



UNIVERSIDAD DE GRANADA

INTELIGENCIA DE NEGOCIO
GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 1

RESOLUCIÓN DE PROBLEMAS DE CLASIFICACIÓN Y ANÁLISIS
EXPERIMENTAL.

Autor

José María Sánchez Guerrero

Rama

Sistemas de Información



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2020-2021

Índice

1. Introducción	2
2. Procesado de datos	3
3. Configuración de algoritmos	5
3.1. K-Nearest-Neighbors (k-NN)	5
3.2. Decision Tree	6
3.3. Naive-Bayes	8
3.4. Neural Network	9
3.5. Support Vector Machine (SVM)	11
4. Análisis de los resultados	13
5. Interpretación de los resultados y conclusión	15
Referencias	18

1. Introducción

En este trabajo vamos a analizar el comportamiento de distintos algoritmos de clasificación en el problema propuesto. Disponemos de un dataset, llamado "*Mammographic Mass dataset*", en el cual se desea predecir el tipo de tumor (benigno o maligno) en una serie de mamografías realizadas para un estudio sobre el cáncer de mama. Este estudio lo vamos a realizar gracias a los siguientes atributos proporcionados en el dataset:

- **BI-RADS.** Este parámetro representa un control de calidad de las mamografías. Consta de 7 categorías distintas, en las que, cuanto más alto sea el valor, hay una mayor probabilidad de que sea maligno.
- **Edad del paciente.**
- **Forma de la masa.** Dependiendo de como sea la masa anormal detectada, se clasifica como **R**edondeada, **O**valada, **L**obulada, **I**rregular ó **N**o definida.
- **Margen de masa.** Circumscribed = 1, microlobulated = 2, obscured = 3, ill-defined = 4, spiculated = 5 (nominal).
- **Densidad de la masa.** Valores entre 1 y 4, siendo 1 la más alta y 4 contenido graso (no tumoral).
- **Severidad.** Es el atributo que se desea predecir, es decir, si es un tumor benigno o maligno.

En el dataset hay datos de 961 pacientes, sin embargo, nos gustaría dejar un porcentaje para validar el modelo y así ver cómo va entrenando los datos. Posteriormente, se explicará cómo se ha determinado qué datos son los de entrenamiento y cuáles son los de test.

2. Procesado de datos

Lo primero que tenemos que hacer es mostrar varios de los datos que tenemos y analizarlos. En mi caso vamos a sacar las 5 primeras filas:

	BI-RADS	Age	Shape	Margin	Density	Severity
0	5.0	67.0	L	5.0	3.0	maligno
1	4.0	43.0	R	1.0	NaN	maligno
2	5.0	58.0	I	5.0	3.0	maligno
3	4.0	28.0	R	1.0	3.0	benigno
4	5.0	74.0	R	5.0	NaN	maligno

