PEC 2 - Machine Learning

Predicción de interacción entre péptido y el complejo mayor de histocompatibilidad tipo I con Artificial Neural Networks (ANN) y Support Vector Machines (SVM)

María Plaza García

18 de Mai, 2020

Índice

1. Introducción

En esta PEC vamos a resolver un análisis relacionado con la modelización de la interacción entre el complejo mayor de histocompatibilidad tipo I (MHCI, en inglés) y péptidos. Para ello implementaremos dos modelos, un algoritmo de red neuronal artificial (ANN) y un algoritmo Support Vector Machine (SVM).

En la inmunoterapia contra el cancer las células T deben activarse al exponerse a péptidos tumorales unidos a MHCI (pMHCI). Al analizar la genética del tumor, se pueden identificar péptidos relevantes y, dependiendo del tipo particular de MHCI que tiene el paciente, podemos predecir qué interacción péptido MHCI (pMHCI) es probable que esté presente en el tumor del paciente y, por lo tanto, qué pMHCI se deben usar para activar las células T.

Los ficheros necesarios para realizar la PEC estan en formato csv con separador punto y coma. Se encuentran dentro de mi repositorio github (Github 2016), asi como cada uno de los archivos creados para la realización de esta PEC:

https://github.com/mariaplaza/PEC2_Machine_Learning

En cada registro del fichero peptidos.csv se tienen dos variables: 1) el péptido, 2) la clase de interacción donde NB significa no interacción y SB significa interacción positiva.

En el caso que nos ocupa, análisis basados en secuencias, se usará la transformación denominada one-hot encoding. El one-hot encoding representa cada aminoácido por un vector de 20 componentes, con 19 de ellas a 0 y una a 1 indicando el aminoacido. Pongamos por ejemplo, el aminoácido A se representa por (1,0,. . . ,0) y el aminoácido R por (0,1,0, . . . , 0). Por tanto, para una secuencia de 9 aminoácidos, como en nuestro caso, se obtendrá un vector de 20*9 componentes, resultado de concatenar los vectores para cada uno de los 9 aminoácidos.

Una vez realizada la transformación, one-hot encoding el objetivo es implementar un algoritmo de red neuronal artificial y de support vector machine para predecir si la secuencia peptídica interacciona o no con MHCI.

2. Algoritmo Red Neuronal Artificial

Una red neuronal, según Freeman and Skapura (1993), es un sistema de procesadores paralelos conectados entre sí en forma de grafo dirigido. Esquemáticamente cada elemento de procesamiento (neuronas) de la red se representa como un nodo, que recibe y envia señales (información). Se crea entonces una red con diferentes capas interconectadas para procesar la información. Cada capa esta formada por un grupo de nodos que transmite la información a los otros nodos de las capas siguientes. Estas conexiones establecen una estructura jerárquica que tratando de emular la fisiología del cerebro busca nuevos modelos de procesamiento para solucionar problemas concretos del mundo real. Lo importante en el desarrollo de la técnica de las ANN es su útil comportamiento al aprender, reconocer y aplicar relaciones entre objetos y tramas de objetos propios del mundo real. Es por tanto, una nueva forma de computación o modelo matemático inspirado en modelos biológicos y compuesto por un gran número de elementos procesales organizados en niveles.

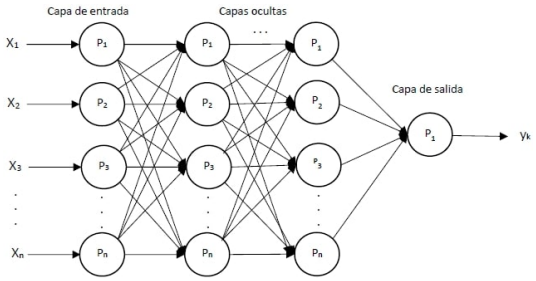


Figura 1: Modelo general de red de neurona ariticial. Fuente: Sanchez(2015)

Debido a que presentan un gran número de características similares a las del cerebro humano, las redes neuronales son capaces de aprender de la experiencia, de abstraer características esenciales a partir de entradas que presentan información irrelevante, de generalizar de casos anteriores a nuevos casos…etc. Todo esto permite su aplicación en un gran número de areas muy diferenciadas (Lantz 2015):

- Aproximación de funciones, o el análisis de regresión, incluyendo la predicción de series temporales, funciones de aptitud y modelado.

- Clasificación, incluyendo el reconocimiento de patrones y la secuencia de reconocimiento, detección y la toma de decisiones secuenciales (diagnostico médico, aplicaciones financieras, sistemas radar, reconocimiento facial, clasificación de señales, control del vehículo, predicción de trayectorias, el control de procesos, manejo de recursos naturales, juegos y la toma de decisiones como el backgammon, ajedrez, póquer…)
- Procesamiento de datos, incluyendo el filtrado, el agrupamiento, la separación ciega de las señales y compresión (diferenciando, por ejemplo, entre informes deseados y no deseados en redes sociales y prevención de spam)
- Robótica, incluyendo la dirección de manipuladores y prótesis.
- Ingeniería de control, incluyendo control numérico por computadora.

Las fortalezas y debilidades de este algoritmo son:

Fortalezas	**Debilidades**
- Adaptable a clasificación o problemas de predicción numérica	- Requiere de gran potencia computacional
- Capaz de modelar patrones más complejos que casi cualquier otro algoritmo	- Propenso a sobreajustar los datos de entrenamiento
- No necesita muchas restricciones acerca de las relaciones subyacentes de los datos	- Es un modelo de caja negra complejo que es difícil, si no imposible, de interpretar
- Aprendizaje Adaptativo: Capacidad de aprender a realizar tareas basadas en un entrenamiento o en una experiencia inicial	- Complejidad de aprendizaje para grandes tareas, cuanto más cosas se necesiten que aprenda una red, mas complicado será enseñarle
- Auto-organización: Una red neuronal puede crear su propia organización o representación de la información que recibe mediante una etapa de aprendizaje	- Tiempo de aprendizaje elevado. Esto depende de dos factores: primero si se incrementa la cantidad de patrones a identificar o clasificar y segundo si se requiere mayor flexibilidad o capacidad de adaptación de la red neuronal para reconocer patrones que sean sumamente parecidos
-Tolerancia a fallos: La destrucción parcial de una red conduce a una degradación de su estructura; sin embargo, algunas capacidades de la red se pueden retener, incluso sufriendo un gran daño	- Elevada cantidad de datos para el entrenamiento, cuanto mas flexible se requiera que sea la red neuronal, mas información tendrá que enseñarle para que realice de forma adecuada la identificación
-Operación en tiempo real: Los cómputos neuronales pueden ser realizados en paralelo; para esto se diseñan y fabrican máquinas con hardware especial para obtener esta capacidad	- Falta de reglas definitorias que ayuden a realizar una red para un problema dado
-Fácil inserción dentro de la tecnología existente: Se pueden obtener chips especializados para redes neuronales que mejoran su capacidad en ciertas tareas. Ello facilitará la integración modular en los sistemas existentes	- Requieren la definición de muchos parámetros antes de poder aplicar la metodología

3. Algoritmo Support Vector Machine

El método de clasificación-regresión denominado Máquinas de Vector Soporte (Vector Support Machines, SVMs) es un algoritmo de aprendizaje supervisado que fue desarrollado en la década de los 90 dentro de campo de la ciencia computacional. Si bien originariamente se desarrolló como un método de clasificación binaria, su aplicación se ha extendido a problemas de clasificación múltiple y regresión. SVMs ha resultado ser uno de los mejores clasificadores para un amplio abanico de situaciones, por lo que se considera uno de los referentes dentro del ámbito de aprendizaje estadístico y machine learning (Suárez 2014).

Las máquinas de vectores de soporte pertenecen a una clase de algoritmos de aprendizaje automático llamados métodos de kernel y también se conocen como máquinas de kernel. Se fundamentan en el Maximal Margin Classifier, que a su vez, se basa en el concepto de hiperplano óptimo como una superficie de decisión de tal manera que se maximiza el margen de separación entre las dos clases en los datos (Lantz 2015). Los vectores de soporte se refieren a un pequeño subconjunto de las observaciones de entrenamiento que se utilizan como soporte para la ubicación óptima de la superficie de decisión.

