PROBLEMA DE APRENDIZAJE DE PESOS EN CARACTERÍSTICAS
1-NN, BÚSQUEDA LOCAL Y GREEDY
MEMORIA P1 METAHEURÍSTICAS

Ignacio Garach Vélez MH Viernes 17:30

10 de abril de 2022

Índice

1. Introducción

En estas prácticas vamos a abordar la implementación y aplicación de varios algoritmos para el cálculo de pesos de calidad de un clasificador 1-NN de modo que se optimice tanto la tasa de acierto del problema de clasificación, como la simplicidad del clasificador, esto es, la detección de que características no influyen de forma determinante en la labor de clasificación y por tanto podrían no tenerse en cuenta.

Como preliminares introducimos brevemente el problema de clasificación. Se trata de construir un modelo que a partir de las características (datos descritos numéricamente) de un objeto sea capaz de predecir su clase (tipo de objeto entre un número finito de posibilidades). Para ello se entrena al modelo con un conjunto de datos correctamente clasificados (aprendizaje supervisado) y despues se prueba su valía con un conjunto independiente de test. Existen numerosos enfoques para resolver este problema con gran éxito, nosotros nos vamos a centrar en probablemente el más sencillo, se trata del algoritmo $1 - NN$ conocido popularmente como el del vecino más cercano.

Este algoritmo, simplemente almacena los datos de entrenamiento en memoria junto con su clase asociada, matemáticamente esto se puede representar como un conjunto de puntos de $\mathbf{R}^n$ junto con un entero representando su clase asociada. Y para cada punto que quiera clasificar, calcula la distancia euclídea con todos los puntos de entrenamiento y escoge la clase del punto para el cuál es mínima, su vecino más cercano.

Nuestro trabajo va a consistir en conseguir pesos que ponderen la importancia en la distancia final de cada característica en cada caso concreto de entrenamiento, es decir si la distancia euclídea usual es:

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

nosotros consideramos el cálculo de los w_i tal que sea:

$$d(x,y) = \sqrt{\sum_{i=1}^{n} w_i(x_i - y_i)^2}$$

Nuestro objetivo es dual, por un lado queremos conseguir buenas tasas de acierto en el problema de clasificación y por otro obtener pesos bajos en las características menos influyentes para la clasificación. El problema de aprendizaje de pesos en características es determinar dichos pesos a partir del conjunto de datos de entrenamiento.

En esta primera práctica trataremos un algoritmo de búsqueda local con trayectorias simples y un algoritmo greedy (voraz) y compararemos su desempeño con el algoritmo 1-NN habitual.

El algoritmo greedy Relief se basa en la idea de aumentar la importancia de aquellas características que mejor separan a ejemplos que son de distinta clase y por otro lado disminuirla en el caso de aquellas que separan ejemplos que son de la misma clase.

Por otro lado, el algoritmo de búsqueda local se basa en ir calculando el valor de la función objetivo (promedio de precisión y simplicidad) para distintos conjuntos de pesos a partir de un conjunto inicial aleatorio, modificandolo, cambiando cada vez un peso con la ayuda de una distribución normal y avanzando hacia los pesos que maximicen el valor de la función objetivo.

2. Datasets considerados

Para poder probar los algoritmos desarrollados se utilizarán los siguientes conjuntos de datos, todos ellos con clase binaria y algunos de ellos con distribución de clases no homogénea.

2.1. Ionosphere

Conjunto de datos recogidos en Goose Bay por un sistema con 16 antenas de alta frecuencia, el objetivo era captar electrones libres en la ionosfera, una devolución positiva indicaba la existencia de algún tipo de estructura. Los datos son las partes reales e imaginarias de 17 señales electromagnéticas complejas.

- 352 instancias
- 34 características
- 2 clases

2.2. Parkinsons

Conjunto de datos recogidos por el Centro Nacional de estudio de la voz de Oxford. Se trata de medidas asociadas a 195 grabaciones de voz de personas, algunas de ellas con enfermedad de Parkinson. La clase sería la presencia de dicha enfermedad.

- 195 instancias
- 22 características
- 2 clases

2.3. Spectf-Heart

Conjunto de datos de medidas asociadas a imágenes tomadas por tomografía computerizada cardíaca tomadas por el colegio de médicos de Ohio. Cada paciente se clasifica entre normal o con anomalías.

