Instituto de Ingeniería Eléctrica

Facultad de Ingeniería

Universidad de la República

Reconocimiento de Patrones

Gimena Urcelay & Milton Bentos Trabajo Final

9 de diciembre de 2014

Índice

1. Generalidades

1.1. Descripción del problema estándar: Reconocimiento de LETRAS

El problema consistirá en identificar letras a partir de un gran número de datos basados en imágenes en blanco y negro de 26 letras mayúsculas del alfabeto. Estas imágenes se generaron para 20 tipos de letra diferente y fueron aleatoriamente distorsionadas para generar 20000 imágenes. A los efectos del proyecto, no se trabajará con las imágenes sino con un conjunto de 16 características extraídas a partir de esas imágenes.

Datos

Se dispone de un conjunto de muestras de entrenamiento previamente etiquetadas en la forma (vector de características, clase). También se dispondrá de un conjunto de muestras de prueba para la evaluación final del algoritmo, el cual será entregado poco antes de la evaluación final. La validación del desempeño de los modelos corre por cuenta de los estudiantes, y debe ser realizada en base al particionado de los datos de entrenamiento. El conjunto de datos de entrenamiento consistirá en 16000 elementos. El conjunto de test consistirá en 4000 elementos extraídos de un texto en español (tomando la $\tilde{\text{N}}$ como una N). Las características son normalizadas en un rango que varía entre 0 y 15, siendo representadas con números enteros.

A efectos del entendimiento general del problema, se describe brevemente en qué consiste cada una de las características calculadas sobre las imágenes de las letras. Por más detalles se puede consultar Frey y Slate 1991.

1. x-box posición horizontal del recuadro (int)
2. y-box posición vertical del recuadro (int)
3. width ancho del recuadro (int)
4. high altura del recuadro (int)
5. onpix # total de pixels encendidos (int)
6. x-bar media de x de pixels encendidos dentro del recuadro (int)
7. y-bar media de y de pixels encendidos dentro del recuadro (int)
8. x2bar varianza de x (int)
9. y2bar varianza de y (int)
10. xybar correlación entre x e y (int)
11. x2ybr valor medio de $x * x * y$ (int)
12. xy2br valor medio $x * y * y$ (int)
13. xegvy correlación de bordes horizontales con y (int)
14. y-ege conteo medio de bordes verticales abajo a arriba (int)
15. yegvx correlación de bordes verticales con x (int)
16. clase letras mayúsculas (26 de la A a la Z)

Objetivo

Dado un conjunto de datos de testing donde N_j es la cantidad de muestras de la clase j y E_j la cantidad de elementos de esa clase mal clasificados por un algoritmo dado, la tasa de acierto (también conocido como accuracy) para la clase j, A_j , viene dada por:

$$A_j = (1 - E_j)/N_j \tag{1}$$

Siendo C el número de clases, se tendrá que la media aritmética de las tasas de acierto por clase se define como :

$$A_{mean} = 1/C \sum_{j=1}^{C} A_j \tag{2}$$

1. Se desea lograr un sistema que sea capaz de clasificar correctamente, y en forma automática, a qué letra corresponde cada vector de características. El requerimiento mínimo a alcanzar es el de lograr que la media aritmética (Amean) de las tasas de acierto por clase esté por encima de 90 Este requisito se impone para un sorteo de los vectores de test, asumiendo equiprobabilidad entre las clases.

2. Se desea construir un sistema para transcribir un texto en español (tomando la Ñ como una N) a partir de los vector de características de imágenes provenientes de las letras de un texto. Se establece como nuevo objetivo que el sistema maximice su tasa de acierto, conociendo que el conjunto de testeo proviene de un texto en idioma español.

3. Comparar y comentar los resultados de ambos sistemas.

1.2. Software a utilizar

Se trabajó exclusivamente con Matlab y WEKA, el primero para preprocesar la información y como programa principal de donde se llamaría a WEKA, dadas las buenas cualidades de este último en aprendizaje y visualización .

1.3. Visualización Primaria

Utilizando la plataforma de software para aprendizaje automático WEKA [1] es inmediato comprobar que la distribución de cada clase (o letra) no es ni balanceada ni corresponde con el idioma español.