Cuando los datos no son separables de forma lineal, es necesario el uso de kernels, o funciones de similitud y especificar un parámetro determinado para minimizar la función de coste. La elección de este parámetro es en base ensayo/error, pero se buscan valores que no sean extremos en la búsqueda del equilibrio sesgo/varianza. Los kernels más populares son el lineal y el gausiano, aunque hay otros como el polinomial, string kernel o chi-square kernel.

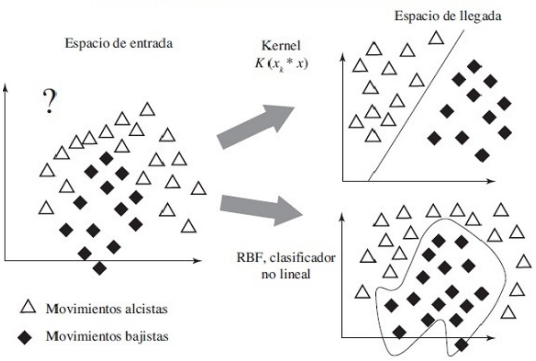


Figura 2: Modelo general de Vector Support Machines. Fuente: Sanchez(2015)

En R, las librerías `e1071` y `LiblineaR` contienen los algoritmos necesarios para obtener modelos de clasificación simple, múltiple y regresión, basados en Support Vector Machines.

Algunos casos de éxito de las máquinas de vectores de soporte son:

- reconocimiento óptico de caracteres
- detección de caras para que las cámaras digitales enfoquen correctamente
- filtros de spam para correo electrónico
- clasificar genes diferencialmente expresados a partir de datos de microarrays
- reconocimiento de imágenes a bordo de satélites (saber qué partes de una imagen tienen nubes, tierra, agua, hielo, etc.)
- procesamiento de lenguaje natural, reconocimiento de voz e imagen y visión por computadora
- clasificación de texto en distintas categorías temáticas
- detección de eventos críticos de escasa frecuencia, como terremotos

La capacitación para una SVM tiene dos fases:

1. Transformar los predictores (datos de entrada) a un espacio de características de alta dimensión.
2. Resuelver un problema de optimización cuadrática para ajustar un hiperplano óptimo y clasificar las características transformadas en dos clases. El número de características transformadas está determinado por el número de vectores de soporte. Solo los vectores de soporte elegidos de los datos de entrenamiento son necesarios para construir la superficie de decisión. Una vez entrenado, el resto de los datos de entrenamiento son irrelevantes.

Las fortalezas y debilidades de este algoritmo son:

Fortalezas	**Debilidades**
- Uso en predicción y clasificación. Uso bastante extendido	- Requiere especificar parámetro C y función de kernel (prueba y error)
- Funciona de forma óptima con ruido	- Lento de entrenar, sobre todo a medida que aumenta el número de características
- Facilidad de uso en comparación de las redes neuronales	-Al igual que las redes neuronales es difícil de interpretar el funcionamiento interno.
- Eficaz en espacios de grandes dimensiones	- No proporcionan directamente estimaciones de probabilidad, éstas se calculan utilizando una validación cruzada quíntuple.
- Todavía eficaz en casos donde el número de dimensiones es mayor que el número de muestras	- Complejidad temporal del algoritmo

Fortalezas	Debilidades
- Utiliza un subconjunto de puntos de entrenamiento en la función de decisión (llamada vectores de soporte), por lo que también es eficiente en memoria - Versátil: se pueden especificar diferentes funciones del núcleo para la función de decisión.	

4. Trabajando con los datos

4.1. Descripción y análisis preliminar de los datos

Para facilitar la reproducibilidad del informe, se han incluido varios parámetros en el encabezado `YAML` del documento cuyos valores se pueden establecer cuando se procesa el informe. Se ha incluido tanto la semilla que emplearemos más tarde en la creación de los datos de test y de entrenamiento así como los nombres de los archivos y la ruta de acceso, de esta forma podemos leer los datos con el siguiente código:

```
# Ahora ya se importan los datos a formato data.frame
library(readr)

peptidos <- read_delim(file=file.path(params$folder.data,params$file1),
                       ";", escape_double = FALSE, trim_ws = TRUE )
length(complete.cases(peptidos)) # comprobamos que el dataset está completo
```

```
[1] 15840
```

Si no disponemos de esta posibilidad, se puede crear un chunk donde se realice la asignación de las variables

```
# peptidos <- read_delim("dataset/peptidos.csv", ";",
# escape_double = FALSE, trim_ws = TRUE)
# length(complete.cases(peptidos))
```

Empezamos echando un breve vistazo a los datos descargados y así tener una idea clara las características de los datos con los que estamos trabajando.

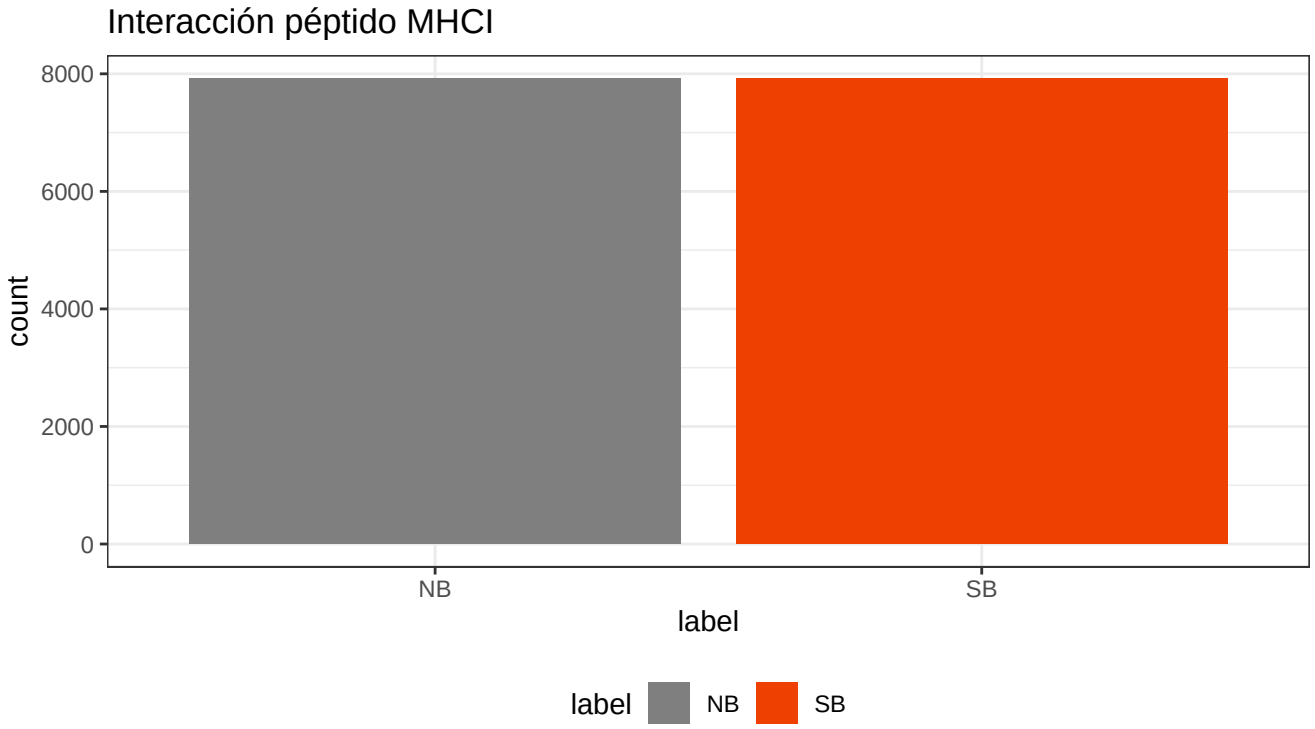
```
str(peptidos)
```

```
tibble [15,840 x 2] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ sequence: chr [1:15840] "LMAFYLYEV" "HQRLAPTMP" "FMNGHTHIA" "WLLIFHHCP" ...
 $ label   : chr [1:15840] "SB" "NB" "SB" "NB" ...
 - attr(*, "spec")=
  .. cols(
  ..   sequence = col_character(),
  ..   label = col_character()
  .. )
```

```
head(peptidos)
```

sequence	label
LMAFYLYEV	SB
HQRLAPTMP	NB
FMNGHTHIA	SB
WLLIFHHCP	NB
MRYRVSVHP	NB
VLNGYSWFA	SB

```
ggplot(data = peptidos, aes(x = label, y = ..count.., fill = label)) +
  geom_bar() +
  scale_fill_manual(values = c("gray50", "orangered2")) +
  labs(title = "Interacción péptido MHCI") +
  theme_bw() +
  theme(legend.position = "bottom")
```




```
# Tabla de frecuencias
table(peptidos$label)
```

```
  NB   SB
7920 7920
```

Nuestro dataframe está compuesto por 15840 observaciones de 2 variables de caracteres, representando la primera de ellas la secuencia del peptido y la segunda el factor que nos indica si la secuencia peptídica interacciona o no con MHCI. Como vemos, tenemos el mismo número de sucencias que interaccionan tanto como secuencias sin interacción.