- 267 instancias
- 44 características
- 2 clases

3. Elementos comunes del problema

En este apartado vamos a describir las funciones que serán comunes durante todo el desarrollo de las prácticas de la asignatura:

- Para representar nuestros conjuntos de entrenamiento y test utilizaremos un vector de la STL de una nueva clase a la que llamaremos Instance que representará una instancia del problema, es decir su vector de características y su clase.
- Los datos proporcionados se encuentran en formato ARFF, un conjunto de cabeceras indicando las características y a continuación cada instancia es una fila con espacios como separadores y con la clase al final de la línea. Esto facilita su lectura para su posterior postprocesado. Hemos desarrollado para ello la funcion *read_arff* que recibe como parámetro el nombre del archivo y un par de vectores donde se almacenan las características y su clase. A posteriori en la función principal se genera el *vector de Instance* y se rellena con dichos datos.
- Los datos requieren ser normalizados para que no influyan las distintas escalas en nuestros algoritmos y no distorsionen los datos. Para esto se ha implementado la función *normalize* que modifica el vector de instancias, mediante escalado y translaciones hasta el intervalo $[0, 1]$, para ello se hace uso de la siguiente fórmula en cada característica:

$$x_i = \frac{x_i - mín(x_i)}{máx(x_i) - mín(x_i)}$$

- Para el cálculo de las distancias se implementa la función *weightedDistance* que recibe un vector de pesos y 2 vectores de características y devuelve la distancia ponderando por sus pesos.
- Para nuestro problema de aprendizaje de pesos, utilizaremos una versión del clasificador 1-NN que permitirá su generalización con la utilización de pesos. Recibe un valor booleano indicando si debe utilizar un vector de pesos al calcular las distancias. Atendiendo a estas características tendríamos el siguiente pseudocódigo:

Clasificador 1-NN (Instance, TrainingSet, Weighted?, Weights[])

Si no weighted
Entonces: Inicializar a 1 todos los pesos W
Para toda instancia de TrainingSet distinta de Instance (Leave One Out)
Hacer:

Calcular la distancia con respecto a Instance.
Si la distancia disminuye respecto al anterior guardamos su posición en pos.

Fin - Para todo

Devolver la clase de TrainingSet[pos]

3.1. Métricas utilizadas

Como medidas de la bondad de funcionamiento de los algoritmo se van a tomar las siguientes 4, relacionadas con los parámetros que se quieren optimizar, a saber, exactitud de clasificación, simplicidad y tiempo:

%**class** : Porcentaje de acierto en la labor de clasificación. Se utiliza el método de prueba basado en entrenamiento y test de 5 particiones cruzadas (se crean mediante la función *makeKFolds* inspirada en las prácticas de la asignatura de aprendizaje automático y que divide de forma homogénea y equilibrada las clases) y se realiza la media.

$$\%_{class} = 100\frac{\text{nº de instancias clasificadas correctamente en Test}}{\text{nº de instancias en Test}}$$

%**red** : Porcentaje de características que podrían obviarse, vamos a considerar aquellas cuyo peso asociado sea menor a 0.1. En efecto, para el algoritmo base no se reduce nada, porque todos los pesos son 1.

$$\%_{red} = 100\frac{\text{nº de características con peso menor que 0.1}}{\text{nº de características}}$$

Agregado : Valor de la función objetivo de nuestro problema. Se trata del promedio ponderado de las tasas de acierto y reducción, en nuestro caso consideramos una ponderación equiparada luego es simplemente la media de ambos valores

Tiempo : Tiempo de ejecución del algoritmo. En nuestro caso la media de 5 ejecuciones.

Para la implementación del cálculo de la tasa de clasificación se utilizará la función *score*, simplemente recorre el vector de predicciones del clasificador y cuenta los aciertos, devolviendo la división entre el número de instancias.
Para el cálculo de la tasa de reducción se hace algo análogo en *reduction* con el vector de pesos obtenido contando todos aquellos menores que 0.1 y devolviendo la división entre el número de características.
La función objetivo agregada se implementa de forma génerica con un parámetro α que pondera la importancia de precisión y simplicidad, aunque en nuestro caso usaremos $\alpha = 0.5$ luego coincidirá con la media de *score* y *reduction*, esto se realiza en *funcionObjetivo*.
Por último para el cálculo de tiempos se usará la librería *chrono* de C++ tomando el tiempo actual

antes y después de las ejecuciones con *high_resolution_clock* y calculando su diferencia. Lo mediremos en segundos.

4. Algoritmo Greedy : RELIEF

Este algoritmo es una primera aproximación al problema que trata de calcular rápidamente una solución sencilla. Como habíamos adelantado anteriormente, se va a aumentar el valor de un peso cuando una característica separa muy bien a puntos de clase distinta (enemigos) y a disminuirlo en las características que oscilan mucho para individuos de la misma clase (amigos). Posteriormente es necesario normalizar los pesos al intervalo $[0, 1]$ para no distorsionar los resultados del clasificador, en caso de que el peso resultado sea negativo, se convierte a 0 pues se asume no es relevante con respecto al resto.