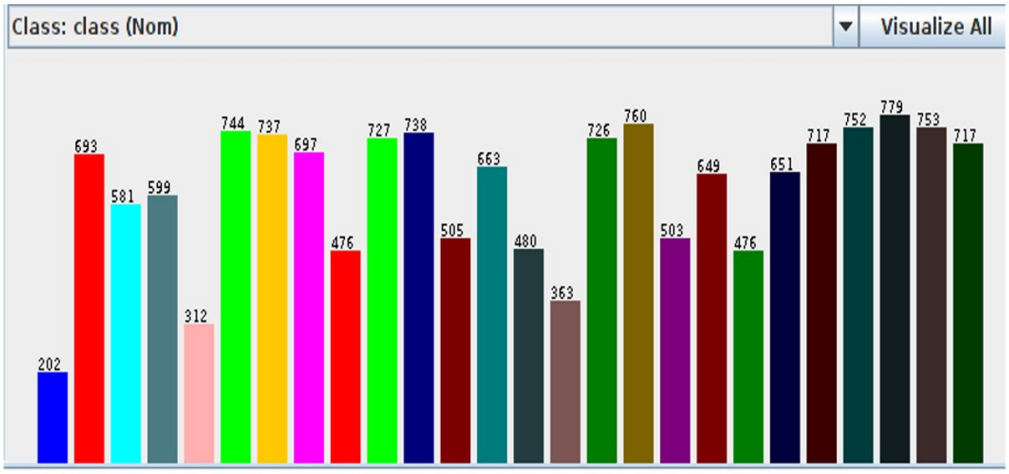


Figura 1: Frecuencias

1.4. Enfoque

Para la Parte I se procesaron los patrones de entrenamiento de modo de lograr la equiprobabilidad entre las clases y se buscó corregir las posibles confusiones ocasionadas por la similitud gráfica entre algunas letras mediante la matriz de costo.

Para la Parte II se entrenaron los patrones con un texto en español de forma que se proveyera la información necesaria sobre la frecuencia de cada letra en dicho idioma.

[1] $http://es.wikipedia.org/wiki/Weka_\%28aprendizaje\_autom\%C3\%A1tico\%29$

2. Clasificadores

A la hora de seleccionar clasificador se decidió evaluar 3 de los estudiados en el curso, como fueron *NaiveBayes, J48* y *k-NN* y comprobar si cumplían con los valores requeridos[2].

2.1. Clasificador: NaiveBayes

NaiveBayes es un clasificador que se basa en Bayes a la hora de decidir, aunque también es de destacar que asume independencia entre sus características. Es de notar que observando la ecuación 3 se observa que la probabilidad a posteriori $P(\omega_i/x)$ depende tanto de las probabilidad a priori $P(\omega_i)$ como de la verosimilitud $P(x/\omega_i)$ y es por eso que este criterio tiene en cuenta ambas probabilidades a la hora de reducir el error.

$$P(\omega_i/x) = \frac{P(x/\omega_i)P(\omega_i)}{P(x)} \tag{3}$$

Finalmente, la regla de decisión del clasificador NaiveBayes es:

Decide ω_1 si $P(\omega_1/x) > P(\omega_2/x)$, de lo contrario se elige ω_2.

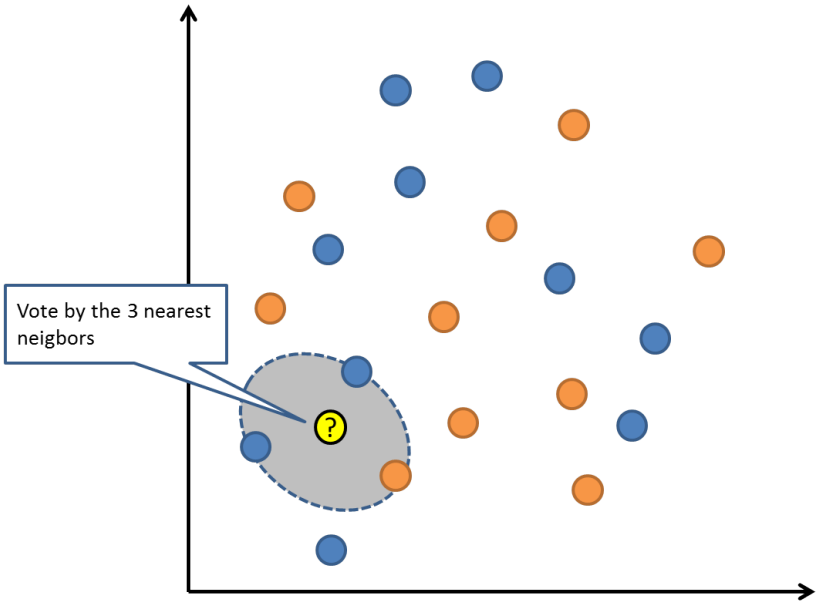
2.2. Clasificador: J48

Se trata de un clasificador sencillo, de bajo costo computacional y de fácil interpretabilidad. Los árboles de decisión parten de un nodo raíz donde se encuentran todos los patrones. Se selecciona la característica que maximiza el decremento de la impureza y a partir de dicha característica "abrir" el arbol generando nodos intermedios. Se repite el proceso hasta llegar a cumplir con el criterio de parada establecido. A su vez, el último nodo del arbol se denomina hoja. Al patrón que llegue a dicha hoja se le etiquetará con la etiqueta que corresponde a esa hoja. Específicamente, WEKA utiliza como variables *ConfidenceFactor* para establecer el podado deseado y *minNumObj* para ajustar el valor de la impureza.

