ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

MÁSTER OFICIAL EN VISIÓN ARTIFICIAL

TRABAJO FIN DE MASTER

DETECTOR BADACOST EN C++

Autor: Jorge Vela Peña

Tutor: José Miguel Buenaposada Biencinto

Curso académico 2019/2020

Resumen

Índice general

Índice de figuras

Capítulo 1

Introducción

Como parte de la visión artificial, se considera muy importante la detección de objetos en las imágenes. La detección se basa en encontrar elementos determinados en una imagen partiendo de su apariencia visual. Para conseguir esto, se parte de la búsqueda de patrones determinados que definan el elemento a encontrar. Para realizar esta detección, se emplean algoritmos de clasificación, los cuales realizan transformaciones sobre imágenes, intentando obtener esas características determinadas para localizar en que parte se encuentra un elemento de interés (coches o caras, por ejemplo).

A lo largo de este capítulo se va a realizar una introducción sobre los clasificadores, los distintos tipos de clasificadores que existen. Tras ello hablar de varias formas que hay de trabajar con ellos para terminar hablando de un método específico de trabajo, entrando así en el contexto mas cercano de este trabajo.

1.1. Tipos de clasificadores

Dentro de los clasificadores, podemos encontrar dos grandes divisiones. Por un lado están los clasificadores débiles, los cuales son ligeramente mejores que los clasificadores

aleatorios. Estos clasificadores tienen pocos datos para determinar la clase correcta, siendo adecuados cuando solo existen dos clases. Su error, siendo gamma la ventaja obtenida sobre la aleatoriedad, es $\varepsilon = 0{,}5 - \gamma$.

Por otro lado, un clasificador fuerte es aquel que está bien correlacionado con la clasificación verdadera.

1.1.1. Clasificadores débiles

En esta sección se explican diferentes tipos de clasificadores de esta clase, su funcionamiento y un ejemplo de como trabajan.

K-Nearest Neighbor: Este método busca las observaciones mas cercanas a la que se intenta predecir, y en función de estos datos clasifica el de interés como la mayoría de su alrededor. Para su funcionamiento, memoriza los datos del entrenamiento, los cuales utiliza para realizar la predicción. Por ello, se trata de un método de aprendizaje supervisado ya que necesita un conjunto de entrenamiento previamente etiquetado.

En el siguiente ejemplo, este método trata de clasificar el circulo verde. Para ello, en caso de que tomemos k = 3, siendo k el número de vecinos, se obtendría que el circulo verde es un triangulo. Sin embargo, si tomáramos k = 5, se trataría de un cuadrado azul ya que serían la mayoría de elementos cercanos.

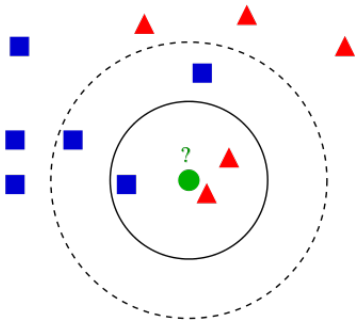


Figura 1.1: Ejemplo gráfico KNN.

Support Vetor Machine: Se trata de un clasificador discriminativo el cual

busca aprender la frontera entre distintas clases. Cuando se trata de n-dimensiones, el clasificador va a buscar el hiperplano que une a las variables de predicción para crear el limite entre categorías. Este hiperplano va intentar tener la máxima distancia con los puntos que estén alrededor de el, obteniendo así un margen entre las clases.

En un clasificador de este tipo, no es posible una separación ideal entre clases, por lo que siempre van a surgir errores en la clasificación. En caso de tener un modelo perfecto que no se pueda generalizar para otro conjunto de datos, se produce lo conocido como sobreajuste.

En la siguiente figura se pueden observar distintos tipos de fronteras entre clases. H1 se trata de una frontera que no hace bien la separación entre clases. H2 es una frontera que hace bien la separación, pero no deja los márgenes óptimos. Sin embargo, H3 realiza una separación correcta entre clases, dejando margenes de distancia con los elementos mas cercanos.

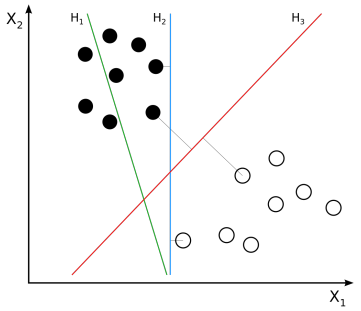


Figura 1.2: Diferentes hiperplanos posibles con SVM.

