Sistemas de Visión Artificial

Práctica 5. Machine Learning

Nombre: Juan Casado Ballesteros

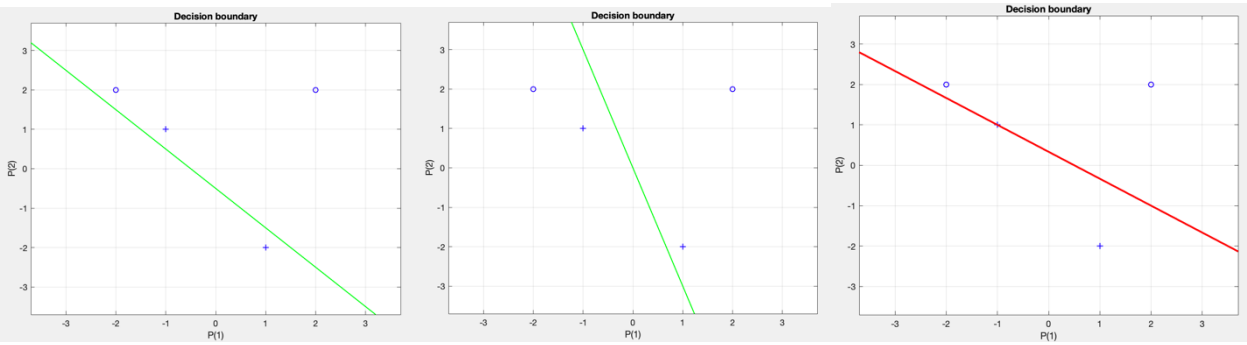
Fecha: 28 de noviembre 2019

Perceptrón

Ejecutamos el código que se nos ha proporcionado realizando una pequeña modificación. A cada iteración del entrenamiento, es decir, a cada llamada que se realiza sobre:

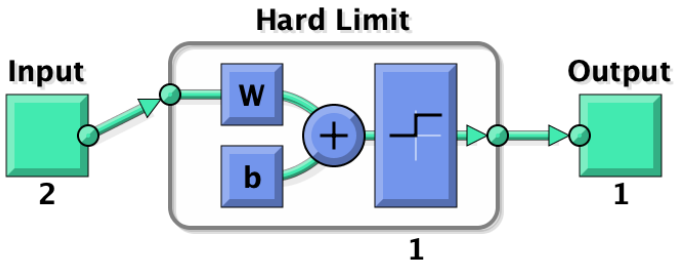
```
[net,Y(n),E(n)] = adapt(net,P(:,n),T(:,n));
```

Tomaremos el valor de los pesos y las bias mostrando la evolución de la frontera de decisión según la red está siendo entrenada.



Tras hacer esto descubrimos que la red no modifica sus pesos en todas las iteraciones si no que solo lo hace en algunas. Deducimos que esto se puede deber a que la red está siendo entrenada con muy pocos datos lo cual no es ideal.
La última foto se corresponde con la frontera de decisión final de la red. Vemos que, aunque el error que esta frontera es nulo, es decir, todos los valores son clasificados correctamente, la frontera en sí no lo es. Podría estar mucho más separada de los datos que clasifica como haría una SVM.

```
view(net)
```



Podemos ver que nuestra red tiene una sola neurona en la capa oculta con la función de activación hardlim, que es la propia de los perceptrones. La red tendrá dos pesos y una bia pues tiene dos entradas y solo una neurona. La salida que la red produce es binaria por lo que podremos clasificar solo en dos clases linealmente separables mediante esta red.

¿Por qué cambian los pesos y el bias?

Los pesos y el bias cambian debido a los errores existentes en la clasificación anterior.
Cuando estos existan los pesos y las bias serán modificados por el descenso del gradiente hasta que el error sea mínimo. Una vez que lleguemos a ese error mínimo las modificaciones dejarán de producirse.
Dependiendo de la suerte que hayamos tenido y de lo bueno que sea nuestro algoritmo de entrenamiento el mínimo en el que nos hayamos parado podrá ser global o local.

¿Es correcta la frontera de decisión final? ¿Por qué?