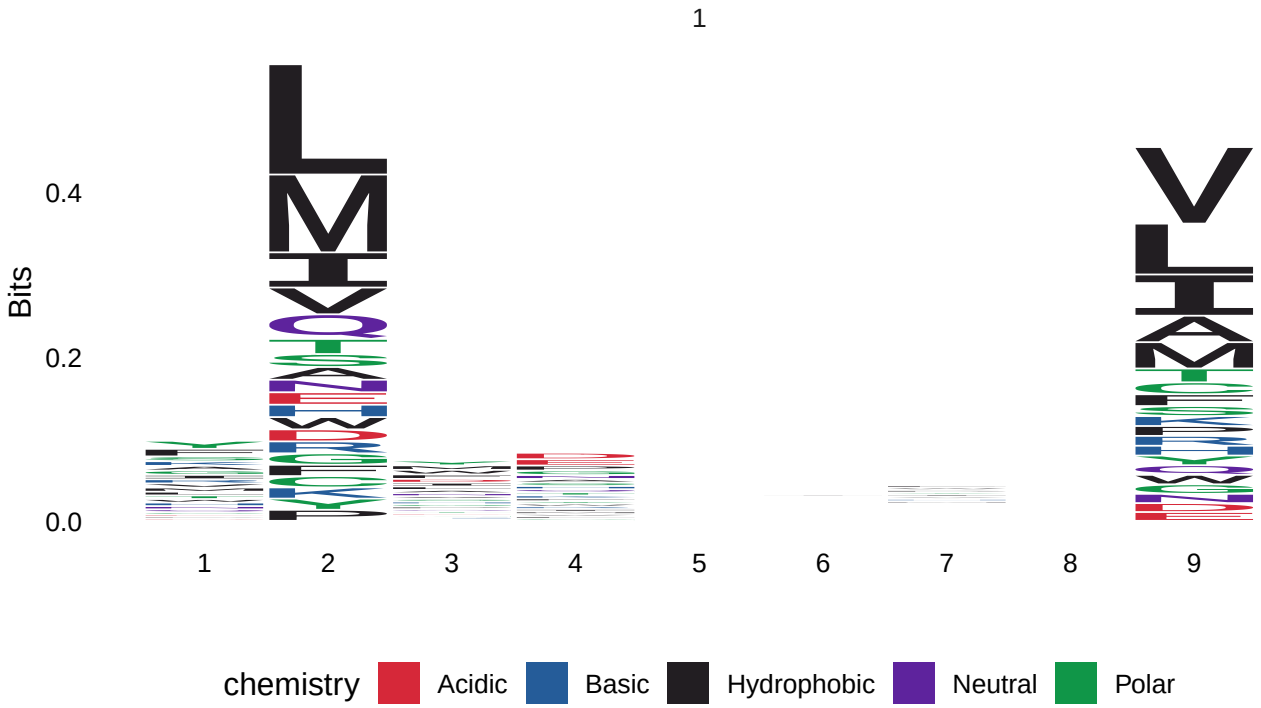
Para representar estos datos, podemos emplear su secuencia logo (https://en.wikipedia.org/wiki/Sequence_logo) con la ayuda del paquete `ggseqlogo`(Wagih 2017). Un logotipo de secuencia consiste en una pila de letras en cada posición. Los tamaños relativos de las letras indican su frecuencia en las secuencias. La altura total de las letras representa el contenido de la información de la posición, en bits o frecuencia.

Los logotipos de secuencia se han convertido en un método de visualización crucial para estudiar los patrones de secuencia subyacentes en el genoma. ggseqlogo es un paquete R construido en el paquete ggplot2 que permite realizar ilustraciones de secuencias de ADN, ARN y proteínas listos para publicación de una manera altamente personalizable con características que incluyen gráficos de múltiples logotipos, esquemas de color cualitativos y cuantitativos, anotaciones de logotipos e integración con otros gráficos (Wagih 2017).

Podemos dibujar una secuencia de logos de nuestros datos usando la función *ggplot* y con *geom_logo*:

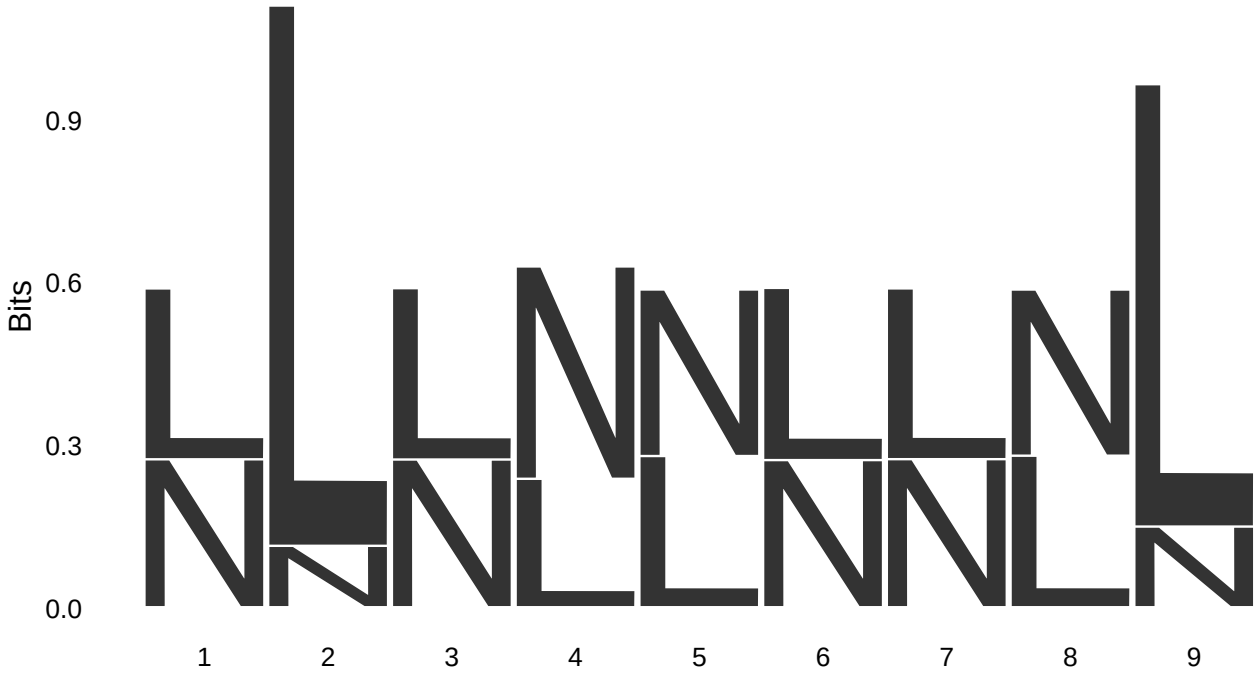
```
library(ggplot2)
library(ggseqlogo)
```

```
ggplot() + geom_logo(peptidos$sequence) + theme_logo() +
facet_wrap(~seq_group, ncol=4, scales='free_x')
```