[2]Para *k-NN* se tomó como condición minimizar valor de k.

2.3. Clasificador: k-NN

Al igual que $J48$, se trata de un clasificador sencillo, pero con gran costo computacional. Como la mayoría de los clasificadores que estamos utilizando, no tiene en cuenta la relevancia de una característica sobre otra. En este caso sólo toma en cuenta cuál es la clase mayoritaria en una celda centrada en el patrón a clasificar. Se utilizarán las opciones por defecto, incluyendo distancia euclidiana.



A partir del Duda, sabiendo que $p_n(x) = \frac{k_n/n}{V}$ se tiene que la estimación de la probabilidad a priori es:

$$P_n = \frac{p_n(x,\omega)}{\sum_{j=1}^{c} p_n(x,\omega_j)} \tag{4}$$

3. Algoritmos de Balanceo

Si el conjunto de test se caracteriza por tener equiprobabilidad en sus letras, el sistema a construir debería entrenarse con un conjunto de entrenamiento similar y así evitar que los clasificadores tiendan a asignar más patrones a la clase con mayor valor de priori.
Sin embargo, el conjunto disponible no cumple con dicha condición como se aprecia en la figura 1. Deberá ser balanceado y para ello se dispone de varias herramientas:

1. Oversampling
2. Undersampling
3. SMOTE
4. Matriz de Costos
5. Modificación de las probabilidades a priori.

También se pueden considerar combinaciones de los anteriores para ciertas situaciones.

3.1. Algoritmo de Balanceo: Oversampling

Oversampling es una técnica que balancea las cargas por el simple expediente de incrementar el número de patrones de la(s) clase(s) minoritaria(s) duplicándolos. Al igual que todas las técnicas descriptas para el balanceado, se deben conocer sus limitaciones para evitar un deterioro del resultado. En este caso el principal riesgo proviene del sobreentrenamiento debido a la duplicación de patrones.

3.2. Algoritmo de Balanceo: Undersampling

Undersampling, al contrario de Oversampling esta técnica directamente desecha patrones de la(s) clase(s) mayoritaria(s) hasta que todas las clases tengan la misma cantidad de datos. Aunque el desechado se realiza en forma aleatoria, no hay garantías de que se pierda información.

3.3. Algoritmo de Balanceo: SMOTE

SMOTE, siglas de *Synthetic Minority Oversampling Technique* es un método que genera patrones mientras que evita el sobreentrenamiento.

Para ello busca los k vecinos más cercanos para cada una de las clases minoritarias y genera un nuevo patrón a partir de esos k vecinos. La posición de dicho patrón varía aleatoriamente para evitar el sobreentrenamiento para el caso en que se generan muchas muestras determinadas por los mismos vecinos. Según la distribución de las clases puede ser contraproducente, no recomendándose para conjuntos cóncavos.

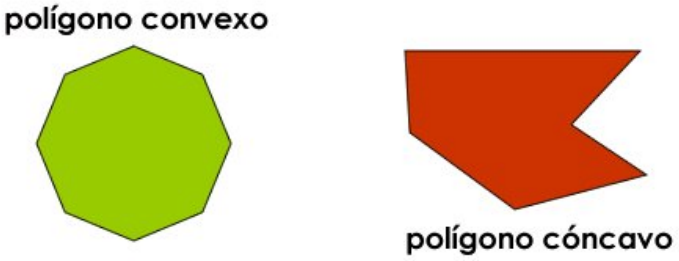


Figura 2: Conjuntos convexos y cóncavos.

3.4. Algoritmo de Balanceo: introducir Matriz de Costos

Otra de las opciones consideradas fue modificar la matriz de costos de manera que equivocarse tenga un costo si la clase es mayoritaria y otro si es la clase es minoritaria. Así, asignando un costo mayor si se clasifica incorrectamente un patrón de la clase minoritaria el Clasificador le dará más oportunidades a dicha clase.
Limitaciones: como se había comentado dependerá de la aplicación la conveniencia de utilizar esta técnica. Asignar un mayor costo a equivocarse con la clase minoritaria también significa que el clasificador tenderá a equivocarse con la clase mayoritaria.

