exclusivamente el contenido que demandamos.

Dado el volumen de datos actual, es inviable tener operadores humanos llevando a cabo dicha filtración, por lo que es necesario técnicas de clasificación automática para separar mensajes no deseados (spam) de mensajes "benignos" (*'ham'*).

Correos, tweets, comentarios, posts… hay multitud de tipos de mensajes que son susceptibles de filtrado.

Como se indica en el enunciado de este, "el objetivo del proyecto es seleccionar un conjunto de datos y utilizar el código de las prácticas para desarrollar un sistema de aprendizaje automático sobre el conjunto de datos elegido"; y en nuestro caso, el conjunto de datos escogido se corresponde con una selección de comentarios escritos por usuarios en 5 vídeos musicales diferentes de la plataforma *'YouTube'*.

Dicho conjunto de datos (o *'dataset'*, como se nombrará en adelante) está disponible de forma abierta a través de internet en el repositorio de Machine Learning de la UC Irvine.

Así pues, pretendemos aplicar las técnicas de Aprendizaje Automático aprendidas a lo largo del curso (regresión, redes neuronales, SVM y *'clustering'*) para probar la efectividad de diferentes clasificadores de spam sobre casos reales, analizando las mejores técnicas y descartando aquellas que no se adecúan a nuestro problema.

Descripción del *dataset*

Como hemos indicado en el Propósito del Proyecto, nuestro conjunto de datos corresponde a una selección de comentarios de vídeos de *YouTube*. Dichos datos están contenidos en formato *'.csv'* de la siguiente forma:

```
Id_comentario,autor,fecha,contenido,spam
```

Con esa forma, el repositorio ofrece cinco ficheros diferentes:

Dataset	**YouTube ID**	**#Spam**	**#Ham**	**Total**
Psy	9bZkp7q19f0	175	175	350
KatyPerry	CevxZvSJLk8	175	175	350
LMFAO	KQ6zr6kCPj8	236	202	438
Eminem	uelHwf8o7_U	245	203	448
Shakira	pRpeEdMmmQ0	174	196	370

Aunque intentamos importar los ficheros de datos directamente al '*workspace*', nos encontramos diversos problemas:

- La **codificación** de los caracteres no es compatible con '*Octave*', lo que genera caracteres extraños que alteran el mensaje original. Además, la presencia de caracteres no alfanuméricos en los ficheros producía problemas a la hora de generar el diccionario de nuestro entrenamiento.
- La **separación por comas** produce errores al haber comas dentro de los mensajes
  `(…2018,"Claro, es cierto",0)`
- La clasificación por mensaje citado entre comillas no funcionaba correctamente en nuestra versión del programa.

Para solucionar los problemas de importación, vimos necesario desarrollar un programa que pre-procesara los datos y generara los ficheros listos para usar por '*Octave*'.

Dicho programa ('csv-*unifier.jar*') permite seleccionar multitud de archivos '*.csv*' que cumplan la estructura de campos. Al ejecutarlo, lee el contenido de todos los archivos, selecciona las columnas relevantes para nuestro problema (`autor,contenido,spam`) y genera dos archivos:

- **plain.txt:** archivo con los datos de entrenamiento ya seleccionados, unificados y estandarizados.
- **vocab.txt:** archivo con todas las palabras que aparecen en los mensajes de los casos de entrenamiento, cada una en una línea y almacenada una única vez, sin caracteres no alfanuméricos.

El código del programa está disponible junto con la entrega en formato '*.zip*' ('*AA_CSV_Processor.zip*') o en un repositorio público de GitHub bajo licencia MIT.

Así pues, tras la ejecución del programa obtenemos los datos listos para importarlos en '*Octave*' y pasar a construir nuestros programas de aprendizaje automático.

Para la ejecución de dichos programas, nos encontrábamos ante el dilema de qué variables emplear de forma significativa para nuestro problema. Si bien es cierto que en la descripción del repositorio figuraban 5 variables, no son empleables de forma directa:

- **`Id_comentario`**: se trata de un identificador único para cada mensaje, por lo que no es relevante para ningún tipo de algoritmo.
- **`Autor`**: En efecto, emplear el id del autor podría ser interesante para detectar cuentas dedicadas a generar mensajes spam. Decidimos emplearlo más adelante para versiones refinadas del programa.
- **`Fecha`**: Lo consideramos irrelevante para la clasificación de los mensajes.