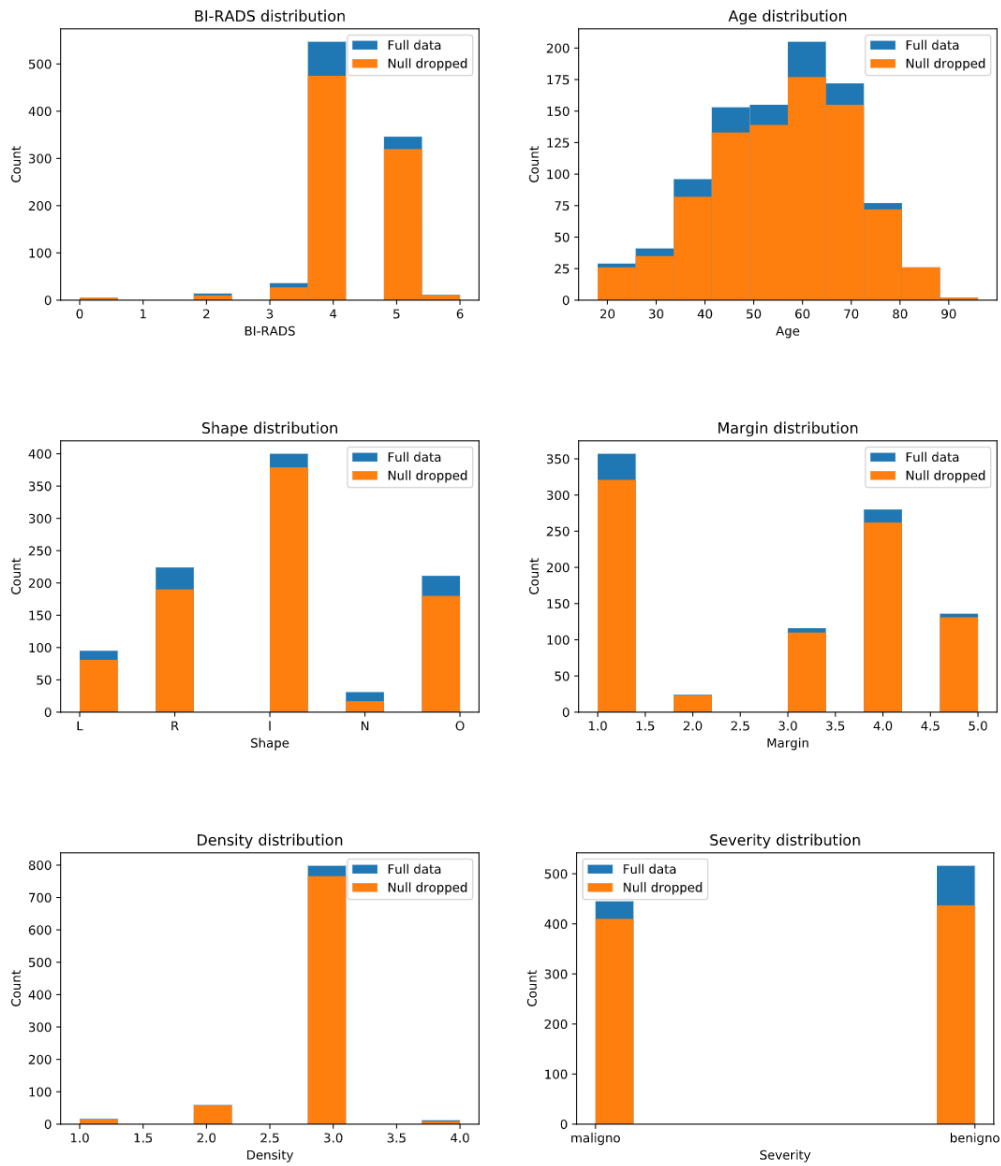
Podemos observar que tenemos tanto datos numéricos, como datos categóricos, como ya comentamos en la introducción. También podemos observar que tenemos varias celdas con datos erróneos o perdidos (representados con el valor *NaN*), por lo que será importante procesarlos para que nuestros algoritmos funcionen correctamente. Primero veamos qué cantidad de estos datos nulos tenemos:

BI-RADS	2
Age	5
Shape	0
Margin	48
Density	76
Severity	0

Son una cantidad bastante alta de datos, en comparación con la cantidad de datos totales que tenemos. Por lo tanto, eliminar toda las filas que contengan uno, puede dejarnos con muy pocos datos para entrenar y validar, y que el modelo sea más débil. No obstante, el introducir datos para reemplazar uno faltante ha de realizarse con cuidado, ya que no son datos reales.

También tenemos que tener en cuenta cuál es la distribución de estos datos antes de trabajar con ellos. Es decir, tenemos que asegurarnos de no sesgar nuestros datos si los eliminamos. Si hay algún tipo de correlación, tendríamos que intentar completarlos de alguna forma. Para ello, vamos a generar una gráfica para cada uno de los atributos del *dataset*, en la que mostraremos la cantidad de datos antes y después de eliminarlos, y así ver cómo están distribuidos.

Los resultados son los siguientes:



Podemos observar que los datos nulos están distribuidos aleatoriamente entre los atributos, por lo que podremos eliminarlos del *dataset* sin ningún problema (y teniendo en cuenta que tendremos menos datos para trabajar).

3. Configuración de algoritmos

Para todos los algoritmos hemos procedido de la misma manera, y así evaluarlos a todos en igualdad de condiciones. Para comenzar, declaramos el clasificador con sus parámetros correspondientes (en nuestro caso, las primeras evaluaciones han sido con los parámetros por defecto y una semilla *random_state* = 0).