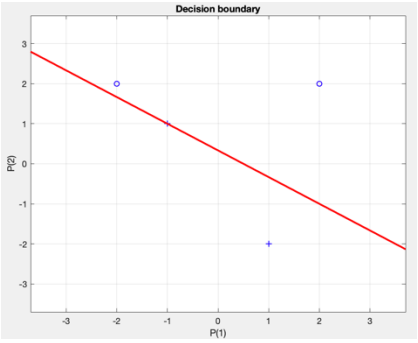
La función de decisión obtenida es correcta pues clasifica de forma adecuada todos los puntos. No obstante, como ya hemos indicado esta aparentemente podría ser mejor para este caso concreto si se pareciera más a la frontera que una SVM generaría proporcionando una frontera de decisión que maximizara una zona de exclusión entorno a dicha frontera en lugar de conformarse con simplemente haber clasificado los puntos.

Finalizar según el error cuadrático

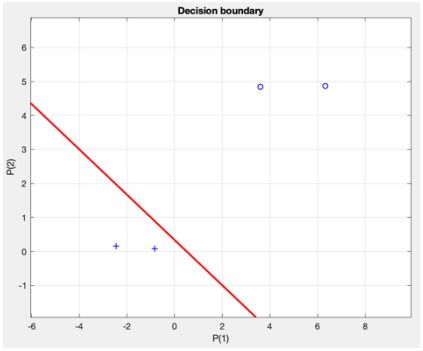
Calculando el error cuadrático cada vez que la red ha sido entrenada con todos lo valores podremos parar de forma más precisa su entrenamiento que si lo hiciéramos solo por iteraciones.
También podríamos medir el error cuadrático cada vez que evaluamos un elemento nuevo.

```
while mean_error > 0.00001 && count < 200
    for n=1:length(P)
        [net3,Y(n),E(n)] = adapt(net,P(:,n),T(:,n));
        Weight(n+1,:) = net.IW{1};
        Bias(n+1,:) = net.b{1};
    end
    mean_error = sse(E)
    count = count + 1;
end
```

Ejecutando e entrenamiento de este modo obtenemos la misma frontera de decisión que ya mostramos al principio del documento. No obstante, ahora estamos seguros de que dicha frontera realmente produce un error suficientemente bueno.



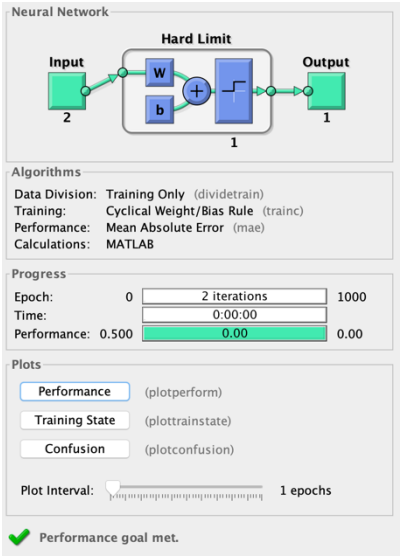
Clasificación con valores nuevos



Obtendremos de forma aleatoria nuevos valores y comprobamos que somos capaces de separarlos mediante la frontera de decisión.

```
out = sim(net,P_sim);
plotpv(P_sim, out);
```

Entrenamiento en batch



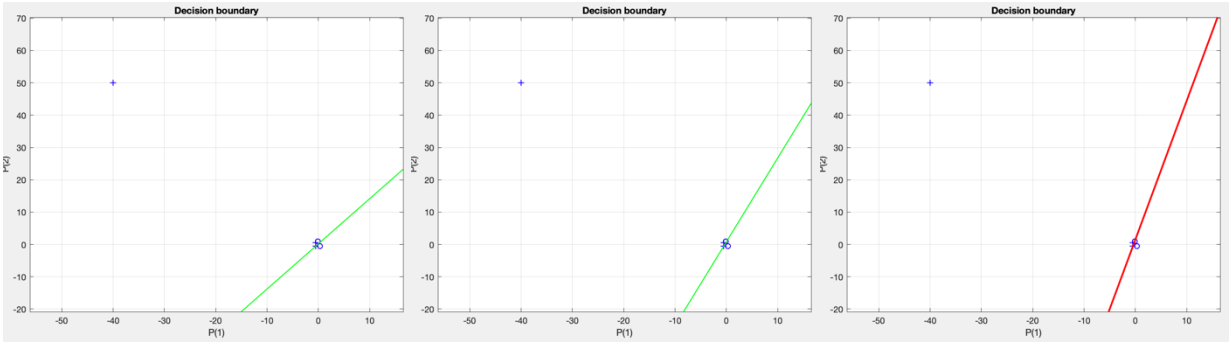
La función que se utiliza para realiza entrenamiento en batch realiza un proceso similar a este para entrenar las redes.

```
net = perceptron;
net = train(net,P,T);
```