- **`Contenido`**: A través del diccionario de palabras que aparecen en mensajes spam, la aparición de dichas palabras en cada mensaje proporciona un array de valores lógicos (0/1) empleable para clasificar en función de la aparición de diversos términos.
- **`Spam`**: se trata de la etiqueta del caso de entrenamiento. Indica si el mensaje es spam (1) o no (0).

Siguiendo este razonamiento, el conjunto de datos que obtenemos para cada caso de entrenamiento es (`autor`,`contenido`,`spam`). Consideramos ampliarlo para versiones posteriores del programa tras implementar los algoritmos básicos (empleando el conjunto apariciones-spam), aunque por falta de tiempo no ha sido posible.

Representación de los datos en *Octave*

Para la generación de los datos en el formato propio de octave, guardados en un fichero reutilizable de extensión *'.mat'* con las variables necesarias, se ha empleado el siguiente script de Octave:

```
loader;
vocabList = getVocabList('vocab.txt');

for i = 1:length(vocabList)
  x = vocabList{i};
  struct.(x) = i;
endfor

[X_c, y_c] = loadComment(names, messages, spam, vocabList, struct);
[X_s, y_s] = loadSpam(names, messages, spam, vocabList, struct);
lim_c = floor(rows(X_c) * 0.8);
lim_s = floor(rows(X_s) * 0.8);

X = [ X_c(1:lim_c,:) ; X_s(1:lim_s,:) ];
y = [ y_c(1:lim_c,:) ; y_s(1:lim_s,:) ];

Xval = [ X_c(lim_c+1:end,:) ; X_s(lim_s+1:end,:) ];
yval = [ y_c(lim_c+1:end,:) ; y_s(lim_s+1:end,:) ];

save data.mat X Xval y yval vocabList;
```

`processData.m`

Este código llama a la función *`Loader`* para obtener las variables con los datos anteriormente: *`names`*, *`messages`* y *`spam`*. Con dichas variables, se genera una estructura con el diccionario de palabras y se llaman a las dos funciones que más carga computacional tienen: *`LoadComment`* y *`LoadSpam`* (análogas a las funciones de la práctica 6)

Son las funciones encargadas de generar las matrices lógicas (contenido de 1/0) que indican la aparición en cada caso de entrenamiento de las palabras de *`vocabList`* para los mensajes etiquetados como spam (X_s, y_s) y como no spam (X_c, y_c).

```
function [X, Y] = loadSpam(names, messages, spam, vocabList, vocabulario)
    X = [];
    messagesIND = find(spam == 1);
    spamMsg = cell(rows(messagesIND),1);
    counter = 1;
    for i = 1:rows(messages)
      if (any(messagesIND == i))
        spamMsg{counter} = messages{i};
        counter++;
      endif
    endfor

    for j = 1:rows(spamMsg)
      wor_s = processEmail(spamMsg{j});
      Xi = zeros(1, length(vocabList));
      while ~isempty(wor_s)

         [str,wor_s] = strtok(wor_s,[' ']);
         if isfield(vocabulario,str)
          Xi(vocabulario.(str))=1;
         endif
      endwhile
      X = [X;Xi];
    endfor
    Y = ones(rows(X), 1);
endfunction
```

loadSpam.m

Una vez se han obtenido dichas matrices diferenciadas, se separan en **casos de entrenamiento** (80% del total, unidas en matrices únicas con datos y etiquetas *X, y*) y **casos de validación** (*Xval*, *yval*).

Por último, las variables relevantes son guardadas en el fichero *'data.mat'* para evitar la repetición de un procesamiento pesado, sustituible por la carga de los datos ya procesados una única vez.

Así pues, ya tenemos los datos listos para ser usados por las diversas técnicas de Aprendizaje Automático, aunque por motivos que desconocemos; durante la importación de *octave* en el método *Loader* se corrompen bastantes filas, quedando en total alrededor de 700 casos de prueba.