Esto puede ser irrelevante en algún caso o puede llegar a ser un comportamiento no deseado.
Para el caso de reconocer las letras se le consideró para compensar pequeños ajustes entre letras muy similares gráficamente. Sin embargo, se debe tener en cuenta que alterar el costo de equivocarse en una letra se está mejorando el α a costa del β o viceversa, según hacia dónde se mueva el costo o umbral. Que la mejora para una letra es el deterioro de la performance para la otra letra se puede apreciar en la figura 3.

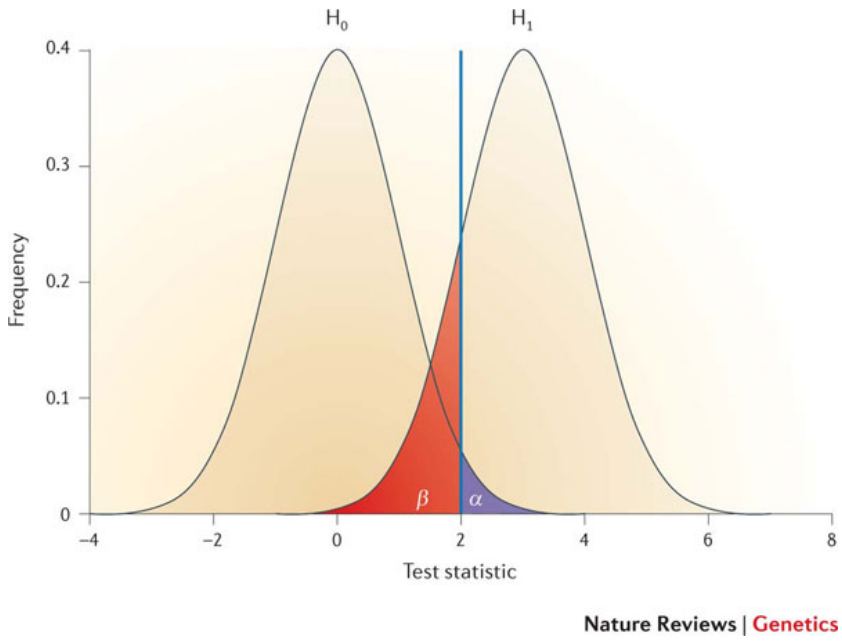


Figura 3: Efecto de modificar el umbral de decisión o costo al equivocarse.

3.5. Algoritmo de Balanceo: introducir Probabilidades a priori

La última opción que se consideró fue incluir directamente las probabilidades a priori. Matlab tiene varias funciones que aceptan un vector con las priori de c/clase. Las funciones estudiadas fueron:

1. *fitcknn*
2. *fitcnb*
3. *fitctree*

Si bien puede ser un método adecuado, se optó por WEKA debido a que ya se ha trabajado con esa plataforma anteriormente. Fue determinante que Matlab utilice denominaciones distintas a las de WEKA.

4. Ajuste de errores por Similitud Gráfica

La frecuencia de las letras y la semejanza entre las mismas pueden llevar a que el Clasificador cometa errores en la clasificación.

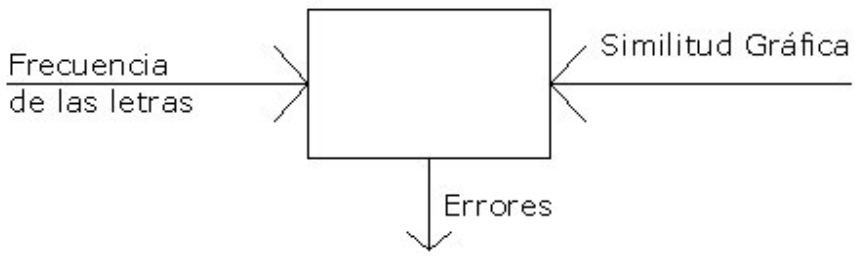


Figura 4: Errores debidos a la similitud gráfica y la frecuencia de las letras

Como en la Parte I la frecuencia de las letras no influye, ya que se trata de clases equiprobables se centró en analizar el problema de la semejanza de las letras. Al inspeccionar las matrices de confusión se advirtió que el clasificador cometía errores a la hora de discernir entre letras muy similares gráficamente como podían ser la H con K o la F con la P. Se intentó suavizar dichos errores con la matriz de costos. De encontrarse una matriz de costos óptima para los datos balanceados se podría utilizar para compensar los mismos inconvenientes con el texto en español de la Parte II.