Esta función calcula el error que se produce al entrenar tanto en los datos de entrenamiento como en los de validación. Dejará de entrenar cuando el error de validación comience a aumentar evitando de este modo el overfitting.
Podremos saber los resultados de nuestro entrenamiento explorando los menús que aparecen al usar esta función.
La frontera de decisión que este tipo de entrenamiento nos proporciona es la misma que ya teníamos.
Usar entrenamiento en batch dividiendo los datos en validación, entrenamiento y test con tan pocos datos no tiene demasiado sentido. Si los datos fueran más y fueran más complejos el entrenamiento en batch puede producir resultados distintos, generalmente mejores que el entrenamiento adaptativo.

Entrenamiento con outliers

Introduciremos ahora un outlier en los datos de entrenamiento. Su presencia hará que la red tarde mucho más en entrenar, en total realiza 35 entrenamientos sobre el conjunto de los datos.
Después de todas las iteraciones logra clasificar todos los datos y finaliza el entrenamiento.

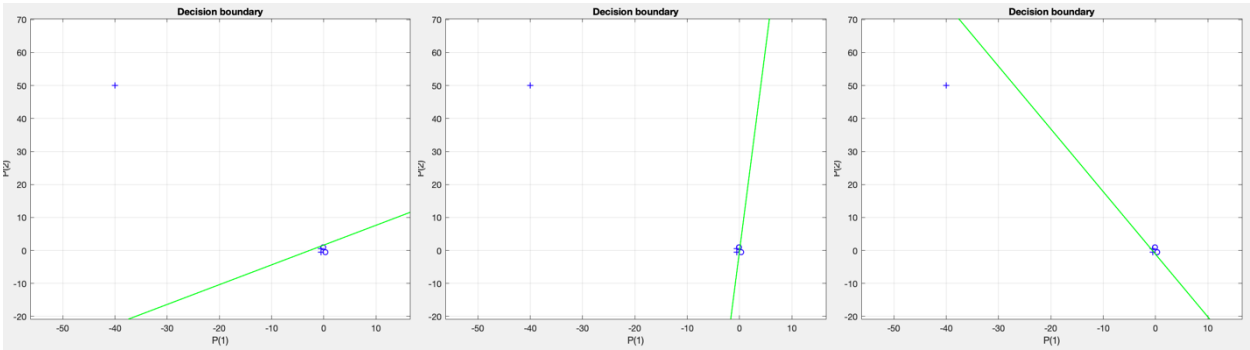


Mostramos dos estados intermedios de la red mientras entrenaba y la frontera de decisión resultado.

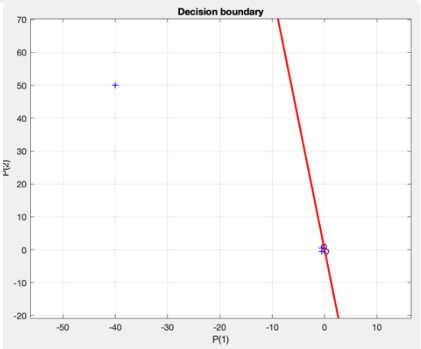
```
P = [-0.5 -0.5 +0.3 -0.1 -40; -0.5 +0.5 -0.5 +1.0 50];
T = [1 1 0 0 1];
net = perceptron;
net = configure(net,P,T);
mean_error = 1000;
Weight(1,:) = net.IW{1};
Bias(1,:) = net.b{1};
Y = [];
E = [];
count = 0;
while mean_error > 0.00001 && count < 200
    for n=1:length(P)
        [net3,Y(n),E(n)] = adapt(net3,P(:,n),T(:,n));
        Weight(n+1,:) = net3.IW{1};
        Bias(n+1,:) = net3.b{1};
    end
    mean_error = sse(E)
    count = count + 1;
end
```

Normalización de valores de entrada

Normalizar los datos que tenemos es una buena práctica pues reduce el impacto de los outliers mejorando tanto los rendimientos como la calidad de los resultados.
Mostramos el resultado en tres iteraciones distintas del entrenamiento y la frontera de decisión final.
Podemos ver que el resultado es distinto al obtenido anteriormente, aparentemente parece un resultado mejor.