Métodos no aplicables

Regresión Lineal

Teniendo en cuenta la naturaleza de nuestro problema, el método de Regresión Lineal no es aplicable sobre nuestro '*dataset*'.

El resultado de aplicar la regresión (hipótesis) debe ser un valor 1 o 0 (spam o no spam), y sólo uno de esos valores. Sin embargo, la Regresión Lineal permite que dicha hipótesis sea cualquier número real; no sólo los dos valores deseados.

Así, aunque deseamos que $h_\theta(x) \in \{0,1\}$ con la regresión lineal se obtiene $h_\theta(x) \in \mathbb{R}$, por lo que este método; al no ser válido para problemas de clasificación; ha sido descartado en nuestro proyecto.

Regresión Logística Multi-clase

Si bien es cierto que la Regresión Logística sí que es interesante para nuestro problema, la Regresión Logística Multi-clase no lo es. Ésta técnica es una adaptación de clasificación logística para problemas de más de dos etiquetas, y en nuestro caso se trata de una clasificación binaria (spam / no spam).

Por ello, no aplicaremos la Regresión Logística Multi-clase en nuestro problema.

Clustering

Hay dos motivos por los cuales no vamos a aplicar algoritmos de '*clustering*' sobre nuestro problema:

1. Nuestro *'dataset'* está orientado a técnicas de aprendizaje supervisado, y las técnicas de agrupamiento son no supervisadas.
2. El algoritmo '*k-means*' (que es el usado durante el curso) no es especialmente preciso con nuestra representación de los datos, y menos con una clasificación binaria.

Así pues, tampoco aplicaremos las técnicas de *clustering* con nuestro conjunto de datos.

Regresión Logística

Al tratarse de un problema de clasificación es posible aplicar el algoritmo de regresión logística. Para ello se ha utilizado el siguiente script, que utiliza el fichero *data.mat*, generado por la función *processData.m*.

```
clear;
warning('off', 'all');

load('data.mat');

x = [ones(rows(y),1), X];
xval = [ones(rows(yval),1), Xval];

theta = [zeros(columns(x),1)];

[J, grad] = coste(theta, X, y);
opciones = optimset('GradObj', 'on', 'MaxIter', 400);
[theta, cost] = fminunc(@(t)(coste(t, X, y)), theta, opciones);
accuracy = acc(theta, xval, yval);
```

logisticReg.m

Tras cargar *data.mat* es necesario adaptar las matrices *X* y *Xval* al problema. Para ello se añade una fila de unos a ambas matrices. A continuación, se crea una matriz de ceros, con una fila y tantas columnas como la matriz *x*. Esta se utiliza como argumento, junto con *X* e *y* en la función coste, mostrada a continuación.

```
function [J, grad] = coste(theta, x, y)
  m = rows(y);
  X = [ones(m,1), x];
  J = (1/m) * ((-y)'*log(funcionSigmoide(X*theta)) –
      (1-y)'*log(1-funcionSigmoide(X*theta)));
  grad = (1/m) * ((funcionSigmoide(X*theta) - y)' * X)';
endfunction
```

coste.m

Esta función recibe como parámetros *theta*, la matriz *X* con el conjunto de ejemplos, y el vector *y*, y devuelve el valor de la función de coste junto con un vector con los valores del gradiente de dicha función. Para ello hace uso de la función sigmoide, tanto para la función de coste como para el gradiente.

Una vez calculado el coste, para calcular el valor óptimo de los parámetros se hace uso de la función *fminunc* de Octave. Finalmente se utiliza la función *acc*, mostrada a continuación para calcular el nivel de acierto del algoritmo, que se guarda en la variable *accuracy*.

```
function a = acc(theta, x, y)
  reg = funcionSigmoide(theta'*x')';
  predicciones = gt(reg,0.5);
  aciertos = sum(predicciones == y);
  a = aciertos / rows(y) * 100;
endfunction
```

acc.m

La función anterior calcula el porcentaje de ejemplos de entrenamiento que se clasifican correctamente, comparando los resultados de aplicar la regresión con el valor suministrado.