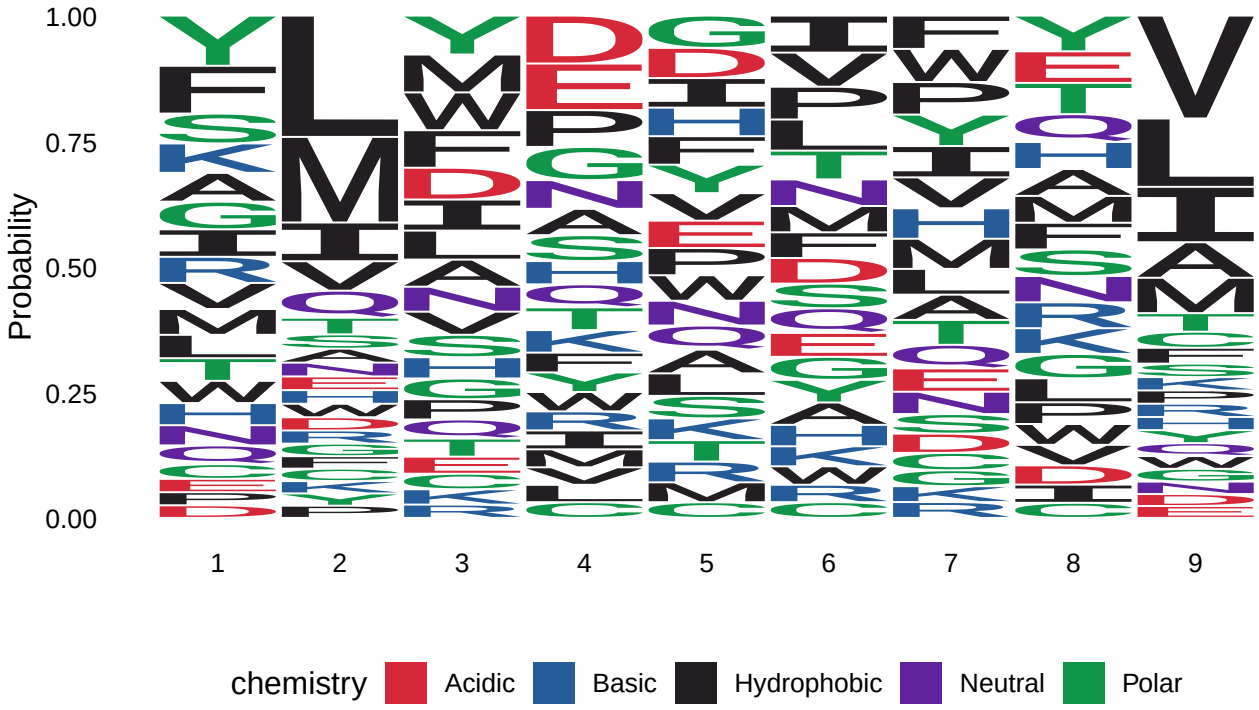


También podemos usar el paquete `ggseqlogo` como acceso directo para hacer lo mismo. Esta es una función que agrega *theme_logo* de forma predeterminada y realiza cualquier facetado requerido si se deben dibujar logotipos de secuencias múltiples. `ggseqlogo` admite dos métodos de logotipo de secuencia: 'bits' y 'probabilidad':

```
library(ggseqlogo)
pepseq <- as.vector(peptidos$sequence)

ggseqlogo(pepseq, seq_type = "auto", namespace = "NULL", font = "helvetica_regular", col_scheme = "nucleot
```




```r
ggseqlogo(pepseq, method = "prob")
```



4.2. Codificación “one-hot” de las secuencias

En este paso vamos a desarrollar una función en R que implemente la codificación “one-hot” (one-hot encoding) de las secuencias.

```
convert <- function(l) {
 code1 <- c("10000000000000000000", "01000000000000000000",
            "00100000000000000000", "00010000000000000000",
            "00001000000000000000", "00000100000000000000",
            "00000010000000000000", "00000001000000000000",
            "00000000100000000000", "00000000010000000000",
            "00000000001000000000", "00000000000100000000",
            "00000000000010000000", "00000000000001000000",
            "00000000000000100000", "00000000000000010000",
            "00000000000000001000", "00000000000000000100",
            "00000000000000000010", "00000000000000000001")
 names(code1) <- c("A", "R", "N", "D", "C", "Q", "E", "G", "H",
                   "I", "L", "K", "M", "F", "P", "S", "T", "W", "Y", "V")
     sapply(strsplit(l, "", perl = TRUE),
     function(x) paste(code1[x], collapse = ""))
}
```

4.3. Secuencias de aminoácidos en vectores numéricos

Empleando la función creada en el apartado anterior, transformamos las secuencias de aminoácidos en vectores numéricos y estos en un data frame que podamos emplear en los siguientes pasos, al que denominamos *data.pep*.

```
data <- convert(peptidos$sequence) # aplicamos la función creada
                                   # en el apartado anterior a
                                   # nuestra variable "sequence"
v1 <- seq(1, 180, by = 1) # le damos un nombre a cada una de las 180 variables

# Lo convertimos en data frame
data.pep <- as.data.frame(t(as.data.frame(strsplit(data, "",
              perl = TRUE))), col.names = v1, row.names = FALSE)
indx <- sapply(data.pep, is.character)
data.pep[indx] <- lapply(data.pep[indx], function(x) as.numeric(as.character(x)))
```

Creamos y guardamos un archivo csv para poder utilizarlo posteriormente y lo denominamos *peptidos_transf_one_hot1.csv*:

```
#write.csv(data.pep, "./dataset/peptidos_transf_one_hot1.csv",
#          sep = ",", col.names = FALSE, row.names = FALSE)
```

4.4. Algoritmo Red Neuronal Artificial

4.4.1. Paso 1. Leer los datos transformados

Podemos emplear directamente el archivo creado en los pasos anteriores *data.pep* o cargar los datos desde la carpeta donde los hemos creado. Seguiremos este segundo paso.

```
library(readr)

data.nn <- read_delim(file=file.path(params$folder.data,params$file2),
                      ",", trim_ws = TRUE )
length(complete.cases(data.nn)) # comprobamos que el dataset está completo
```

```
[1] 15840
```

```
# Para  tener un primer contacto con los datos, se presenta
# los seis primeros registros y las ultimas 5 variables,
# así como las dimensiones de la tabla que hemos importado:

dim(data.nn)
```

```
[1] 15840   180
```

```
head(data.nn[,(ncol(data.nn)-4):ncol(data.nn)])
```

V176	V177	V178	V179	V180
0	0	0	0	1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

```
# Podemos comprobar si todas las observaciones están completas y no falta ningún valor.
# colSums(is.na(dataRNA))
```

4.4.2. Paso 2. Partición de los datos en training/test

En primer lugar, creamos las variables binarias en lugar de usar la variable factor en nuestra base de datos transformada.

```
n <- nrow(data.nn)

# Creación de variables binarias en lugar de usar la variable factor
data.nn$SB <- peptidos$label=="SB"
data.nn$NB <- peptidos$label=="NB"
```

Se realiza una extracción de los datos *aleatoriamente* de 66.67 % de todas las observaciones, $1{,}056 \times 10^4$, para entrenar al modelo y del resto 5280 para evaluarlo (test).

```
# creamos las observaciones para entrenamiento y test.
set.seed(params$seed.train)

train <- sample(n,floor(n*params$p.train))
data.nn.train <- data.nn[train,]
data.nn.test  <- data.nn[-train,]

dim(data.nn.train)
```

```
[1] 10560   182
```

```
dim(data.nn.test)
```

```
[1] 5280  182
```

4.4.3. Paso 3. Modelos de ANN de capa oculta con 1 y 3 nodos

Para la construcción de la red neuronal artificial se usa la función `neuralnet()` del paquete *neuralnet*:

La fórmula del modelo tiene 2 (número de columnas de `data.nn`) nodos de entrada y 2 (niveles de `peptidos$label`) nodos de salida:

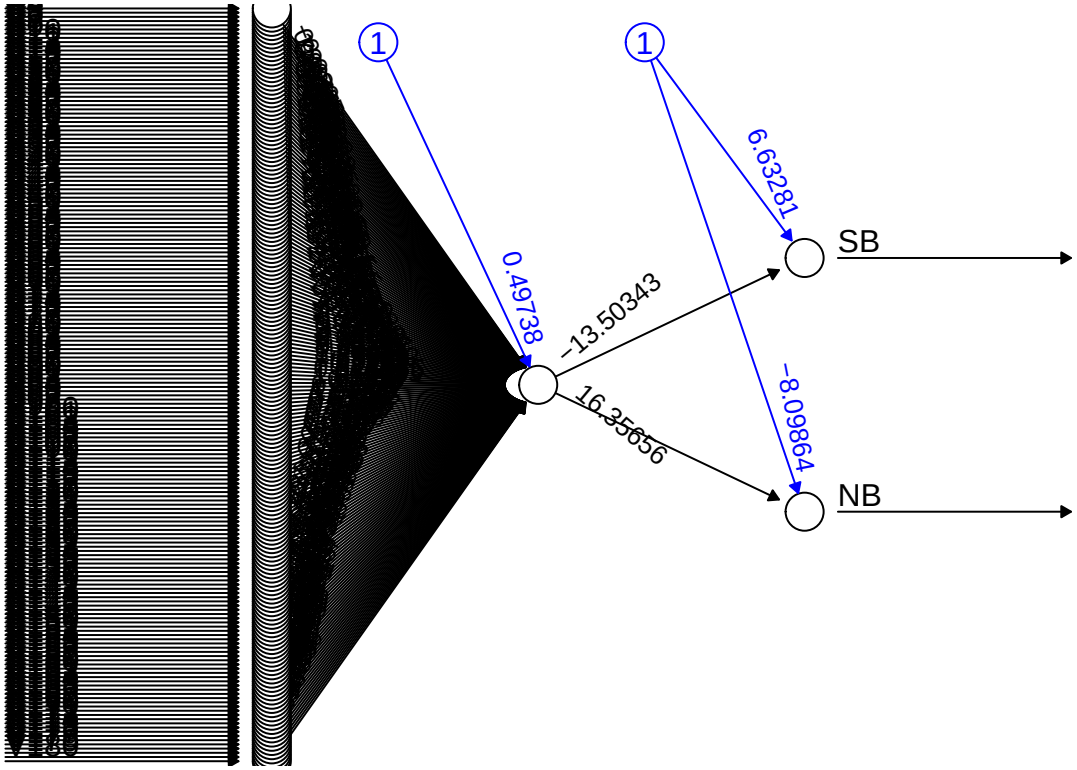
Con un nodo

```
SB + NB ~ V1 + V2 + V3 + V4 + V5 + V6 + V7 + V8 + V9 + V10 +
    V11 + V12 + V13 + V14 + V15 + V16 + V17 + V18 + V19 + V20 +
    V21 + V22 + V23 + V24 + V25 + V26 + V27 + V28 + V29 + V30 +
    V31 + V32 + V33 + V34 + V35 + V36 + V37 + V38 + V39 + V40 +
    V41 + V42 + V43 + V44 + V45 + V46 + V47 + V48 + V49 + V50 +
    V51 + V52 + V53 + V54 + V55 + V56 + V57 + V58 + V59 + V60 +
    V61 + V62 + V63 + V64 + V65 + V66 + V67 + V68 + V69 + V70 +
```

```
    V71 + V72 + V73 + V74 + V75 + V76 + V77 + V78 + V79 + V80 +
    V81 + V82 + V83 + V84 + V85 + V86 + V87 + V88 + V89 + V90 +
    V91 + V92 + V93 + V94 + V95 + V96 + V97 + V98 + V99 + V100 +
    V101 + V102 + V103 + V104 + V105 + V106 + V107 + V108 + V109 +
    V110 + V111 + V112 + V113 + V114 + V115 + V116 + V117 + V118 +
    V119 + V120 + V121 + V122 + V123 + V124 + V125 + V126 + V127 +
    V128 + V129 + V130 + V131 + V132 + V133 + V134 + V135 + V136 +
    V137 + V138 + V139 + V140 + V141 + V142 + V143 + V144 + V145 +
    V146 + V147 + V148 + V149 + V150 + V151 + V152 + V153 + V154 +
    V155 + V156 + V157 + V158 + V159 + V160 + V161 + V162 + V163 +
    V164 + V165 + V166 + V167 + V168 + V169 + V170 + V171 + V172 +
    V173 + V174 + V175 + V176 + V177 + V178 + V179 + V180
```

El modelo aplicado es de un nodo en la capa oculta, esto se consigue con el argumento `hidden=1`. El modelo se construye con el argumento `linear.output=FALSE` ya que se trata de un problema de clasificación.

```
# simple ANN con una única neurona en la capa oculta
set.seed(params$seed.clsfier)  # semilla que nos garantiza resultados reproducibles
nn.model.1 <- neuralnet(fmla, data = data.nn.train, hidden = 1, linear.output = FALSE)

# Visualizamos la topología de la red neuronal:
plot(nn.model.1, rep = "best")
```



Podríamos también representar el mismo modelo usando el paquete *NeuralNetTools* con el siguiente código.

```
library(NeuralNetTools)
# plotnet(nn.model.1, prune_col = 'lightblue', alpha=0.6)
```

El resultado de la matriz de confusión con los datos de test en este modelo podemos obtenerla:

```
nn.model.1.matrix <- predict(nn.model.1, data.nn.test[, 1:180])

# Creamos una salida binaria múltiple a nuestra salida categórica
max.idx <- function(xxy) {
```

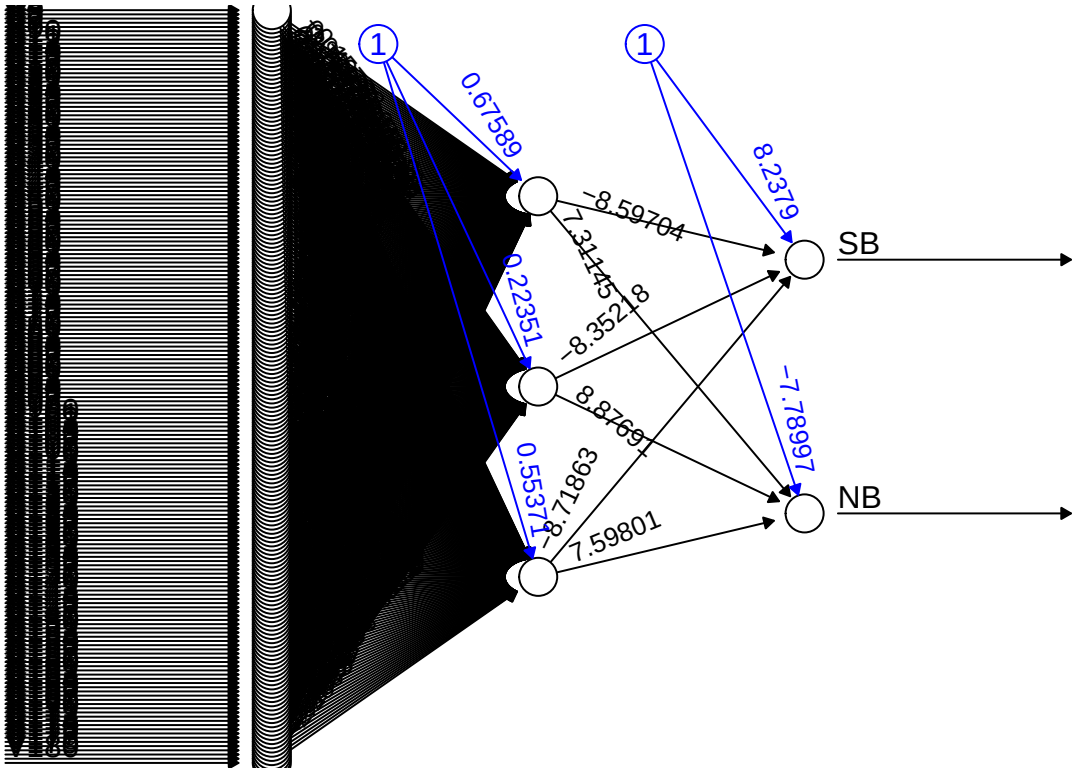
```
    return(which(xxy == max(xxy)))
}

idx1 <- apply(nn.model.1.matrix, 1, max.idx)
prediction1 <- c("SB", "NB")[idx1]
res1 <- table(prediction1, peptidos$label[-train])

# Resultados require(caret)
library(e1071)
(cmatrix1 <- confusionMatrix(res1, positive = "SB"))
```

```
Confusion Matrix and Statistics


prediction1   NB   SB
         NB 2654    1
         SB    5 2620

               Accuracy : 0.9989
                 95% CI : (0.9975, 0.9996)
    No Information Rate : 0.5036
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.9977

 Mcnemar's Test P-Value : 0.2207

            Sensitivity : 0.9996
            Specificity : 0.9981
         Pos Pred Value : 0.9981
         Neg Pred Value : 0.9996
             Prevalence : 0.4964
         Detection Rate : 0.4962
   Detection Prevalence : 0.4972
      Balanced Accuracy : 0.9989

       'Positive' Class : SB
```