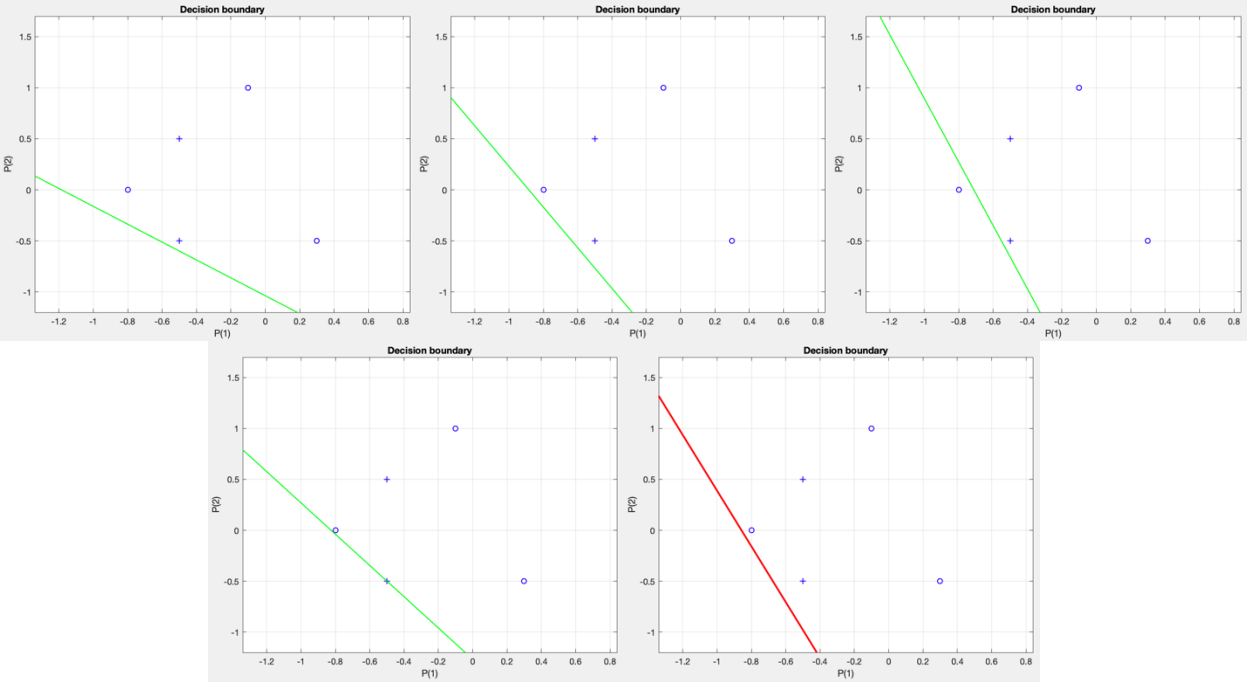
Se han necesitado solo 5 iteraciones sobre el conjunto de los datos para llegar a una clasificación sin error.
La normalización se consigue con el parámetro learnpr.




```
net = perceptron('hardlim','learnpn');
```

Datos no separables linealmente

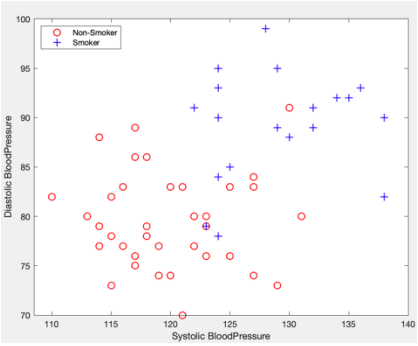
Si a una red le proporcionamos datos que no es capaz de clasificar agotará las iteraciones máximas que hayamos establecido sin haber logrado reducir el error. En el caso del perceptrón estos serán datos no separables linealmente.
Con los datos indicados alcanzamos las 200 iteraciones sin lograr una clasificación válida.



Se puede ver que la red no realiza progresos en la clasificación ya que tampoco puede hacerlos.

K-Nearest Neighbours

Comenzamos el análisis de los datos por visualizarlos. Visualizar los datos no siempre es posible debido a que estos pueden tener un número de dimensiones elevado. No obstante, cuando sí se puede este suele ser un buen punto de partida.



Podemos ver que los datos no son linealmente separables. Están organizados en dos clusters no demasiado separados.

```
k = 1;
ClassifierModel = fitcknn(data_train,label_train,'NumNeighbors',k);
```

Creamos un clasificador de K-NN para K=1, es decir, cada dato se clasificará solo según su vecino más cercano.