Sabiendo que mejorar la performance de una letra en particular llevaría a un posible deterioro en otra(s), tal como se vió en la figura 3. Es deseable que no sólo mejorara los casos donde se aplicaba el costo, sino que minimizara los errores en general llevando a una mejor performance del sistema como un todo.

Vistas las anteriores consideraciones se siguieron 3 aproximaciones:

1. Modificar valores puntuales a partir de los valores hallados en la matriz de confusión de los datos balanceados, si la cantidad de errores sobrepasa cierto valor.

2. Modificar toda la matriz de costos a partir de los valores de la matriz de confusión de los datos balanceados, normalizando con una función cuadrática.

3. Modificar toda la matriz de costos a partir de los valores de la matriz de confusión de los datos balanceados, normalizando con una función logarítmica.

4.1. Matriz de costos, modificación de valores puntuales

Para el caso 1 bastó con modificar un caso particular, específicamente las $H's$ que eran confundidas y etiquetadas como $K's$.
A pesar de variar el valor del costo entre 2 y 25 no se encontró una mejora significativa, esto es una mejora del $0,5\,\%$, a pesar de que cambió la semilla repetidamente para descartar un resultado particular.

4.2. Matriz de costos normalizada con función cuadrática.

Se creó la función *matrizcosto(opción)* que recibe como entrada la matriz de confusión que resulta de aplicar $k - NN(5)$ a los datos balanceados. La opción requerida para la función cuadrática es 1.
La función imprime en consola la matriz de costos y guarda los resultados en un archivo *matriz_costo.txt*. Se adjunta el archivo *matrizCosto_raiz.txt*.

4.3. Matriz de costos normalizada con función logarítmica.

Se creó la función *matrizcosto(opción)* que recibe como entrada la matriz de confusión que resulta de aplicar $k - NN(5)$ a los datos balanceados. La opción requerida para la función logarítmica es 2.
La función imprime en consola la matriz de costos y guarda los resultados en un archivo *matriz_costo.txt*. Se adjunta el archivo *matrizCosto_log.txt*.

5. Programa automático

Se crearon dos programas automáticos, uno para la Parte I y otro para la Parte II. Cada uno de ellos tiene la finalidad de clasificar un conjunto de vectores de características y generar un archivo con el conjunto de letras obtenidas de aplicar el clasificador correspondiente.

La clasificación de los datos se realiza utilizando el clasificador de WEKA en Matlab, en particular se utiliza IBk con $k = 5$. Para este fin se importan las siguientes clases *java*:

- *import weka.classifiers.lazy.IBk*: Importa la clase IBk.class, para clasificar.
- *import weka.core.Instances*: Importa la clase Instance.class, para crear una instancia de los datos de testeo y entrenamiento.
- *import java.io.FileReader*: Importa la clase FileReader.class, para leer los datos de testeo y entrenamiento.

Ambos reciben como parámetro el nombre del archivo que contiene los vectores de características a clasificar y en ambos casos, concluida la clasificación se genera un archivo con el resultado de la clasificación y se imprime en consola el error de clasificación, la matriz de confusión obtenida y el vector de etiquetas (letras) resultantes.

Para la Parte I el programa se denomina:
$clasificadorWeka\_parte\_a(nombre\_archivo\_a\_clasificar)$
y se genera el archivo *letras_clasificadas_parte_a.txt*

Para la Parte II el programa se denomina:
$clasificadorWeka\_parte\_b(nombre\_archivo\_a\_clasificar)$.
y se genera el archivo $cuento\_sin\_espacios\_clasificado.txt$.

Particularmente para la Parte II, al archivo de texto generado con el resultado, $cuento\_sin\_espacios\_clasificado.txt$, se le aplica el método $main\_agregar\_espacios()$ para agregarle al texto los espacios correspondientes, y se imprime en consola.

6. Parte I: Clases Equiprobables.

6.1. Balanceo: métodos empleados

Habiéndose indicado los pros y los contras de cada uno de los algoritmos, es necesario seleccionar uno para balancear los patrones de la figura 1 antes de entrenar el Clasificador de la Parte I.

No suele ser recomendable descartar patrones que pudieran tener información que posteriormente fuera necesaria para una correcta clasificación, por lo que se evitó descartar patrones de las clases minoritarias.
Por el contrario, se corrió el riesgo de sobreentrenar teniendo en cuenta a que los datos originales no estaban seriamente desbalanceados. El filtro aplicado fue *Resample* de WEKA que el implementa *Oversampling* si el valor seleccionado es mayor a 100 %. Utilizando las opciones:

1. $biasToUniformClass = 1$
2. $sampleSizePercent = 125\,\%$

se obtienen los datos que se muestran en la figura 5.