En nuestro caso, aplicando el modelo obtenido sobre los casos de validación se obtiene un acierto del 71.8%.

Redes Neuronales

La implementación de la red neuronal sobre el código que poseíamos de las prácticas ha sido relativamente sencilla. La ejecución del entrenamiento y de la predicción de la red se realiza mediante el siguiente script de Octave:

```
clear;
warning('off', 'Octave:possible-matlab-short-circuit-operator');

load('data.mat');

lambda = 0.5;
num_entradas = rows(vocabList);
num_ocultas = 25;
num_etiquetas = 2;
options = optimset('MaxIter', 50);

r_theta1 = pesosAleatorios(num_entradas, num_ocultas);
r_theta2 = pesosAleatorios(num_ocultas, num_etiquetas);
params_rn = [r_theta1(:) ; r_theta2(:)];

theta = fmincg(@(t)(costeRN(params_rn, num_entradas, num_ocultas, num_etiquetas,
X, y, lambda)), params_rn, options);

t1 = reshape(theta(1:num_ocultas * (num_entradas + 1)),
                num_ocultas, (num_entradas + 1));
t2 = reshape(theta((1 + (num_ocultas * (num_entradas + 1))):end),
                num_etiquetas, (num_ocultas + 1));

acc = acierto(t1, t2, Xval, yval);

fprintf('\nTraining Set Accuracy: %f%%\n', acc * 100);
```

NNScript.m

Tras cargar los datos de entrenamiento y de validación, se generan las variables propias de una red neuronal:

- **`lambda`**: tras probar distintos valores (`0.01, 0.1, 0.5, 1, 2`), hemos comprobado que no hay apenas diferencia a la hora de ejecutarse, así que nos hemos quedado con `0.5`, que parecía dar el resultado ligeramente más alto.
- **`num_entradas`**: como es dependiente del número de variables, se obtiene comprobando el número de palabras del diccionario (`rows(vocablist)`).
- **`num_ocultas`**: Es el número de unidades en la capa oculta. Al igual que lambda, no se ha percibido un cambio demasiado alto al variar el número de unidades, por lo que se ha mantenido en 25.
- **`num_etiquetas`**: número de posibles resultados. En nuestro caso sólo hay 2 (spam / no spam).
- **`options`**: argumentos necesarios para emplear la función `fmincg`. Establecemos `50` iteraciones como número necesario para obtener valores fiables de theta, ya que el cambio de órdenes de magnitud (`5, 500, 5000`) producen una variación casi nula.
- **`params_rn`**: contiene los pesos de la red neuronal iniciales. Como no poseemos ninguna aproximación inicial, se obtienen aleatoriamente mediante la función `pesosAleatorios`.

Una vez inicializados los parámetros, se obtienen los valores de theta llamando a la función `fmincg` para reducir el coste del cómputo de la red neuronal. La función que calcula dicho coste es `costeRN`, que realiza la computación de la red neuronal.

Una vez obtenidos los thetas óptimos, se separan en dos variables (theta 1 y theta 2) y se computa con la función de predicción para obtener la precisión con respecto a los casos de entrenamiento.

```
function [J, grad] = costeRN(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, lambda)

  Theta1 = reshape(params_rn(1:num_ocultas * (num_entradas + 1)),num_ocultas, (num_entradas + 1));
  Theta2 = reshape(params_rn((1 + (num_ocultas * (num_entradas + 1))):end), num_etiquetas, (num_ocultas + 1));

  m = size(X, 1); %tamaño de casos de entrenamiento
  I = eye(num_etiquetas); % Matriz cuadrada con el número de etiquetas
  Y = zeros(m, num_etiquetas); %tamaño de casos de entrenamiento
  for i=1:m
    Y(i, :)= I(y(i), :);
  end

  % Computación de la red neuronal
  a1 = [ones(m, 1) X];
  z2 = a1 * Theta1';
  a2 = [ones(size(z2, 1), 1) sigmoide(z2)];
  z3 = a2*Theta2';
  a3 = sigmoide(z3);
  h_theta = a3;

  prim = (1/m)*sum(sum((-Y).*log(h_theta) - (1-Y).*log(1 - h_theta), 2));
  termino_reg = (lambda/(2*m))*(sum(sum(Theta1(:, 2:end).^2, 2)) + sum(sum(Theta2(:,2:end).^2, 2)));

  J = prim + termino_reg;

  Sigma3 = a3 - Y;
  Sigma2 = (Sigma3*Theta2 .* sigmoideDER([ones(size(z2, 1), 1) z2]))(:, 2:end);

  Delta_1 = Sigma2'*a1;
  Delta_2 = Sigma3'*a2;

  Theta1_grad = Delta_1./m + (lambda/m)*[zeros(size(Theta1,1), 1) Theta1(:, 2:end)];
  Theta2_grad = Delta_2./m + (lambda/m)*[zeros(size(Theta2,1), 1) Theta2(:, 2:end)];

  grad = [Theta1_grad(:) ; Theta2_grad(:)];

endfunction
```