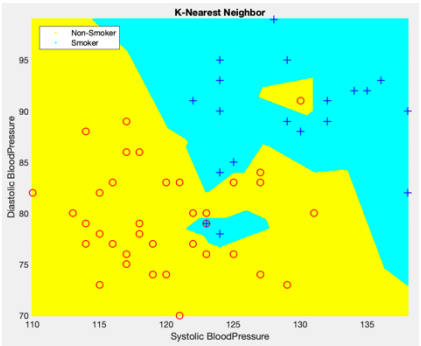
Mostramos a continuación la superficie de decisión que se genera. Ya que estamos utilizando K=1 aparecen dos pequeñas islas, cada una de una clase, completamente rodeadas por zonas de clasificación de la otra clase.
Aparentemente esto parece indicar que los datos de esas islas podrían ser outliers.

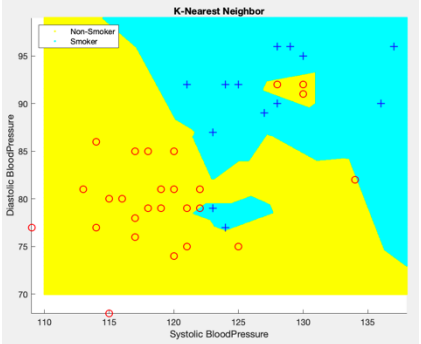
Utilizamos ahora los datos de test que nos habíamos reservado anteriormente para comprobar la veracidad de nuestro modelo.
Mostramos el resultado de la clasificación sobre las superficies de decisión anteriores.

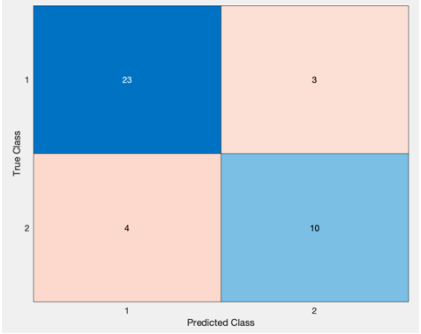
```
test_clasification = predict(ClassifierModel, data_test);
```

Para comprobar la valided del modelo utilizaremos una matriz de confusión en las que se nos indicará qué valores fueron clasificados como pertenecientes a cada clase siendo de ella tanto como los valores clasificados como de una clase no siéndolo.
Podemos ver que la mayoría de los datos fueron bien clasificados habiendo sido solo 7 mal clasificados.

```
ConfMat = confusionmat(label_test, test_clasification);
```







Adicionalmente implementamos el mismo cálculo que se realiza en la función `confusionmat` obteniendo los mismos resultados. Calculamos también la especificidad y la sensibilidad.

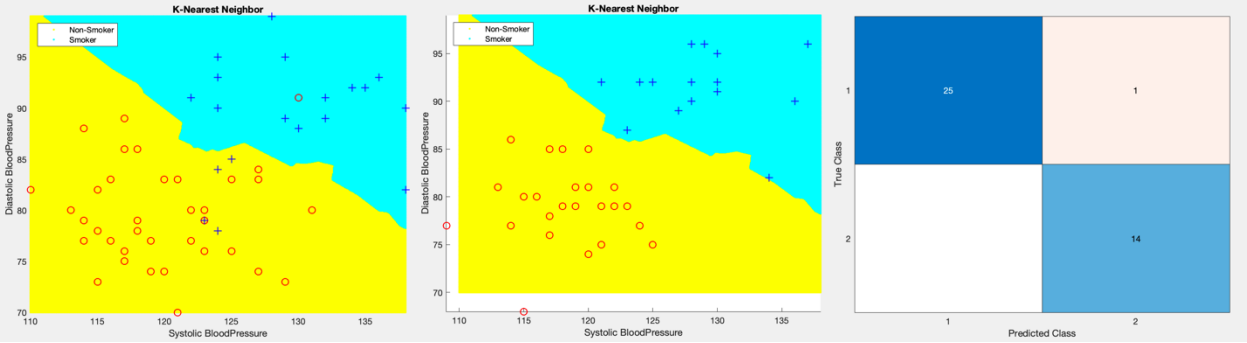
```
for i = 1:size(label_test,1)
    if test_clasification(i) && label_test(i)      % Era 1 y da 1
        TP = TP + 1;
    elseif test_clasification(i) && ~label_test(i) % Era 0 y da 1
        FP = FP + 1;
    elseif ~test_clasification(i) && label_test(i) % Era 1 y da 0
        FN = FN + 1;
    else                                           % Era 0 y da 0
        TN = TN + 1;
    end
end
especificidad = TN/(TN+FP);
sensibilidad = TP/(TP+FN);


TP = 10
TN = 23
FP = 3
FN = 4
especificidad = 0.8846
sensibilidad = 0.7143
```

Podemos ver que los resultados obtenidos ponderados por la cantidad de datos que tenemos realmente no eran tan buenos como parecían.