1.1.2. Clasificadores fuertes

En esta sección se explican distintos clasificadores de este tipo y su funcionamiento.

Naive Bayes: Se trata de un clasificador probabilístico basado en el teorema de Bayes, el cual supone que los atributos de la clase son condicionalmente independientes. Este clasificador trabaja realizando el máximo a posteriori, y pese a que la superposición

anterior no es valida para muchos casos, ya que en muchas ocasiones los atributos de las clases son dependientes entre si, Naive Bayes obtiene resultados muy buenos con un coste computacional reducido. Este tipo de algoritmos son útiles para los casos de multi clases.

Redes de convolución neuronal: Es un algoritmo de aprendizaje profundo. Estas obtienen una imagen de entrada, las cuales procesa para poder identificar objetos, por ejemplo. Este tipo de algoritmos buscan solos las características deseadas, por ello se necesitan muchas imágenes de un mismo elemento, para que la red pueda obtener esas características comunes.

Existen distintas arquitecturas para este tipo de algoritmos. En la mayoría de casos los modelos de retroalimentación ofrecen resultados muy precisos, especialmente en trabajos con imagen. Estas arquitecturas pueden tener distintos niveles de capas. Una red debe tener al menos tres capas, una capa de entrada, encargada de recibir datos , una capa oculta, encargada de procesar la información y obtener los datos de interés, y una capa de salida, encargada de retornar la salida de la red. Se puede dar el caso en el que existan varias capas ocultas, haciendo el sistema más complejo, pero también se necesita mas tiempo para entrenamiento.

En el caso de un sistema complejo con varias capas, las primeras capas son las encargadas de detectar elementos simples (lineas rectas y curvas), para poco a poco especializarse en formas específicas, llegando a reconocer caras y coches, por ejemplo.

1.2. Combinación de clasificadores

En determinadas ocasiones, para realizar un clasificador complejo se hace una combinación de clasificadores. Esto tiene objetivo mejorar un clasificador combinándolo con los demás. Para realizar la combinación de clasificadores existen diferentes razones.

Razón estadística: Uno puede encontrarse con conjuntos de datos pequeños en comparación con el tamaño de la hipótesis. Para ello, el método de clasificación

puede tener múltiples hipótesis que tengan una precisión similar en el conjunto de datos disponible. Si se combinan diferentes hipótesis, al realizar un promedio de los resultados se reduce el riesgo de seleccionar un clasificador con menor precisión.

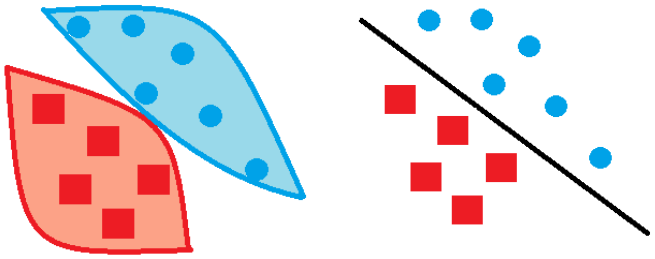
Razón computacional: Se da debido a que el algoritmo no garantiza el mejor clasificador. Los clasificadores realizan una búsqueda local y se pueden quedar atrapados en un óptimo local. Aunque se de el caso en el que el conjunto de datos sea suficiente, es computacionalmente complicado encontrar un sistema de aprendizaje con la mejor hipótesis. Por lo tanto, si se realiza una búsqueda desde diferentes puntos (distintos clasificadores) se obtendrá una mejor aproximación a la hipótesis correcta.

Aproximación de modelos mas complicados: En muchas ocasiones, el clasificador con resultados más óptimos no se podrá obtener con ningún clasificador lineal. Sin embargo, utilizando varios clasificadores sencillos se puede llegar a la mejor aproximación del clasificador correcto.

Se pueden distinguir dos tipos de métodos:

Por un lado, están los métodos discriminativos, los cuales buscan aprender la frontera entre clases. Para ello, buscan la dependencia de la variable objetivo en función de la variable observada. Por otro lado, se encuentran los modelos generativos, los cuales buscan aprender la clase, proporcionando un modelo de como se generan los datos. Por tanto, el modelo discriminativo aprende la distribución de la probabilidad condicional $(X|Y)$, mientras que el modelo generativo aprende la distribución de la probabilidad conjunta (X, Y). A continuación, se dos imágenes de como se representan estos modelos gráficamente:

Tras esto, se procede ha explicar distintos métodos de combinación existentes. Primero se hablara de bagging, explicando su funcionamiento y explicando con algún ejemplo su funcionamiento, para posteriormente hacer lo mismo con el boosting.