Con tres nodos

El primer modelo fue con *un nodo* en la capa oculta. Ahora se plantea *3 nodos* en la capa oculta para tratar de mejorar el rendimiento.

```
# simple ANN con una tres neuronas en la capa oculta
set.seed(params$seed.clsfier)
nn.model.3 <- neuralnet(fmla, data = data.nn.train, hidden = 3, linear.output = FALSE)

# Visualizamos la topología de la red neuronal:
plot(nn.model.3, rep = "best")
```



El resultado de la matriz de confusión con los datos de test:

```
nn.model.3.matrix <- predict(nn.model.3, data.nn.test[, 1:180])

idx <- apply(nn.model.3.matrix, 1, max.idx)
prediction <- c("SB", "NB")[idx]
res <- table(prediction, peptidos$label[-train])

# Resultados require(caret)
library(e1071)
(cmatrix3 <- confusionMatrix(res, positive = "SB"))
```

```
Confusion Matrix and Statistics


prediction   NB   SB
        NB 2650    2
        SB    9 2619

               Accuracy : 0.9979
                 95% CI : (0.9963, 0.999)
    No Information Rate : 0.5036
    P-Value [Acc > NIR] : < 2e-16

                  Kappa : 0.9958

 Mcnemar's Test P-Value : 0.07044

            Sensitivity : 0.9992
            Specificity : 0.9966
         Pos Pred Value : 0.9966
         Neg Pred Value : 0.9992
             Prevalence : 0.4964
```

```
         Detection Rate : 0.4960
   Detection Prevalence : 0.4977
      Balanced Accuracy : 0.9979

       'Positive' Class : SB
```

4.4.4. Paso 4. Comentar los resultados

El nuevo modelo con 3 nodos ocultos obtiene una precisión de 0.998 y una sensitividad y especificidad de 0.999 y 0.997 respectivamente. Vemos que el modelo obtenido con un solo nodo tiene una mayor precisión

El modelo con mayor precisión es el modelo más sencillo. Los modelos más complejos también son más susceptibles de tener overfitting.

4.4.5. Paso 5. Modelo mlp con el paquete caret

El paquete `caret` (abreviatura de Classification And REgression Training) contiene funciones para simplificar el proceso de entrenamiento de modelos para problemas complejos de regresión y clasificación. El paquete utiliza una cantidad de paquetes R (Kuhn 2012) pero intenta no cargarlos todos al inicio (al eliminar las dependencias formales del paquete, el tiempo de inicio del mismo puede disminuir considerablemente).

La función `nnet` admite datos de tipo factor, así que no es necesario transformar la variable *label* en variables binarias. Para el nuevo data.frame con la variable *label* incluida, creamos de nuevo los grupos de training (2/3) y test (1/3) con la funcion `createDataPartition`.

```
#Partición de datos
set.seed(params$seed.train)
# El 75% de los datos lo usamos para el entrenamiento
data.Train <- createDataPartition(y=peptidos$label,
                        p=0.66666666666666667, list=FALSE)
dim(data.Train)
```

```
[1] 10560     1
```

```
# Creamos el dataset codificado con la variable label como variable 181
data.label <- cbind(data.nn[,1:180],Class=peptidos[,2])

train.set <- data.label[data.Train,]
test.set  <- data.label[-data.Train,]

nrow(train.set)/nrow(test.set) # debe estar alrededor de 2
```

```
[1] 2
```

```
# Comprobamos que se ha creado adecuadamente
# y el grupo tiene las dimensiones deseadas.
dim(train.set)
```

```
[1] 10560   181
```

Creamos ahora el modelo con la funcion `train` indicando el *method* = *mlp*, para lo que `caret` indica que necesitamos el paquete `RSNNS`.

5-fold crossvalidation

El paquete `caret` en `R` proporciona varios métodos para estimar la precisión de un algoritmo de aprendizaje automático.

El método de validación cruzada *k-fold* (5-fold crossvalidation) implica dividir el conjunto de datos en k-subconjuntos. Para cada subconjunto se mantiene mientras el modelo se entrena en todos los demás subconjuntos. Este proceso se completa hasta que se determina la precisión para cada instancia en el conjunto de datos, y se proporciona una estimación de precisión general.

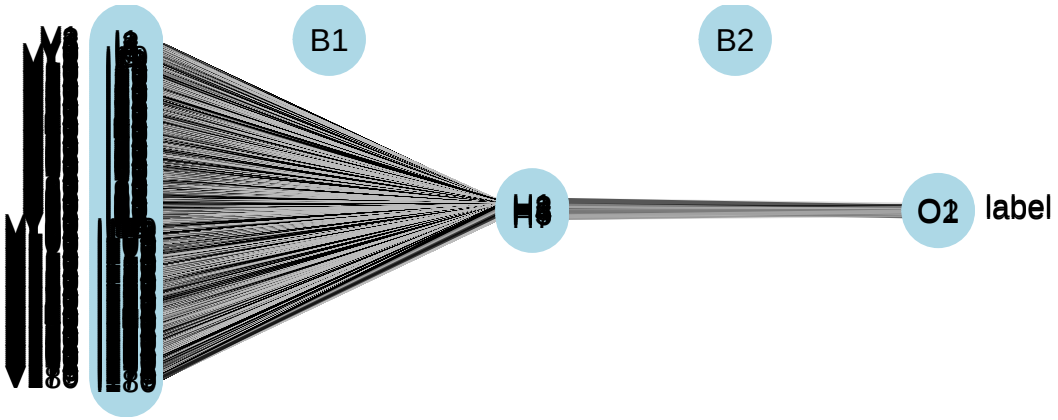
Es un método robusto para estimar la precisión y el tamaño de k y ajustar la cantidad de sesgo en la estimación, con valores populares establecidos en 3, 5, 7 y 10.

El siguiente ejemplo utiliza la validación cruzada 5 veces para estimar el metodo "mlp" en nuestro conjunto de datos. Con el argumento *trControl* definimos el control de entrenamiento y con *size* indicamos el número de nodos.

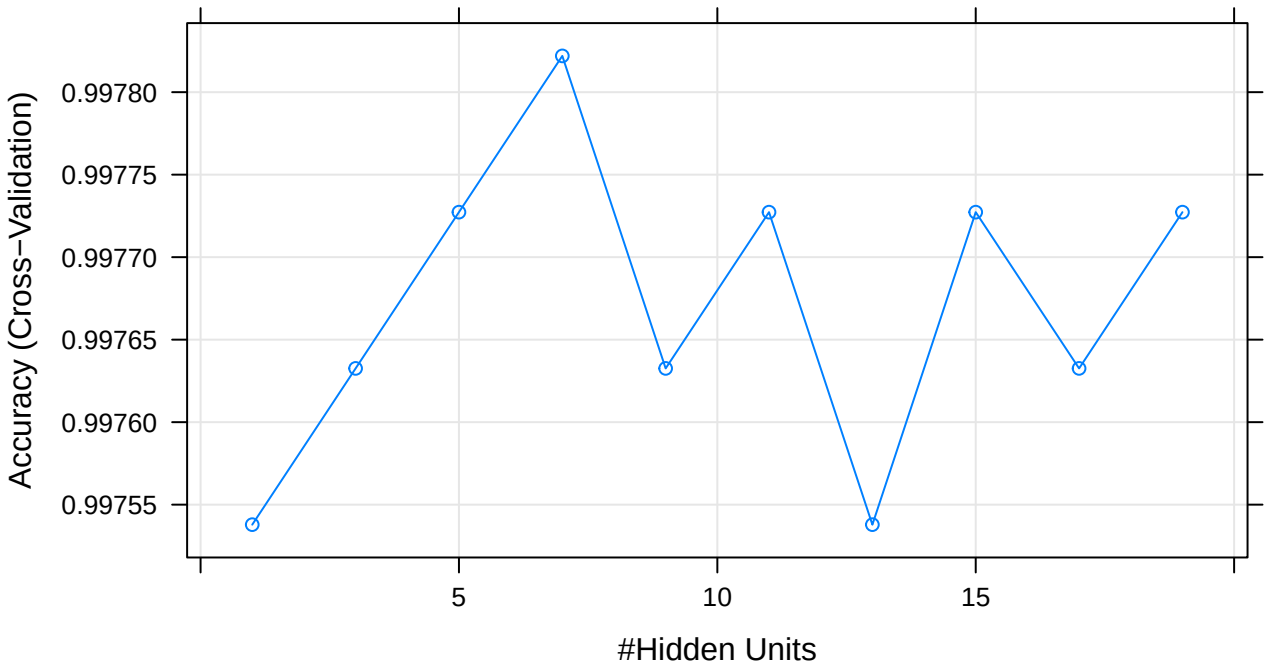
```
# modelo 5-crossvalidation sin repetición
model.fold5 <- train(label ~ ., train.set, method='mlp',
                trControl= trainControl(method='cv', number=5),
                tuneGrid= NULL, tuneLength=10 ,trace = FALSE, size=3)
model.fold5
```

```
Multi-Layer Perceptron

10560 samples
  180 predictor
    2 classes: 'NB', 'SB'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 8448, 8448, 8448, 8448, 8448
Resampling results across tuning parameters:

  size  Accuracy   Kappa
   1    0.9975379  0.9950758
   3    0.9976326  0.9952652
   5    0.9977273  0.9954545
   7    0.9978220  0.9956439
   9    0.9976326  0.9952652
  11    0.9977273  0.9954545
  13    0.9975379  0.9950758
  15    0.9977273  0.9954545
  17    0.9976326  0.9952652
  19    0.9977273  0.9954545

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was size = 7.
```

```
# Lo graficamos y vemos los resultados
plotnet(model.fold5, alpha=0.6)
```




```
plot(model.fold5)
```




```
# empleamos el modelo para predecir y
# comparamos la predicción con los datos de test
prediction.fold5 <- predict(model.fold5, test.set[-181])
table(prediction.fold5, test.set$label)
```

```
prediction.fold5   NB   SB
              NB 2626    1
              SB   14 2639
```

```
# Con la funcion "predict" también podemos obtener la probabilidad para cada clase:
prediction.fold5.prob <- predict(model.fold5, test.set[-181], type="prob")
head(prediction.fold5.prob)
```