Podemos ver que con distintos modelos obtenidos cambiando el número de vecinos en función de los que realizamos la clasificación obtenemos distintos resultados. El valor de K que elijamos dependerá del tamaño a partir del cual consideremos que una agrupación de puntos forma un clúster con significado o simplemente son outliers.

K=5

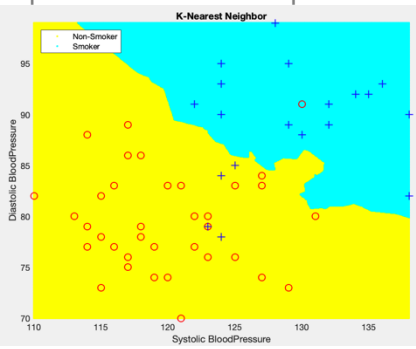
Con una k mayor vemos que las islas que apreciábamos antes ya no están lo cual repercute positivamente en la clasificación obtenida, esos valores eran realmente outliers.



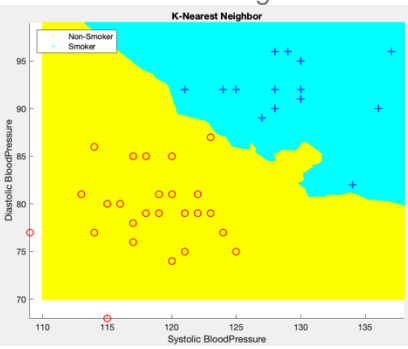
Datos de entrenamiento | Datos de test | Matriz de confusión

K=10

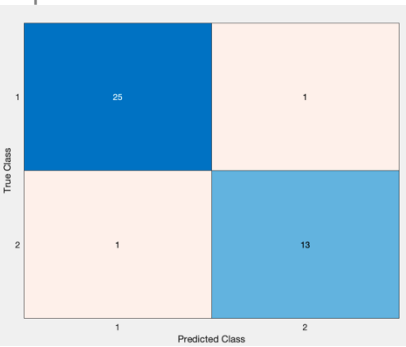
Del mismo modo que con una k demasiado pequeña obtendremos malos resultados con una k demasiado grande también sucederá lo mismo. Si utilizamos demasiados vecinos para determinar la pertenencia de cada punto al final deformaremos la generalización que k-NN realiza.



Datos de entrenamiento



Datos de test

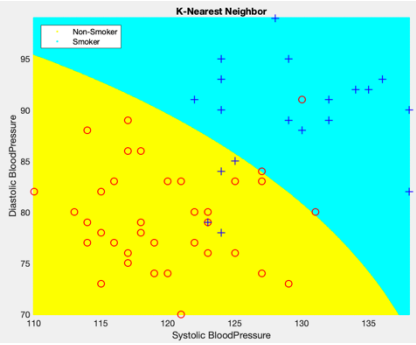


Matriz de confusión

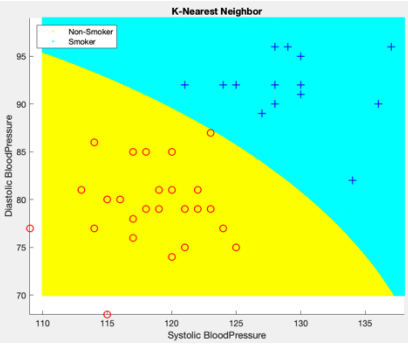
Clasificación naive-bayes

Cambiamos ahora de modelo de clasificación, pero seguiremos utilizando los mismos datos.
Comprobaremos también como este modelo es capaz de realizar clasificaciones y obteniendo superficies de decisión que nos permitan inferir la clase más probable a la que un nuevo dato pertenecería.