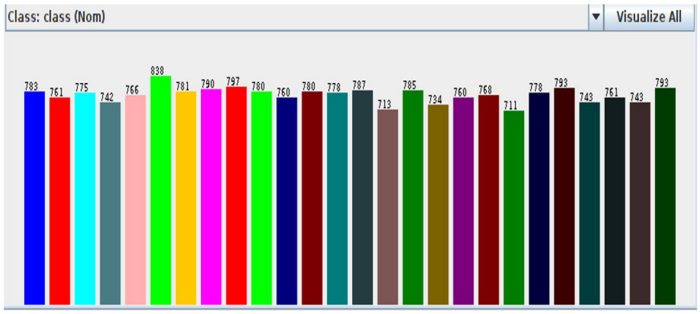


Figura 5: Datos balanceados con el filtro Resample de WEKA.

6.2. Clasificadores: desempeño con los datos originales.

Con los datos originales, tal como se aprecian en la figura 1, se procedió a evaluar los clasificadores seleccionados.

Clasificador	Correctamente clasificados
NaiveBayes	63.875 %
$J48$	86.186 %
$k - NN(5)$	94.356 %
$k - NN(10)$	93.118 %

Cuadro 1: Desempeño de los datos originales utilizando 10-Folds Cross-Validation

En el cuadro 1 los resultados de la evaluación, NaiveBayes y $J48$ tienen un pobre desempeño, mientras que $k - NN$ logran una mucho mejor performance [3].

6.3. Clasificadores: desempeño con datos balanceados.

Se procedió a evaluar los clasificadores seleccionados con los datos balanceados, esto es el archivo *Recpat 2014 - Entrenamiento - Letras-Equilibradas125.arff*.

Clasificador	Correctamente clasificados
NaiveBayes	63.67 %
$J48$	93.155 %
$k - NN(5)$	94.91 %
$k - NN(10)$	94.115 %

Cuadro 2: Desempeño de los datos balanceados utilizando 10-Folds Cross-Validation

[3] Se descartaron los valores menores a 5 para evitar los efectos del ruido.

Visto los resultados de los cuadros 1 y 2 es notoria la mejora de J48 al balancear los datos.
De todas maneras, se decidió seleccionar a $k - NN(5)$ como Clasificador que ofrece la mejor performance, una mejor resistencia al ruido[4] y el menor costo computacional[5].

6.4. Matriz de costos: desempeño

Se procedió a comprobar si el ajuste de costos podría tener una mejora significativa sobre los datos balanceados, archivo *Recpat 2014 - Entrenamiento - Letras-Equilibradas125.arff*. Las opciones a testear fueron las descriptas en la sección 4:

1. Modificar valores puntuales de la matriz de costos.
2. Utilizar una función cuadrática.
3. Utilizar una función logarítmica.

Matriz de Costos	Tipo de ajuste	Correctamente clasificados	Rango de los coeficientes
Caso 1	-	94.95 %	[5]
Caso 2	Logarítmico	95 %	[1;6.4]
Caso 3	Cuadrático	94.74 %	[0.62;25]

Cuadro 3: Valores obtenidos modificando la Matriz de Costos

También se puede apreciar en el cuadro el rango en que variaron los costos, tanto con la aproximación cuadrática como con la logarítmica. El cambio de semilla no mejoró los resultados, por lo que descartamos la opción de mejorar la performance con una matriz de costos compensando los errores cometidos por la similitud gráfica entre ciertas letras. Finalmente, ya que en la Parte II se utilizará un texto en español para entrenar y que no se logró hallar una matriz de costos adecuada para minimizar los errores generados por la similitud gráfica tampoco se le utilizará cuando el texto a clasificar sea en español.

[4] comparado con $k - NN(1)$
[5] comparado con $k - NN(10)$

6.5. Evaluación de los patrones de Test

Se procedió a evaluar las opciones estudiadas en la Parte I, esto es $k - NN(5)$ entrenado con el archivo *Recpat 2014 - Entrenamiento - Letras-Equilibradas125.arff*.

Datos Balanceados	Correctamente clasificados
Sin Costo	94.7308 %
Matriz de costo cuadrática	94.5385 %
Matriz de costo logarítmica	94.2692 %

Cuadro 4: Desempeño de los datos balanceados testeando con el archivo *RP14_letter_test_parte1.arff*

7. Parte II: Texto en Idioma Español

7.1. Características del Idioma Español

Mientras que en la Parte I se sabía que el texto a decodificar se componía de una cantidad muy similar de cada letra del alfabeto, en esta segunda parte se conoce de antemano que el texto está en español.