Consideramos el siguiente pseudocódigo, la primera función *nearestEnemyandFriend* calcula los elementos más cercanos, su amigo y enemigo más cercano, al elemento que se le pasa como parámetro. Finalmente se incluye también la estructura del algoritmo RELIEF en la función *computeReliefWeights* que hace uso de la anterior.

nearestEnemyandFriend (Instancia, TrainingSet, posEnemy, pos Friend)

Para toda instancia de TrainingSet distinta de Instance
Hacer:

Calcular la distancia con respecto a Instance y si es de su misma clase y más cercano al actual posFriend almacenar en posFriend.
Calcular la distancia con respecto a Instance y si es de distinta clase y más cercano al actual posEnemy almacenar en posEnemy.
Fin para todo
Al finalizar la ejecución, hemos calculado el amigo y enemigo más cercanos para la instancia

ComputeReliefWeights (Training[], W[])

Inicializar a 0 todos los pesos W.
$\forall i \in (0, |training| - 1)$
Calcular la posición del enemigo y el amigo mas cercano de Instance.
$\forall j \in (0, |W| - 1)$

Actualizar W[j] sumándole la distancia al enemigo mas cercano y restándole la distancia al amigo mas cercano. Fin-paratodo

Fin para todo
Normalizar pesos al intervalo $[0, 1]$

5. Algoritmo de Búsqueda Local

Para este algoritmo, partiremos de una solución (vector de pesos) inicial definida aleatoriamente a partir de una distribución uniforme en el intervalo $[0, 1]$, calcularemos el porcentaje de acierto en clasificación para el conjunto de entrenamiento mediante la estrategia leave one out, ya está generalizada en la implementación del clasificador, y por otro lado la tasa de reducción. A partir de ellas se calcula la función objetivo. Vamos generando vecinos sumando a una componente de los pesos un valor obtenido mediante una normal de media 0 y baja varianza. Se van comprobando los cálculos que se han indicado para cada mutación (modificación de los pesos) y nos vamos quedando con los pesos que optimicen la función objetivo, por tanto aumentamos simplicidad y acierto.

Puede finalizarse el algoritmo por 2 criterios, máximo número de iteraciones o máxima generación de vecinos. Por tanto tendríamos el siguiente pseudocódigo:

LocalSearch (Training[], W[])

Inicializar con distribución uniforme en $[0, 1]$ todos los pesos W.
Calcular la función objetivo con el conjunto training con LeaveOneOut.
Mientras no se llegue al máximo de iteraciones o no se mejore trás calcular nuevos pesos determinadas veces:
Hacer:
Actualizar W en cierta componente cada vez mediante la suma de un valor de distribución normal de media 0. Recalcular la función objetivo.
Si hay mejora, almacenarla y aceptar los pesos.
Sino, volver a los antiguos.

Si se mejoró o se probó en todas las W[i], reiniciar aleatoriamente el orden en el que se cambian las componentes
Fin Mientras

En ningún caso se permite que los pesos se salgan del $[0, 1]$, se truncan si esto ocurre.

6. Implementación y guía de usuario

Para la implementación se ha utilizado el lenguaje C++, las librerías habituales de entrada y salida y lectura de fichero, toma de tiempos y funciones auxiliares. Se usó un generador de números aleatorios inicializado con la semilla 2022.

Se ha especificado por secciones la estructura en funciones del código, por simplicidad se ha optado por implementar todo en un mismo archivo sin usar cabeceras.

Se utiliza optimización en la compilación con g++, concretamente se usa la siguiente orden para compilar:

```
g++ -std=c++11 -O3 main.cpp
```

Para ejecutar se llama en terminal a ./a.out seguido de la semilla que se desee.

7. Análisis y presentación de resultados

Trás la ejecución de los algoritmos, encontramos que era preocupante el tiempo de ejecución del algoritmo de búsqueda local, pues se elevaban a 30 minutos todas las ejecuciones. Sin embargo añadiendo optimización O3 al compilar lo reducimos en torno a los 5 minutos totales de ejecución. Se obtienen entonces los siguientes resultados, las primeras 5 filas son sobre las particiones y la sexta es la media:

Cuadro 1: Cuadro de resultados para 1-NN

Ionosphere				**Parkinsons**				**Spectf-heart**			
%cl	***%red***	***Agr.***	**T**	***%cl***	***%red***	***Agr.***	**T**	***%cl***	***%red***	***Agr.***	**T**
84,51	0	42,25	0,003	95	0	47,5	0	58,57	0	29,29	0,002
87,32	0	43,66	0,003	92,5	0	46,25	0	71,43	0	35,71	0,002
87,32	0	43,66	0,003	95	0	47,5	0	70	0	35	0,002
91,55	0	45,77	0,001	97,5	0	48,75	0	65,71	0	32,86	0,002
80,6	0	40,3	0,001	100	0	50	0	63,77	0	31,88	0,002
86,26	0	43,13	0,002	96	0	48	0	65,9	0	32,95	0,002