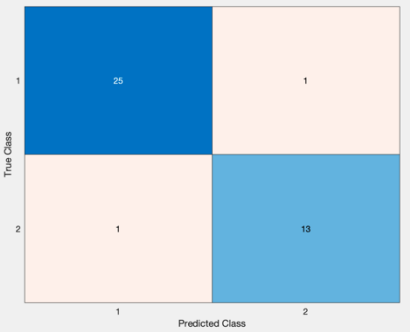
```
ClassifierModel = fitcnb(data_train,label_train);
```



Datos de entrenamiento



Datos de test



Matriz de confusión

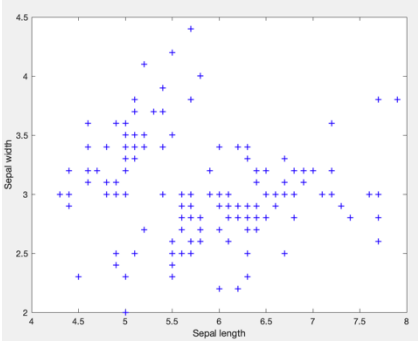
En base a otros principios y con unas superficies de decisión muy distintas naive-bayes nos proporciona una buena clasificación capaz de inferir adecuadamente sobre los datos de test.

Tanto k-means como naive-bayes son algoritmos de clasificación supervisada en los que proporcionamos unos datos ya clasificados a partir de los que entrenamos un modelo que posteriormente será capaz de predecir.
Sin esos datos de entrenamiento ya clasificados no podríamos utilizar ninguno de estos métodos.

K-means

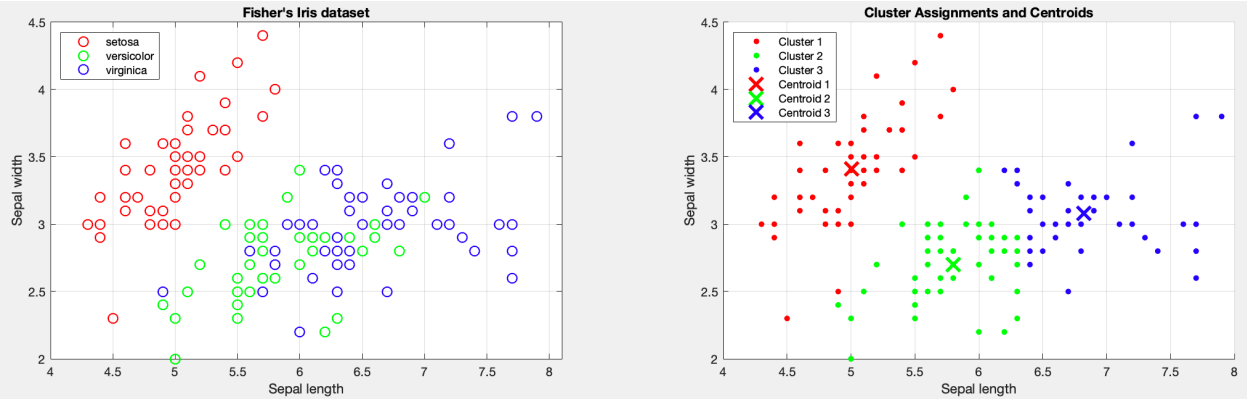
Es un algoritmo de cauterización o clasificación no supervisada. Para utilizarlo no nos hacen falta datos ya clasificados, el será capaz de determinar las posibles clases de los datos a partir de su estructura.
Una desventaja de esto es que, aunque cada dato sea asignado con una clase puede que no sepamos identificar qué características del mundo real son las que determinan dicha clase. En este algoritmo las clases son solo nomenclatura.

Cargamos y visualizamos los datos que k-means utilizará. El algoritmo verá los datos como en esta foto, sin ninguna clase asignada. Nuestros datos no obstante si tienen clases ya asignadas en base a las que comprobaremos la eficacia de k-means para clusterizar estos datos.



Realizamos la clusterización sobre 3 clases pues ya sabemos que son 3 las clases existentes en nuestros datos.

```
k = 3;
[idx,C] = kmeans(data_iris,k);
```



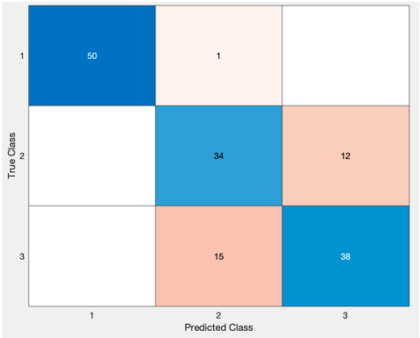
Podemos ver que sobre los datos reales dos de las clases no están demasiado separadas por lo que de antemano sabemos que será en ellas donde k-means más fallará.
Visualizamos las clases obtenidas junto a los centroides que las representan y a partir de los cuales se obtuvieron.