	NB	SB
5	0.9999990	0.0000010
7	0.0005617	0.9994408
8	0.9999990	0.0000010
14	0.9999999	0.0000001
15	0.0005133	0.9994807
17	0.9999998	0.0000002

En este caso, la precisión se utilizó para seleccionar el modelo óptimo utilizando el valor más grande 99.78 %, que ocurre en el caso de 7 nodos. Como vemos, con un modelo *mlp* con 3 nodos se consigue un *accuracy* de validación promedio del 99.76 %. La diferencia en precisión con los modelos anteriores de 1 y 3 nodos es muy leve. Aunque es un modelo mucho más robusto que los anteriores, obtenemos más precisión con el modelo más sencillo, es decir, con un único nodo.

4.5. Algoritmo Support Vector Machine

4.5.1. Paso 1. Leer los datos transformados

Podemos emplear directamente el archivo creado en los pasos anteriores *data.pep* o cargar los datos desde la carpeta donde los hemos creado. Seguiremos este segundo paso:

```
# Mis datos

data.nn <- read_delim(file=file.path(params$folder.data,params$file2),
                      ",", trim_ws = TRUE )
length(complete.cases(data.nn))
```

```
[1] 15840
```

```
dim(data.nn)
```

```
[1] 15840   180
```

```
# Las caracteristicas de los datos las hemos visto en apartados anteriores.
```

4.5.2. Paso 2. Partición de los datos en training/test

Empleamos las variables binarias de base de datos transformada con la variable categorica *label* como última variable, al igual que hemos realizado para el modelo de red neuronales con el paquete `caret`.

```
#Partición de datos
set.seed(params$seed.train)
# Dataset codificado con la variable label como variable 181
data.label <- cbind(data.nn[,1:180],Class=peptidos[,2])

# Se convierte la variable respuesta a factor
data.label$label <- as.factor(data.label$label)

n <- nrow(data.nn)

# El 75% de los datos lo usamos para el entrenamiento
```

```
# empleamos el porcentaje indicado en el encabezado, p.train = 2/3
train <- sample(n,floor(n*params$p.train))
dataset.train <- data.label[train,]
dataset.test  <- data.label[-train,]

head(dataset.train[,(ncol(dataset.train)-4):ncol(dataset.train)])
```

	V177	V178	V179	V180	label
2463	1	0	0	0	SB
2511	0	0	0	0	SB
10419	0	0	0	1	SB
8718	0	0	0	0	NB
12483	0	0	0	0	NB
2986	0	0	0	1	SB

4.5.3. Paso 3. Función lineal y Gaussiana en el modelo SVM

El algoritmo de SVM que podemos emplear es la función `ksvm()` del paquete *kernlab*. Aunque hay otras opciones, como la funcion svm() del paquete `e1071` que identifica automáticamente si se trata de un problema de clasificación o regresión y si la variable respuesta es de tipo factor o de tipo numérico.

Modelo lineal

Para garantizar que los resultados sean reproducibles, incluimos la siguiente semilla; como tenemos el valor de la semilla incluida en el encabezado `YAML` del documento, solo debemos llamarla con el siguiente script. Se construye el modelo más sencillo: lineal, usando como kernel el valor `vanilladot`

```
set.seed(params$seed.clsfier) # garantiza resultados reproducibles

# Para que la función svm() calcule el Support Vector Classifier,
# se tiene que indicar que la función kernel es lineal.
model.svm <- ksvm(label~.,data=dataset.train, kernel='vanilladot')
```

```
 Setting default kernel parameters
```

```
# Vemos la información básica del modelo
model.svm
```

```
Support Vector Machine object of class "ksvm"

SV type: C-svc  (classification)
 parameter : cost C = 1

Linear (vanilla) kernel function.

Number of Support Vectors : 171

Objective Function Value : -16.3433
Training error : 9.5e-05
```

Se puede observar que la función lineal no tiene parámetros adicionales ('hiperparametros') al de coste.

En el siguiente paso empleamos los datos de test, datos nuevos, para ver como generaliza el modelo y evaluar su rendimiento y obtenemos la matriz de confusión.

```
model.svm.pred <- predict(model.svm, dataset.test)

# Modelo lineal
```

```r
res <- table(model.svm.pred, dataset.test$label)
(cmatrix <- confusionMatrix(res, positive="SB"))
```

```
Confusion Matrix and Statistics


model.svm.pred   NB   SB
            NB 2654    3
            SB    5 2618

               Accuracy : 0.9985
                 95% CI : (0.997, 0.9993)
    No Information Rate : 0.5036
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.997

 Mcnemar's Test P-Value : 0.7237

            Sensitivity : 0.9989
            Specificity : 0.9981
         Pos Pred Value : 0.9981
         Neg Pred Value : 0.9989
             Prevalence : 0.4964
         Detection Rate : 0.4958
   Detection Prevalence : 0.4968
      Balanced Accuracy : 0.9985

       'Positive' Class : SB
```

El modelo de SVM lineal con categoria positiva 'SB'(interacción péptido MHCI) obtiene una precisión de 0.998 y una sensitividad y especificidad de 0.999 y 0.998 respectivamente.

Modelo RBF

Ahora trataremos de mejorar el rendimiento de nuestro modelo implementando un SVM pero con un kernel Gaussiano, `rbfdot`. Tambien se podría aplicar la funcion `tune` del paquete `e1701`.

```r
set.seed(params$seed.clsfier)
modelo.svm.rbf <- ksvm(label ~ ., data = dataset.train, kernel = "rbfdot")

# Vemos la información básica del modelo
modelo.svm.rbf
```

```
Support Vector Machine object of class "ksvm"

SV type: C-svc  (classification)
 parameter : cost C = 1

Gaussian Radial Basis kernel function.
 Hyperparameter : sigma =  0.00291971510799512

Number of Support Vectors : 1289

Objective Function Value : -467.0374
Training error : 0.001989
```

```r
# Predicción
modelo.svm.rbf.pred <- predict(modelo.svm.rbf, dataset.test)
```

El resultado de la matriz de confusión con los datos de test es:

```r
# Modelo lineal
res2 <- table(modelo.svm.rbf.pred, dataset.test$label)

# Resultados
(cmatrix2 <- confusionMatrix(res2, positive = "SB"))
```

```
Confusion Matrix and Statistics


modelo.svm.rbf.pred   NB   SB
                 NB 2632    0
                 SB   27 2621

               Accuracy : 0.9949
                 95% CI : (0.9926, 0.9966)
    No Information Rate : 0.5036
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9898

 Mcnemar's Test P-Value : 5.624e-07

            Sensitivity : 1.0000
            Specificity : 0.9898
         Pos Pred Value : 0.9898
         Neg Pred Value : 1.0000
             Prevalence : 0.4964
         Detection Rate : 0.4964
   Detection Prevalence : 0.5015
      Balanced Accuracy : 0.9949

       'Positive' Class : SB
```

4.5.4. Paso 4. Comentar los resultados

El nuevo modelo de SVM con kernel gaussiano obtiene una precisión de 0.995 y una sensitividad y especificidad de 1 y 0.99 respectivamente. Vemos que el modelo obtenido con SVM lineal tiene una mayor precisión

Además, el modelo lineal que es mas sencillo, lo que ayuda a evitar en parte el overfitting de los modelos complejos.