A la hora de codificar o decodificar un texto se debe tener en cuenta que cada idioma tiene su propia frecuencia de aparicion de letras que lo distingue y lo define. A su vez, dicha frecuencia puede variar sensiblemente según el tipo de texto en que se esté trabajando. Se describen a continuación algunos ejemplos para ilustrar el concepto:

- El estilo narrativo. Si hay muchos verbos en infinitivo, habrá muchas R.
- El vocabulario específico del documento. Si se habla de ríos, habrá muchas I; si uno de los protagonistas se llama Wenceslao, aumentará el número de W y así sucesivamente.
- En el diccionario de la RAE la letra más frecuente es la A, pero en cualquier texto castellano, la frecuencia de las partículas *que*, *el*, *se*, *me*, etc. hace que la e sea más frecuente[6].

[6]http:es.wikipedia.orgwikiFrecuencia_de_aparici %C3 %B3n_de_letras

En la figura 6 se indica la frecuencia de aparicion en un texto normal en español. Es de notar que dicha información se encuentra en el libro *Secret and Urgent: the Story of Codes and Ciphers* de Fletcher Pratt, 1939.

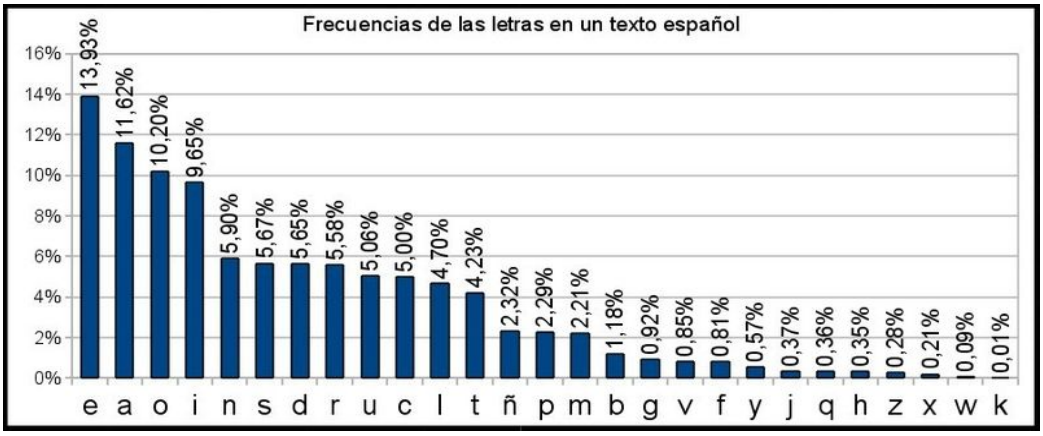


Figura 6: Frecuencia de las letras en un texto en español.

7.2. Algoritmo Seleccionado

De los algoritmos analizados en la sección 3 se descartaron inmediatamente *Oversampling* y *Undersampling*, ya que se tendría sobreentrenamiento o pérdida de información, respectivamente. Esto es debido a las características del idioma español con letras con una altísima frecuencia de aparicion y otras casi sin utilización. Para visualizar el problema, considérese el caso extremo de las letras A y K:

La A tiene una frecuencia de 12,53 %, mientras que la K de sólo 0,01 %. Entonces para llevar las 202 muestras de la A disponibles a la misma relación con las 779 K's, se debería multiplicar por 976.087. Por el contrario, si se desean descartar muestras de las clases minoritarias se deberían descartar casi todas.

Otro de los métodos que debió descartarse fue SMOTE. Si bien hubiera sido laborioso crear la enorme cantidad de muestras necesarias para que las clases mayoritarias lograran tener la frecuencia indicada en la figura 6, el método debió abandonarse porque en este momento WEKA no puede soportar tal cantidad de procesamiento. Se intentó aumentar la memoria java, pero el aviso de *OutOfMemory* siguió apareciendo luego de unas cuantas clases procesadas. La razón de intentar ajustar las frecuencias con SMOTE recaía en la suposición de que las clases estaban relativamente separadas entre si y que, al menos, no habría problemas generados por conjuntos no convexos.

Finalmente se decidió entrenar el clasificador con un texto suficientemente grande como para que su frecuencia de aparicion de las letras fuera similar a la referencia utilizada, figura 6. Teniendo 16.000 patrones como materia prima, se consideró que había suficiente margen para construir un texto de 16.000 letras sin tener sobreentrenamiento. El texto se obtuvo de los siguientes links:

- $http://www.ilce.edu.mx/documents/Convencion\_de\_viena\_relac\_diplo.pdf$
- $http://diplomatshandbook.org/pdf/Diplomats\_Handbook\_Spanish.pdf$

7.2.1. Detalles de la construccion del texto en español

La función *textoEspanol()* convierte un texto en español en un conjunto de vectores de características y los guarda en el archivo *csvarchivo.csv*.

textoEspanol() llama a la función *texto()* que contiene el texto a traducir y lo convierte en un array de caracteres. A cada caracter se le asocia un vector de característica proveniente de los datos originales. Se realiza una iteración con todos los caracteres del texto en español.