Cada ejecución de k-means dará un resultado ligeramente distinto a las otras. Esto se debe a que los centroides se inician de forma aleatoria. Hemos repetido las ejecuciones hasta que los identificadores de los centroides han coincidido con los identificadores de las clases originales.
Para no tener que hacer esto habría que buscar qué clase tiene mayor coincidencia con las otras.

Calculamos ahora la matriz de confusión de la clasificación para comprobar la efectividad de k-means obteniendo las clases que ya conocíamos.

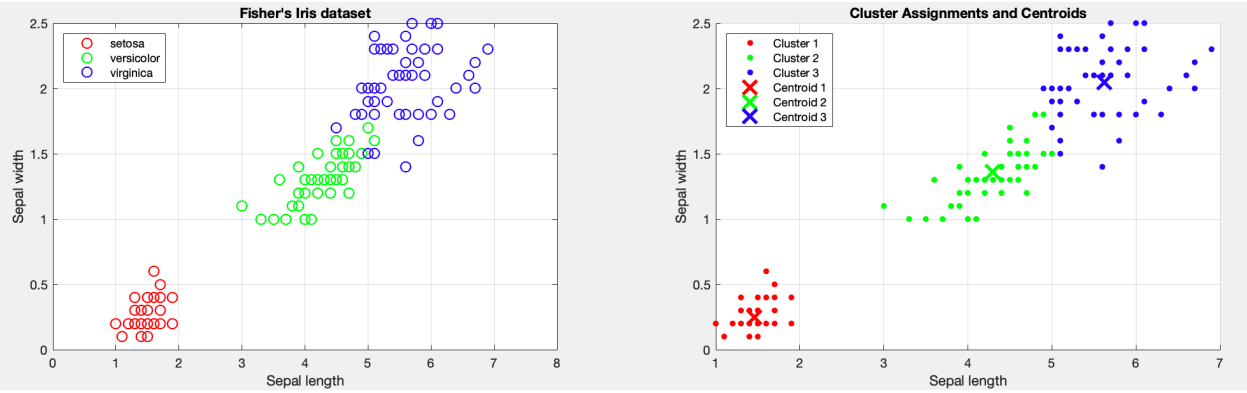
```
names = categorical();
for i = 1:size(idx, 1)
    names = [names; names_iris{idx(i)}];
end
ConfMat = confusionmat(names, label_iris);
figure;
confusionchart(ConfMat);
```

Obtenemos la matriz de conclusión a partir de la cual calculamos la calidad de los resultados obtenidos.
Podemos ver como una de las clases es clasificada con gran precisión mientras que entre dos de ellas hay errores cruzados. Estas dos clases son las que no estaban tan separadas entre si por lo que a k-means le cuesta más diferenciarlas.
Calculamos también la precisión de la clasificación obtenido un resultado moderadamente bueno, hemos clasificado correctamente en solo el 80% de los casos.

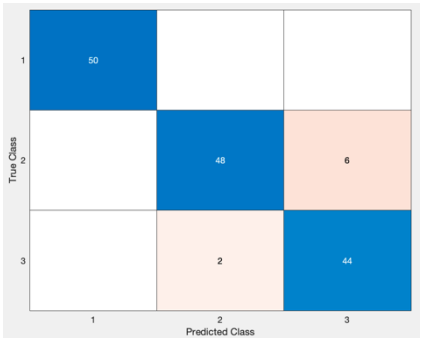



```
total = sum(ConfMat,"all");
hits = 0;
for i = 1:size(ConfMat, 1)
    hits = hits + ConfMat(i,i);
end
acc=hits/total
%acc = 0.8133
```

Repetimos ahora el análisis esta vez realizando la cauterización por otra de las variables de las que disponemos. En este caso se puede ver como las clases originales a la izquierda son más heterogéneas que en el caso anterior por lo que es de esperar que k-means proporcione mejores resultados que antes.



Obtenemos la matriz de confusión de la clasificación. Una de las clases ha sido completamente clasificad de forma adecuada mientras que en dos hay una cantidad mínima de errores cruzados. Estos errores se producen en la frontera entre ambas clases la cual es algo difusa.
Obtenemos una precisión mayor que antes por lo que clasificar estos datos por esta variable proporciona mejores resultados que hacerlo por la que utilizamos en la primera parte.




```
acc = 0.9467
```