(a) Generativos (b) Discriminativos

1.2.1. Bagging

Este método consiste en utilizar diferentes clasificadores en paralelo. Esto se hace ya que hay independencia entre los distintos clasificadores, por lo tanto los resultados que aporten uno no tendrá relación con la clasificación de otro, lo que permitirá reducir el error.

Este método proviene de *agregación de bootstrap*, algoritmo diseñado para mejorar estabilidad y precisión de los algoritmos. Su principal característica es reducir la varianza y evitar un sobreajuste en la clasificación. Para obtener la clasificación final se realiza una votación de los distintos métodos.

Como ejemplo de esta clasificación por votación, sería si tenemos 25 clasificadores iid (independientes e idénticamente distribuidas), cada uno con una tasa de error de 0.35, la probabilidad de que la mitad de clasificadores se equivoquen es de 0.06.

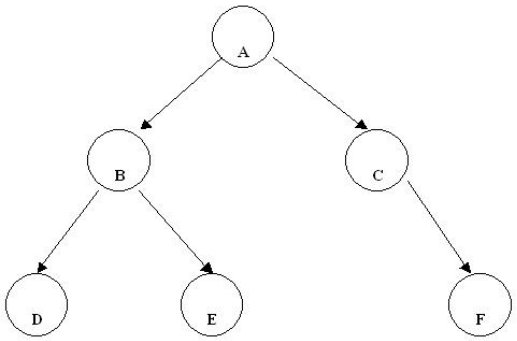
Cabe destacar que será mas sencillos tener clasificadores iid si cada uno de ellos se entrena con parámetros distintos.

1.2.1.1. Random Forest

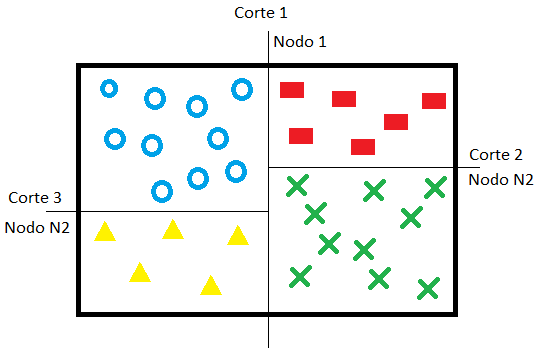
Este método consiste en un *bagging* de árboles binarios. Los árboles binarios son arboles cuyos nodos tendrán como máximo dos nodos descendientes, es decir, arboles de respuesta binaria. Cuando un nodo tiene solo un descendiente, este será el nodo hoja

y se corresponderá a la clase del elemento de entrada. El entrenamiento de este método consiste en dividir el conjunto de datos en dos recursivamente, hasta que queda todo dividido en clases. Lo normal es dividir en cortes paralelos a los ejes, pero en ocasiones se puede obviar esta restricción.

A continuación, en la imagen se observa lo contado de este método, como es gráficamente el árbol binario y la realización del entrenamiento con el conjunto de datos:



(a) Estructura árbol binario.



(b) Entrenamiento para el arbol binario

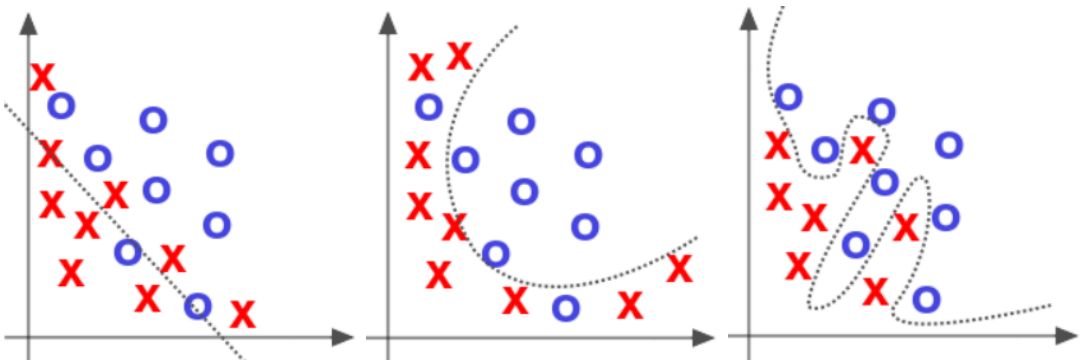
Para saber si un corte es bueno, hay que mirar la entropía. Para ello, se parte de la distribución de datos original y se mira la distribución a cada lado del corte. Si hay mucho desorden (varias clases mezcladas a cada lado) o queda ordenado.