Cabe destacar que el archivo .arff con los datos originales fue pasados a .csv para ser leídos por la función.

Para asegurar que a mismas letras se le asocie diferentes vectores de caracerísticas y así utilizar todos los vectores de características disponibles, se mezclan los vectores de caracerísticas de los datos originales en cada iteración.

Una vez obtenido el archivo *csvarchivo.csv*, como en Matlab la etiqueta de los patrones fue ingresada como el caracter ASCII de las letras y no la letra correspondiente, se debió utilizar Excel para convertir el caracter ASCII en letra con la función *char(ASCII)*. Una vez obtenido el conjunto de vectores de caracerísticas con la etiqueta adecuada se guardan en el archivo $caracteristica\_text\_espanol.arff$.

En la figura 7 la frecuencia de aparicion de letras del texto construido con la función *textoEspanol()*.

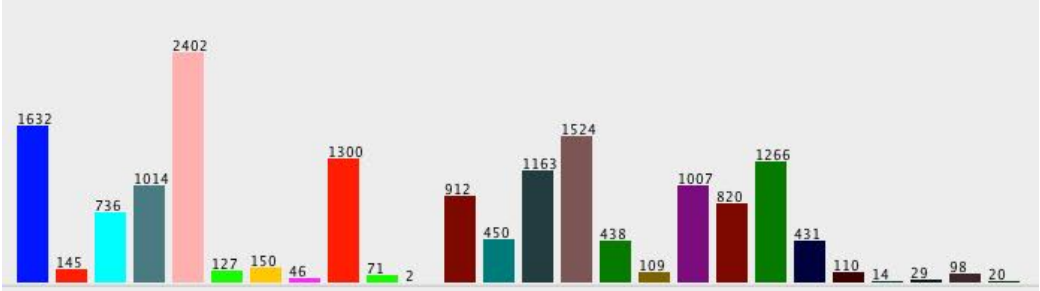


Figura 7: Frecuencia de las letras del texto en español construido.

7.3. Evaluación de los patrones de Test

Se procedió a evaluar las opciones estudiadas en la Parte II, esto es $k - NN(5)$ entrenado con el archivo *caracteristica_text_espanol.arff*.

Texto en español	Correctamente clasificados
Sin Costo	93.16 %

Cuadro 5: Desempeño de los datos balanceados testeando con el archivo *RP14_letter_test_parte2.arff*

8. Conclusiones

- Se logró cumplir con el objetivo propuesto para la Parte I, ya que $k - NN(5)$ supera el 90 % requerido: 94, 73 % de Accuracy.
- Se estudiaron distintas variaciones de la matriz de costos para corregir errores producto de la similitud gráfica entre ciertas palabras. Se comprobó que la mejora en clasificar cierta letra tenía el efecto adverso sobre otras letras, por lo que globalmente no había mejora.
- Se creó un programa que construye un texto en español a partir de los vectores de características y se entrenó el clasificador con dicho texto.
- Se logró implementar el automatismo para la Parte I que clasifica cada vector de características con la letra correspondiente asumiendo equiprobabilidad entre las clases.
- Se logró implementar el automatismo para la Parte II que transcribe un texto en español a partir de un conjunto de vectores de características.
- Finalmente indicar que la elección de k-vecinos con $k = 5$ parece haber sido una elección acertada ya que ha tenido muy buena performance, incluso con el texto en español. Posiblemente por su resistencia al ruido.

9. Referencias

[1] **Pattern Classification**.
Richard O. Duda, Peter E. Hart & David G. Stork.

[2] **Secret and Urgent: the Story of Codes and Ciphers Blue Ribbon Books**.
Fletcher Pratt.

[3] **Letter Recognition Using Holland-Style Adaptive Classifiers**.
Frey, Peter W y David J Slate.

[4] **WEKA**
http://www.cs.waikato.ac.nz/ml/weka/

[5] **Predictive Analytics: NeuralNet, Bayesian, SVM, KNN**
http://horicky.blogspot.com/2012/06/predictive-analytics-neuralnet-bayesian.html

[6] **WEKA classification likelihood of the classes**
http://stackoverflow.com/

[7] **Statistical power and significance testing in large-scale genetic studies**
http://www.nature.com/