4.5.5. Paso 5. Modelo svmRBF 5-fold crossvalidation con caret

Aun podemos profundizar más sobre la precisión del modelo aplicando la técnica cross-validation en la elección de los conjuntos de train y test. De esta manera minimizamos el efecto de la elección del conjunto de datos a la hora de determinar la bondad del modelo. Usamos los grupos de entrenamiento y test creados en el apartado anterior con la funcion `createDataPartition` para el paquete `caret`, ya que este paquete admite datos de tipo factor.

```r
#Partición de datos
set.seed(params$seed.train)
# El 75% de los datos lo usamos para el entrenamiento
data.Train <- createDataPartition(y=peptidos$label,
                                  p=0.66666666666666667, list=FALSE)
# Dataset codificado con la variable label como variable 181
```

```
data.label <- cbind(data.nn[,1:180],Class=peptidos[,2])

train.set <- data.label[data.Train,]
test.set  <- data.label[-data.Train,]

nrow(train.set)/nrow(test.set) # debe estar alrededor de 2
```

```
[1] 2
```

```
# Comprobamos que se ha creado adecuadamente y
# el grupo tiene las dimensiones deseadas.
dim(train.set)
```

```
[1] 10560   181
```

Empleamos ahora el paquete **caret** con el modelo **svmRBF** para aplicar el algoritmo de SVM con 5-fold crossvalidation.

```
# modelo 5-crossvalidation
set.seed(params$seed.clsfier)
model.rbf.fold5 <- train(label ~ ., train.set, method='svmRadial',
                trControl= trainControl(method='cv', number=5),
                tuneGrid= NULL, trace = FALSE)

model.rbf.fold5
```

```
Support Vector Machines with Radial Basis Function Kernel

10560 samples
  180 predictor
    2 classes: 'NB', 'SB'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 8448, 8448, 8448, 8448, 8448
Resampling results across tuning parameters:

  C     Accuracy   Kappa
  0.25  0.9881629  0.9763258
  0.50  0.9913826  0.9827652
  1.00  0.9946023  0.9892045

Tuning parameter 'sigma' was held constant at a value of 0.002905427
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were sigma = 0.002905427 and C = 1.
```

```
# empleamos el modelo para predecir y
# comparamos la predicción con los datos de test
prediction.rbf.fold5 <- predict(model.rbf.fold5, test.set[-181])
res <- table(prediction.rbf.fold5, test.set$label)
confusionMatrix(res, positive="SB")
```

```
Confusion Matrix and Statistics


prediction.rbf.fold5   NB   SB
                  NB 2609    0
                  SB   31 2640
```

```
               Accuracy : 0.9941
                 95% CI : (0.9917, 0.996)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9883

 Mcnemar's Test P-Value : 7.118e-08

            Sensitivity : 1.0000
            Specificity : 0.9883
         Pos Pred Value : 0.9884
         Neg Pred Value : 1.0000
             Prevalence : 0.5000
         Detection Rate : 0.5000
   Detection Prevalence : 0.5059
      Balanced Accuracy : 0.9941

       'Positive' Class : SB
```

Como vemos, con un modelo mlp con 3 nodos se consigue un accuracy de validación promedio del 99.41 %. Hay una leve diferencia en precisión con los modelos anteriores de 1 y 3 nodos, que dan una precisión algo superior. Sin embargo, es un modelo mucho más robusto que los anteriores, pero debido al tiempo de ejecución, quizá en este caso no es necesario su aplicación.

4.6. Discusión final

Tanto los modelos de redes neuronales artificiales (NNA) como las máquinas de soporte vectorial (SVM) son aplicables en este estudio, debido a su capacidad para clasificar datos y representar relaciones desconocidas a partir de los mismos datos. En primera instancia, se puede enfatizar que los 2 modelos mostraron resultados satisfactorios. La mejor precisión de pronóstico para las NNA y las SVM fue de 99.89 % y 99.85 % respectivamente, ambos en los modelos más sencillos.

Para evaluar los métodos, se analizan las matrices de confusión para cada problema y en cada método. Una matriz de confusión mide el número de clasificaciones que se realizan correctamente y también las clasificaciones que pertenecen a una clase y que se asumen como otra, y de estas pueden extraerse indicadores como la precisión, la exactitud, la sensibilidad y la especificidad, indicadores claros del desempeño de un clasificador automático.

modelo	**Accuracy**	**Sensitivity**	**Specificity**	**kappa**	**SB+**
ANN 1 nodo	0.9989	0.9996	0.9981	0.9977	2620
ANN 3 nodos	0.9979	0.9992	0.9966	0.9958	2619
caret 'mlp'	0.9976			0.9953	2639
SVM lineal	0.9985	0.9989	0.9981	0.9970	2618
SVM radial	0.9949	1.0000	0.9898	0.9898	2621
caret svmRBF	0.9941	1.0000	0.9883	0.9883	2640

Indiferentemente de los resultados del experimento, se debe tener en cuenta un punto a favor de utilizar modelos de SVM: el algoritmo detrás del modelo permite ajustarse a problemas no lineales y la solución se realiza bajo programación cuadrática, lo cual hace que su solución sea única y generalizable.

Otro aspecto relevante que se obtuvo del experimento fue determinar qué funciones kernel eran las que mejor se adaptaban al modelo de SVM. Se implementaron dos funciones kernel (base radial y lineal). Los resultados revelaron que el mejor desempeño de las SVM se obtuvo aplicando kernels de base lineal - RBF (99.85 %). Por otro lado, en este tipo de escenario, en el que el número de predictores es varios órdenes de magnitud mayor que el de observaciones, los modelos son proclives a sufrir overfitting. Esto sugiere que, de entre los diferentes tipos de kernels, sea adecuado emplear el de menor flexibilidad, el kernel lineal.

Por lo que tanto el modelo NNA de un nodo como el modelo SVM de kernel lineal serian aplicables en nuestro caso y nos proporcinan una alta precisión en los resultados.

5. Bibliografía

Core Team, RCTR, and others. 2013. "R: A Language and Environment for Statistical Computing." *R Foundation for Statistical Computing, Vienna*.

Freeman, JA, and DM Skapura. 1993. "Algoritmos Aplicaciones Y Técnicas de Propagación."

Github, Inc. 2016. "GitHub."

Kuhn, Max. 2012. "The Caret Package." *R Foundation for Statistical Computing, Vienna, Austria*. https://cran.r-project.org/package= caret.

———. 2017. "A Short Introduction to the Caret Package." *R Foundation for Statistical Computing, Vienna, Austria*. https://cran.r-project.org/web/packages/caret/vignettes/caret.pdf.

Lantz, Brett. 2015. *Machine Learning with R*. Packt Publishing Ltd. http://www.packtpub.com/books/content/machine-learning-r.

Sánchez Anzola, Nicolás. 2015. "Máquinas de Soporte Vectorial Y Redes Neuronales Artificiales En La Predicción Del Movimiento Usd/Cop Spot Intradiario." *ODEON-Observatorio de Economía Y Operaciones Numéricas*, no. 9.

Suárez, Enrique J Carmona. 2014. "Tutorial Sobre Máquinas de Vectores Soporte (Svm)." *Tutorial Sobre Máquinas de Vectores Soporte (SVM)*, 1–12.

Wagih, Omar. 2017. "ggseqlogo: a versatile R package for drawing sequence logos." *Bioinformatics* 33 (22): 3645–7. https://doi.org/10.1093/bioinformatics/btx469.