La ventaja de este método es que es rápido y sencillo de realizar.

La desventaja principal, es muy complicado llegar a un árbol que clasifique todo correctamente con los datos del entrenamiento. Si se da este caso, probablemente este ocurriendo *sobreaprendizaje*.

A continuación se observan diferentes imágenes. En la primera de ellas se da el desajuste, que se da cuando no se encuentran los patrones que clasifican correctamente las clases. En la segunda se da el clasificador apropiado para el conjunto de datos dado. Por último, en la tercera imagen se muestra el sobreajuste, es decir, que se clasifica a

partir de los datos del conjunto en lugar de con los patrones que las clases ofrecen.



(a) Desajuste. (b) Clasificador óptimo (c) Sobreajuste del clasificador

Para evitar ese sobreajuste, se utiliza **bagging** de arboles binarios, obteniendo así la técnica **Random Forest**.

Este método consiste en generar diversas muestras (inyectando aleatoriedad), entrenar diferentes arboles y realizar la votación por mayoría para la clasificación de un elemento.

Las características principales de este método son:

- **Paralelismo:** Permite el entrenamiento y ejecución de los distintos arboles en paralelo. Dichos arboles son arboles independientes.
- **Reducción de la varianza:** Como de diferentes serán las predicciones de un modelo si se toman muestras distintas del mismo conjunto de datos.
- **Reducción del sesgo:** Diferencia en promedio existente entre los valores obtenidos y los valores reales.

Otros parámetros relacionados con este método son:

- **Número de árboles:** Utilizando varios arboles se consigue minimizar el error, pero utilizar demasiados arboles puede ser innecesario.

- **Número máximo de niveles:** Cada árbol tendrá un número determinado de niveles. Cuanto más niveles, existirá una mayor precisión, pero evitando llegar al sobreajuste previamente comentado.

1.2.2. Boosting

El *boosting* consiste en la creación de un clasificador fuerte combinando varios clasificadores básicos. Cuando se combinan diferentes clasificaciones, cada uno contribuye según su peso, el cual se asigna en función de la exactitud en sus predicciones.

La idea principal de este método es que ningún clasificador ofrece siempre el mejor resultado que la combinación de todos los clasificadores, ya que esto va a suavizar los errores de cada uno, obteniendo un clasificador mas eficiente.

Como principales características de este método, hay que destacar:

- Los clasificadores se construyen secuencialmente. cada clasificador se construye con información del clasificador previo.
- Se realiza un muestreo selectivo para el conjunto de datos de entrenamiento.
- Es mas probable que se seleccione un elemento mal clasificado para el entrenamiento del siguiente clasificador.

El procedimiento original de boosting se basa en realizar los siguientes pasos:

1. Extraer un subconjunto de la muestra de aprendizaje y entrenar un primer clasificador con el.
2. Extraer otro subconjunto de la muestra de aprendizaje y añadir la mitad de las muestras mal clasificadas anteriormente. Con esto se entrena un segundo clasificador.

3. Encontrar un subconjunto de muestras con los cuales los clasificadores previamente entrenados retornen resultados diferentes y utilizar este subconjunto para entrenar un tercer clasificador.

4. Combinar los tres clasificadores realizando una votación por mayoría.

A continuación, se procede a hablar del algoritmo mas utilizado de boosting, llamado adaboost, y fue el primero de este tipo que consiguió resultados óptimos.

1.2.2.1. AdaBoost

Su nombre proviene de *Adaptative Boosting* y consiste, como se ha explicado previamente, en conseguir un clasificador fuerte a partir de clasificadores débiles. Para ello, este método utiliza el siguiente procedimiento:

1. Se inicializan los pesos para cada observación de la muestra, teniendo inicialmente todas las muestras el mismo peso:

$$w_i = \frac{1}{N}, i = 1, 2, ..., N.$$

2. Para m = 1, 2, ..., M

   a) Entrenar el clasificador básico

   b) Estimar el error $G_m(x)$

   c) Clasificar la importancia del error. Hay que tener en cuenta las observaciones previas:

$$err_m = \frac{\sum_{i=1}^{n} w_i I(y_i \neq G(x_i))}{\sum_{i=1}^{n} w_i}$$

d) Establecer nuevos pesos(si clasifica bien, se asigna un peso bajo, en caso contrario se asignará un peso alto).

$$\alpha_m = \log\left(\frac{1 - err_m}{err_m}\right)$$

e) Actualizar los pesos.