Cuadro 2: Cuadro de resultados para Greedy Relief

Ionosphere				**Parkinsons**				**Spectf-heart**			
%cl	***%red***	***Agr.***	**T**	***%cl***	***%red***	***Agr.***	**T**	***%cl***	***%red***	***Agr.***	**T**
83,1	2,941	43,02	0,01	95	0	47,5	0,002	72,86	13,64	43,25	0,01
90,14	2,941	46,54	0,01	92,5	0	46,25	0,002	74,29	20,45	47,37	0,01
88,73	2,941	45,84	0,01	97,5	0	48,75	0,002	74,29	20,45	47,37	0,01
90,14	2,941	46,54	0,01	95	0	47,5	0,002	74,29	22,73	48,51	0,01
80,6	2,941	41,77	0,01	100	0	50	0,002	71,01	31,82	51,42	0,01
86,54	2,941	44,74	0,01	96	0	48	0,002	73,35	21,82	47,58	0,01

Cuadro 3: Cuadro de resultados para búsqueda local

Ionosphere				**Parkinsons**				**Spectf-heart**			
%cl	***%red***	***Agr.***	**T**	***%cl***	***%red***	***Agr.***	**T**	***%cl***	***%red***	***Agr.***	**T**
94,37	88,24	91,3	18,52	90	95,45	92,73	4,524	65,71	75	70,36	28,28
90,14	85,29	87,72	19,53	87,5	63,64	75,57	1,475	77,14	70,45	73,8	26,08
84,51	88,24	86,37	18,72	97,5	77,27	87,39	2,807	74,29	68,18	71,23	18,67
87,32	79,41	83,37	18,96	92,5	100	96,25	3,761	75,71	68,18	71,95	24,53
88,06	91,18	89,62	18,21	97,14	77,27	87,21	4,197	71,01	72,73	71,87	18,26
88,88	86,47	87,68	18,79	92,93	82,73	87,83	3,353	72,77	70,91	71,84	23,16

En primer lugar, observamos que el algoritmo 1-NN obtiene buenos resultados para Ionosphere, excelentes para Parkinsons y bastante malos para Spectf-Heart (un clasificador aleatorio tendría teóricamente un 50 por ciento de acierto). Evidentemente no puede reducir pesos por construcción, luego la función objetivo está acotada superiormente por 0.5.

Al comparar con el greedy Relief observamos que en general mejora ligeramente los porcentajes de acierto en clasificación para los 3 datasets. Por otro lado, reduce una característica en Ionosphere aunque era trivial de reducir pues si observamos los datos en crudo, casi todos sus valores son iguales y por tanto no discriminan y pueden eliminarse. En Parkinson no logra reducir y por tanto se limita mucho la función objetivo. Finalmente en Spectf-Heart si logra reducir en torno a un 20 por ciento de características pero sigue obteniendo mala función objetivo agregada pues se le da gran importancia a la reducción en la ponderación.

Cuadro 4: Cuadro resumen de resultados generales

	Ionosphere				**Parkinsons**				**Spectf-heart**			
	%cl	***%red***	***Agr.***	**T**	***%cl***	***%red***	***Agr.***	**T**	***%cl***	***%red***	***Agr.***	**T**
1-NN	86,26	0	43,13	0,002	96	0	48	0	65,9	0	32,95	0,002
RELIEF	86,54	2,941	44,74	0,01	96	0	48	0,002	73,35	21,82	47,58	0,01
BL	88,88	86,47	87,68	18,79	92,93	82,73	87,83	3,353	72,77	70,91	71,84	23,16

Para búsqueda local la desventaja es el tiempo de ejecución, aunque podría acotarse cambiando las condiciones de parada. Para nuestros datasets es asumible. Observamos que se encuentran resultados mucho mejores que con Relief en todos los datasets, en Ionosphere y Parkinsons, la función objetivo se dispara casi al 90 por ciento, consiguiendose además reducir en torno al 80 por ciento de características. En Spectf-Heart conseguimos un resultado similar pero no tan bueno, nuestra tasa de clasificación sube al 72 por ciento y podemos descartar un 70 por ciento de características.

Por ello consideramos que es un buen punto de partida el algoritmo de búsqueda local para compararlo con los modelos bioinspirados de las siguientes prácticas y ver su funcionamiento, converge a un óptimo local al menos para nuestros casos aunque pueda tener periodos de oscilación por como se mutan las componentes, el algoritmo Greedy por su parte es muy rápido y genera una buena primera aproximación, aunque como damos mucha importancia a la simplificación de características no obtiene buenos resultados en estos datasets.

8. Bibliografía

Fuentes de consulta on-line, en general documentación de las librerías:

1. https://www.cplusplus.com/reference/random/default_random_engine/

2. https://en.cppreference.com/w/cpp/chrono/high_resolution_clock