costeRN.m

Esta función realiza la computación de la red neuronal de forma sencilla, vectorial (no iterativa) y regularizada.

Con esta implementación se obtiene un resultado del Training Set Accuracy del 52.348993%.

Support Vector Machines

Para aplicar el algoritmo de Support Vector Machines se utiliza el siguiente script.

```
clear;
warning('off', 'all');

load('data.mat');

[C, sigma] = generador(X,y,Xval,yval);
model = svmTrain(X, y, C, @(x1, x2) kernelGaus(x1,x2,sigma));
```

mainSvm.m

La función hace uso del fichero *data.mat*, obtenido a partir de *processData*. Una vez cargadas las matrices lógicas se llama a la función *generador* para calcular los valores óptimos de *C* y *sigma*. Dicha función se muestra a continuación.

```
function [C, sigma] = generador(X, y, Xval, yval)
  conjunto = [0.01 0.03 0.1 0.3 1 3 10 30];
  error_min = inf;
  for C = conjunto
    for sigma = conjunto
      model = svmTrain(X, y, C, @(x1, x2) kernelGaus(x1,x2,sigma));
      err   = mean(double(svmPredict(model, Xval) ~= yval));
      if( err <= error_min )
        C_final = C;
        sigma_final = sigma;
        error_min = err;
      end
    endfor
  end

C = C_final;
sigma = sigma_final;

endfunction
```

generador.m

La función '*generador*' prueba diversos pares de *c* y *sigma*, los entrena y prueba si el error obtenido es el mejor hasta el momento. Si es así, lo almacena y sigue probando; hasta que finalmente se obtiene el mejor par C-Sigma, que pasa a entrenarse.

Con el modelo obtenido a partir del par óptimo de *C* y *sigma* se puede aplicar la función mean(double(svmPredict(model, Xval) ~= yval)), que devuelve un error de 0.235, por lo que la precisión media del algoritmo es de un 76,4%.

Conclusión

Tras ejecutar nuestras implementaciones, se obtienen los siguientes valores:

	Regresión Logística		**Red Neuronal**		**SVM**	
	Precisión	T de ejecución	Precisión	T de ejecución	Precisión	T de ejecución
Caso 1	71,812%	25s	52,35%	10s	76,5%	41m
Caso 2	71,812%	27,4s	52,34%	10,5s	77,3%	40m
Caso 3	71,812%	30s	53%	10,28s	77,1%	40m
Media:	**71,812**	**27,46s**	**52.56**	**10,26s**	**76.9**	**40m**

Observando dichos valores, podemos asegurar que sin duda el método óptimo para clasificar los mensajes como spam o no spam es la **Regresión Logística**. Aunque la red neuronal sea más rápida, acierta poco más del 50%, y aunque el SVM sea un 5% más preciso (76.9% de media), el coste temporal es completamente insostenible.

Así pues, observamos que es sencillo programar un filtro de spam que atendiendo únicamente a las palabras contenidas en el cuerpo del mensaje sea bastante preciso. Empleando 591 casos de entrenamiento y 149 de validación (proporción 80-20) obtenemos un **acierto del 71%**.