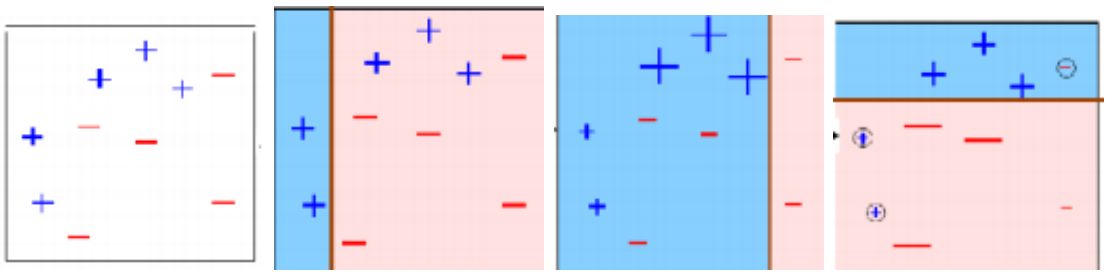
$$w_i = w_i e^{\alpha_m I(y_i \neq G(x_i))}$$

3. Retornar el clasificador final:

$$G(x) = sign\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$$

Por todo esto, suponiendo que $err_m < 0{,}5$, cuando $\alpha_m \geq 0$, en cada iteración aumentan el peso las observaciones mal clasificadas, mientras que lo disminuyen las bien clasificadas.

Las siguientes imágenes corresponden a un ejemplo de este clasificador:

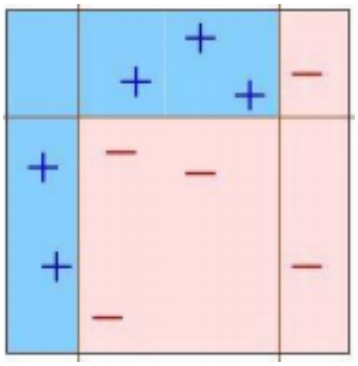


(a) Conjunto de datos. (b) Clasificador 1. (c) Clasificador 2. (d) Clasificador 3.

El primer clasificador intenta separar positivos y negativos, cometiendo 3 errores. El peso de estas muestras incrementa para el siguiente clasificador, mientras que el peso de las otras muestras disminuye. En el siguiente clasificador, las muestras previamente mal clasificadas ahora se clasifican correctamente, pero 3 de los datos negativos ahora se

clasifican mal, por lo que suben su peso. En el último clasificador, los últimos negativos mal clasificados se clasifican de forma correcta debido al aumento de su peso, mientras que otro de los negativos se clasifica de forma incorrecta.

Con todo esto, el clasificador final formado por el conjunto de los tres clasificadores quedaría de la siguiente manera:



(a) Clasificador final.

Obteniendo así un clasificador mejor que los tres vistos previamente por separado.

1.2.2.2. SAMME

Se trata de otro método de boosting, el cual sigue un proceso parecido al clasificador previamente explicado pero con algunas variaciones, como se cuenta a continuación:

1. Los pesos en la inicialización se obtienen de igual forma:

$$w_i = \frac{1}{N}, i = 1, 2, ..., N.$$

2. Para m = 1, 2, ..., M

   a) Se obtiene el clasificador que minimice ϵ_m (error ponderado):

$$\epsilon_m = \sum_{y_i \neq C_m(x_i)} w_i$$

b) Calcular:

$$\alpha_m = 1/2 \log 1 - \epsilon_m \epsilon_m + \ln(K-1)$$

Siendo K el número de clases.

c) Actualizar los pesos:

$$w_i = w_i e^{\alpha_m(x) I(y_i \neq C_m(x_i))}$$

d) Escalar pesos para que sumen 1:

$$w_i = w_i \sum_{j=1}^{n} w_j$$

3. Siendo el clasificador final:

$$C(x) = \arg \max_j \sum_{m=1}^{M} \alpha_m I(C_m(x) = j) Y = 1, ..., K clases.$$

Una vez explicado el contexto mas general sobre el que se basa el trabajo, se continua explicando el entorno mas cercano de este trabajo.

1.3. BAdaCost

El algoritmo sobre el que se trabaja en este trabajo es BAdaCost, el cual se define como un algoritmo de clasificación sensible al costo de varias clases. Para la realización de este algoritmo, se utiliza un conjunto de clasificadores débiles multi-clase sensible a los costes para obtener una regla de clasificación basada en boosting. Para poder explicar este algoritmo, primero hay que hablar sobre CMEL *Cost-sensitive Multi-class Exponential Loss*, algoritmo que optimiza la perdida en varios algoritmos de clasificación como AdaBoost, SAMME, Cost-sensitive AdaBoost o PIBoost. BAdaCost

consigue importantes mejoras de rendimiento en comparación con los algoritmos anteriores.

Este algoritmo busca minimizar la pérdida de CMEL, $\sum_{n=1}^{N} L_c(l_c, f(x_n))$ siendo (x_n, l_n) los datos de entrenamiento. Siendo un algoritmo de Boosting, la minimización se realiza mediante un modelo aditivo por etapas, $f(x) = \sum_{m=1}^{M} \beta_m g_m$. El clasificador seleccionado trata de optimizar β_m en cada iteración a lo largo de g_m. Para calcular esto se siguen los siguientes pasos:

Búsqueda de (β_m,$g_m(x)$) óptimos para CMEL. Siendo C la matriz de costes de un problema multiclase:

$$(\beta_m, g_m(x) = \arg\min_{\beta,g} \sum_{n=1}^{N} exp(C^*(ln, -)(f_m - 1(x_n) + \beta_g(x_n))$$

es la misma solución para:

$$(\beta_m, g_m(x)) = \arg\min_{\beta,g} \sum_{n=1}^{N} S_j exp(\beta C^*(j, j)) + \sum_{j=1}^{K} \sum_{k \neq j} E_j, k exp((\beta C^*(j, j))$$

Donde $S_j = \sum_{n:g(x_n)=l_n=j} w(n)$, $E_{j,k} = \sum_{n:g(x_n)=l_n=j}$ y el peso de la instancia de entrenamiento n es:

$$w(n) = exp(C^*(ln, -) \sum_{t=1}^{m-1} \beta_m f_m(x_n))$$

Dada una dirección conocida de g, el β óptimo se puede obtener como solución de:

$$\sum_{j=1}^{K} \sum_{k \neq j} E_{j,k} C(j, k) A(j, k)^{\beta} = \sum_{j=1}^{K} \sum_{h=1} S_j C(j, h) A(j, j)^{\beta}$$

siendo $A(j, k) = exp(C^*(j, k))j, k \epsilon L$. Por último, dado un β conocido, la dirección descendiente óptima g, equivalente a G, es dada por:

$$\arg\min_{G} \sum_{n=1}^{N} w(n) A(l_n, l_n)^{\beta} I(G(x_n) = l_n) + \sum_{n=1}^{N} w_n \sum_{k \neq l_n} A(l_n, k)^{\beta} I(G(x_n) = K)$$

El código de ejecución de este método se puede explicar como:

1. Se inicializan los pesos para cada observación de la muestra, teniendo inicialmente todas las muestras el mismo peso:

$$w_i = \frac{1}{N}, i = 1, 2, ..., N.$$

2. Se calculan las matrices C^* y A para C.

3. Para m = 1, 2, ..., M

   a) Se obtiene G_m para $\beta = 1$

   b) Se traduce G_m en g_m

   c) Se calculan $E_{j,k}$ y S_j para todo j,k.

   d) Se calcula β_m

   e) $w(n) = w(n)exp(\beta_m C^*(ln, -)g_m(x_n))$

   f) Se normaliza el vector w

4. Se obtiene la salida $H(x) = \arg\min_k C^*(k, -)(\sum_{m=1}^{M} \beta_m g_m(x))$

A continuación, se muestra una tabla de costes promedio y desviación estandar de distintos algoritmos de clasificación, comparando BAdaCost con Ada.C2M1, MultiBoost y Lp-CSB. Se puede observar como en general se obtienen mejores resultados:

	Ada.C2M1	MultiBoost	Lp-CSB	BAdaCost
Cardval	26 (±9)	202 (±36)	30 (±15)	**24** (±15)
Chess	29 (±5)	262 (±34)	**4** (±3)	160 (±9)
Isolet	289 (±48)	140 (±15)	149 (±18)	**66** (±14)
Satimage	478 (±62)	187 (±23)	170 (±26)	**152** (±11)
Segment	212 (±123)	154 (±48)	94 (±18)	**50** (±30)
Shuttle	**2.1** (±0.07)	8.9 (±0.08)	3.3 (±0.13)	3.9 (±0.3)
Coast.Math.	980 (±129)	1056 (±214)	938 (±359)	**920** (±253)
CNAPS	307 (±103)	**171** (±57)	241 (±108)	181 (±51)
OptDigits	366 (±126)	134 (±27)	170 (±34)	**30** (±8)
PenDigits	326 (±29)	160 (±34)	162 (±72)	**18** (±5)
Segmenta	212 (±123)	154 (±48)	94 (±18)	**50** (±30)
Waveform	903 (±113)	515 (±128)	622 (±201)	**367** (±96)

(a) Tabla comparativa.

Tras haber una breve introducción, se va a explicar lo realizado en este proyecto. En el capítulo 2 se habla sobre los objetivos propuestos y la metodología utilizada para llegar a ellos. Tras ello, en el capítulo tres se va a hablar sobre el algoritmo realizado, la implementación del diseño y el código realizado. Después, en el capítulo 4 se van a comentar los experimentos realizados y los resultados obtenidos. Finalmente, en el capítulo 5 se explicaran las conclusiones obtenidas y los próximos trabajos a realizar.

Capítulo 2

Objetivos

Una vez presentado el contexto de este TFM en el primer capítulo, yendo de lo más general (tipos de clasificadores) a lo más cercano (el clasificador a utilizar), en este capítulo se van a explicar los objetivos concretos a conseguir y la metodología utilizada para llegar a ello.

2.1. Objetivo principal

El objetivo principal es hacer funcionar BAdaCost detector, desarrollado para matlab, en C++. Para ello, utilizando la misma imagen de entrada y el mismo detector, hay que obtener la misma salida (mismas detecciones o muy parecidas). Para llegar a esto, se han propuesto distintas fases:

- **Extracción de características:** Para poder clasificar bien los elementos de las imágenes, hay que extraer de forma correcta las características. En este caso los detectores trabajan con diferentes características (espacios de color, magnitud del gradiente y magnitud del histograma). Por ello, se realiza el desarrollo para C++, comparando siempre que los resultados obtenidos sean los mismos que en Matlab.

- **Procesamiento de las características:** Una vez se realiza la extracción de características, pueden darse distintos procesos de transformación como redimensionados, suavizados o relleno (padding) sobre lo obtenido. El siguiente paso era asegurar que estos procesos retornan también el mismo resultado en C++ que en Matlab.

- **Realizar la detección:** Una vez se tienen las características procesadas igual que en Matlab, lo siguiente es realizar la detección. Para ello, hay que guardar el detector de Matlab en un formato legible en C++, leer los parámetros del detector y poder utilizarlos correctamente. Una vez se pueden cargar los parámetros del detector, desarrollar el algoritmo de detección en C++.

2.2. Requisitos

Para la realización de los objetivos anteriormente citados, se tiene que conseguir una solución que cumpla las siguientes características:

- Que se trate de un algoritmo robusto, capaz de trabajar con distintos clasificadores y que no dependa de unos parámetros determinados, sino que sea capaz de obtener resultados óptimos en distintos casos.

- Que el algoritmo sea capaz de trabajar en distintos tipos de máquinas, no necesitando unas características específicas, solo que permita ejecutar C++ y tenga la librería OpenCV instalada.

2.3. Metodología

La metodología propuesta es un desarrollo en espiral. Para ello se proponen semanalmente reuniones con el tutor, en las cuales se presentaba lo conseguido en

el último periodo en relación a los objetivos propuestos y se proponían los nuevos objetivos para la siguiente reunión. Sobre las tareas propuestas se evaluaban los riesgos que podían darse y los avances que se conseguirían en función del camino tomado. Después, se continuaba desarrollando el algoritmo para conseguir los objetivos.Se probaba en todo momento mediante tests si lo desarrollado funcionaba o no. En función del resultado de estos tests, se determinaba si el objetivo se había conseguido o no. Si no se conseguía, analizar el fallo para solventarlo. En caso de que se consiguiese o estar cerca, se proponían nuevos objetivos para continuar con el trabajo.

Figura 2.1: Método de desarrollo en espiral

Durante todo el desarrollo, se utilizado GitHub como repositorio del código, explicando cuales eran los últimos avances subidos en los *commits* y generando un *issue* por cada tarea a realizar, poniendo en él las diferentes dudas que surgían e indicando la resolución final. De esta forma, se conseguía un desarrollo y una comunicación fluida, quedando el código públicamente accesible en https://github.com/RoboticsURJC-students/2017-tfm-jorge$_v ela$.

2.4. Plan de trabajo

La planificación seguida en el desarrollo ha incluido las siguientes fases:

- Investigación: Comprender el funcionamiento del *toolbox* de badacost, ver la organización del código y su funcionamiento, tanto de las funciones por separado como el algoritmo en conjunto, para ver la extracción de características y el funcionamiento del conjunto final.

- Desarrollo del código: Una vez visto el funcionamiento, trabajar en el código para que BAdaCost funcionara en C++ de forma robusta. Para ello, por cada clase o función que se creaba, se realizaba un test de prueba. Con esto, cuando se obtiene el resultado de C++ se compara con el de Matlab, de forma que se asegura un resultado correcto. Los tests también ayudan si a lo largo del desarrollo, se produce algún cambio en el código. Al comprobar continuamente que todo lo anterior funciona, si un cambio provoca fallo es muy fácil de detectar y saber porque ocurre.

- Compatibilidad entre Matlab y C++: Bien sea para comparar los datos, o para utilizar el mismo detector, tiene que haber un método que permita tener datos iguales en ambos lenguajes. Para ello se ha utilizado el formato YML. Con esto, se pueden guardar todo tipo de datos desde matlab (como el resultado de funciones determinadas o los parámetros de un detector) y leerlo desde C++. En Matlab se ha creado una función que guarda lo deseado con un formato determinado, el cual se lee desde el código desarrollado en C++.

Capítulo 3

Algoritmo BAdaCost en C++

En este capítulo se describe el algoritmo a través del cual se ha llegado a la solución de lo planteado en los objetivos.

Este algoritmo tiene que permitir la detección de distintos elementos utilizando el clasificador BAdaCost. Para ello, en este capítulo primero se explica el diseño y organización del código. Tras ello, se explica como se ha desarrollado cada clase y función y los resultados obtenidos por cada una. Finalmente explicar el detector obtenido, el cual utiliza todo lo desarrollado previamente, retornando si en la imagen de entrada se encuentra el elemento de interés y donde está situado.

3.1. Estructura del código.

Para la realización de este algoritmo, se ha seguido un a estructura que, siguiendo una estructura de lo mas general a lo mas específico, se puede resumir de la siguiente forma:

- **BadacostDetector:** Clase del detector que se utiliza en este proyecto. Para su funcionamiento, contiene una función la cual requiere como parámetro una

imagen de entrada y un fichero que contenga los parámetros del clasificador. Como salida retorna un vector con las distintas detecciones en un formato especifico.

- **ChannelsPyramid:** Encargada de calcular la pirámide de características a partir de la imagen de entrada. Para ello, calcula los distintos tamaños que tiene que tener la imagen en función de los parámetros de entrada y llama al calculo de estas características para los distintos tamaños.

- **ChannelsExtractorLDCF:** A partir de una imagen de entrada, extrae las características de la imagen, obteniendo imágenes (cv::Mat) como resultados, y realiza una convolución con filtros cargados en el detector.

- **ChannelsExtarctorACF:** Calcula las características de la imagen de entrada (canales de color en LUV, magnitud del gradiente y características HOG) retornando cada característica en un cv::Mat.

- **ChannelsExtractorGradMag:** Calcula la magnitud del gradiente y la orientación por cada píxel de la imagen.

- **ChannelsExtractorGradHist:** Calcula el histograma de gradiente orientado. Para cada región de tamaño determinado, calcula un histograma de gradientes, con cada gradiente cuantificado por su ángulo y ponderado por su magnitud.

- **ChannelsExtractorGradLUV:** Dada una imagen de entrada RGB retorna un vector con los canales de color L, U, V.

- **Utils:** Contiene diferentes funciones que utilizan las clases previamente explicadas.

Una vez hecha una introducción al algoritmo, habiendo definido cada una de las partes, se pasa a redactar cada una de las clases y funciones desarrolladas:

Capítulo 4

Experimentos

Capítulo 5

Conclusiones

5.1. Líneas futuras