Posteriormente, hacemos las predicciones correspondientes para los datos de entrenamiento. Lo hacemos mediante validación cruzada de 5 particiones, gracias a la función:

$$\text{cross_val_predict}(\text{classifier}, x_train, y_train, cv = 5)$$

Por último, para evaluar los resultados obtenidos vamos a usar:

- **Classification report.** Crea un informe que muestra las principales métricas de clasificación: precisión, recall, f1-score, y promedios macro, ponderado y de la muestra.
- **Score.** Misma medida del informe anterior mostrada con un poco más de precisión.
- **Confusion matrix.** Muestra una matriz para evaluar la precisión de la clasificación. Cada fila representa las instancias de una clase predicha, mientras que cada columna representa las instancias reales de ésta.

Los algoritmos que evaluaremos han sido elegidos porque, cada uno de ellos, tiene una forma de procesar los datos diferente a todos los demás. Estos algoritmos son los siguientes:

3.1. K-Nearest-Neighbors (k-NN)

Comenzamos por este algoritmo ya que es uno de los más utilizados, debido a su simplicidad. Este algoritmo funciona de la siguiente manera. Cuando tenemos un nuevo ejemplo a clasificar, calcula la distancia (Euclídea) con respecto a los datos ya existentes, y considerando los *k* más cercanos, determina si pertenece a una clase u otra.

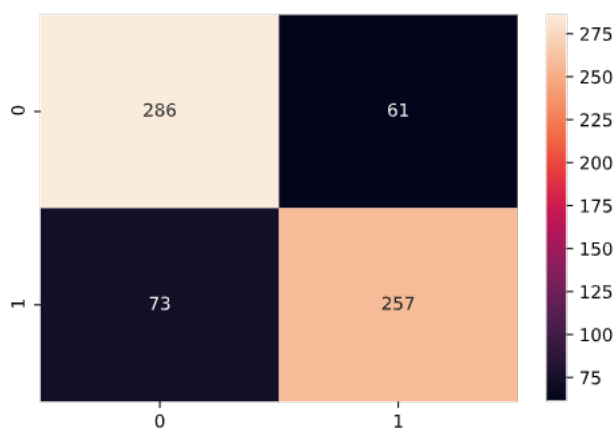
A la hora de implementarlo lo hacemos de la siguiente forma:

KnnClf = *KNeighborsClassifier*(*n_neighbors* = 5)

Seleccionamos un número **k=5** de vecinos a evaluar y dejamos el resto de parámetros que trae el algoritmo por defecto. El parámetro anterior es el más relevante, sin embargo, algunos otros interesantes a estudiar pueden ser, por ejemplo: **p**, que sirve para cambiar el tipo de distancia utilizada (Euclídea, Manhattan o Minkowski); ó **weights**, que determina la influencia de los vecinos en la predicción (todos ponderan igual, los cercanos influyen más o tu propia función).

Resultados obtenidos

	precision	recall	f1-score	support
0	0.80	0.82	0.81	347
1	0.81	0.78	0.79	330
accuracy			0.80	677
macro avg	0.80	0.80	0.80	677
weighted avg	0.80	0.80	0.80	677



SCORE: 0.7991137370753324

3.2. Decision Tree

Este clasificador, como su propio nombre indica, es un árbol en el que cada **hoja** es una clase y cada **nodo** es un nodo de decisión con una prueba simple a realizar.

El algoritmo funciona de la siguiente forma. Todos los ejemplos de entrenamiento comienzan desde el nodo raíz y se dividen recursivamente los ejemplos en base a los atributos seleccionados. Esta selección de atributos se suele realizar mediante el criterio "*gini*", pero también hay otros como "*InfoGain*" o "*GainRatio*". Posteriormente, quitamos las ramas con ruido o con datos anómalos.

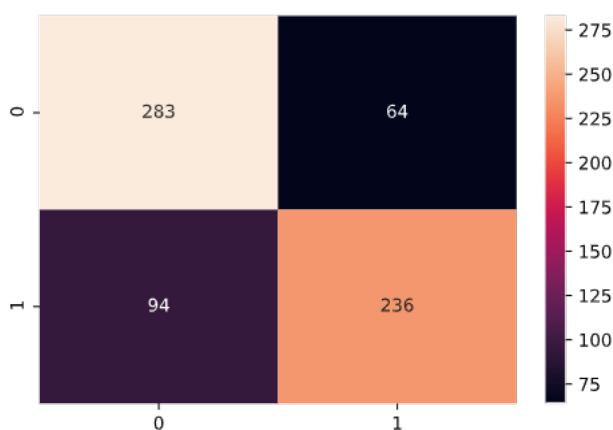
A la hora de implementarlo lo hacemos de la siguiente forma:

DecTreeClf = *tree.DecisionTreeClassifier*(*random_state* = 0)

El parámetro **random_state** nos sirve para seleccionar la semilla. El resto de parámetros los dejamos por defecto, ya que la mayoría son utilizados para limitar el número de hojas, el número de características, etc. que en nuestro ejemplo no necesitaremos. También tenemos el parámetro **criterion**, que nos sirve para elegir el criterio de selección de atributos comentado anteriormente.

Resultados obtenidos

	precision	recall	f1-score	support
0	0.75	0.82	0.78	347
1	0.79	0.72	0.75	330
accuracy			0.77	677
macro avg	0.77	0.77	0.77	677
weighted avg	0.77	0.77	0.77	677



SCORE: 0.7666174298375185

3.3. Naive-Bayes

Se ha utilizado porque es el modelo de red bayesiana orientada a clasificación más simple. Este algoritmo se considera un estándar y sus resultados son competitivos, pese a utilizar una hipótesis poco realista. Es decir, suponemos que todos los atributos son independientes a partir de su clase, así que la hipótesis MAP (Máximo a posteriori) en un Naive-Bayes queda así:

$$C_{MAP} = \arg_{c \in \Omega_C} \max P(c|a_1, \dots, a_n) = \arg_{c \in \Omega_C} \max P(c) \prod_{i=1}^n P(a_i|c)$$

En la práctica, existen dependencias entre variables, por lo que puede llevar a una falta de precisión que no se puede retocar en un clasificador como este (existen redes de creencia bayesianas para solucionar esto).

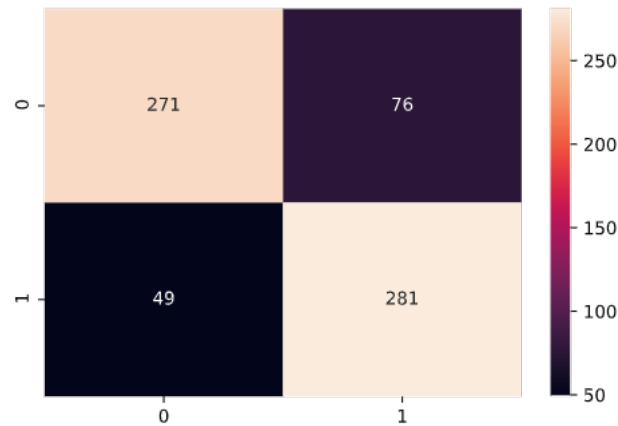
A la hora de implementarlo lo hacemos de la siguiente forma:

$$GaussianClf = GaussianNB()$$

No ponemos ningún parámetro ya que es un clasificador que no se puede modular o retocar, como acabamos de decir.

Resultados obtenidos

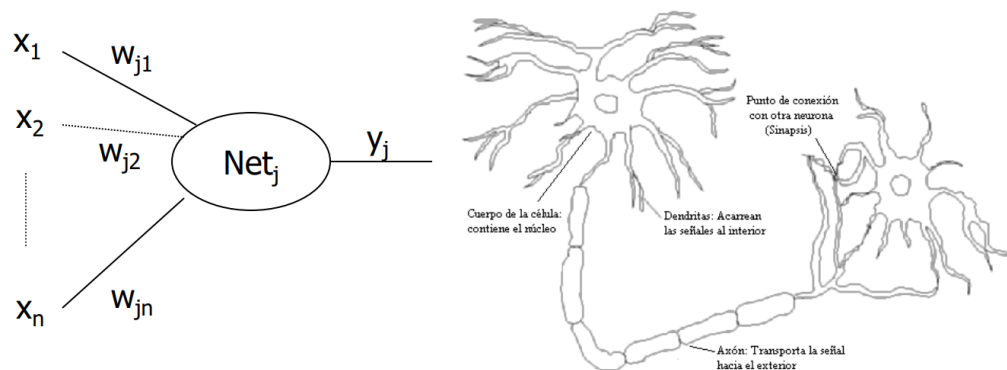
	precision	recall	f1-score	support
0	0.85	0.78	0.81	347
1	0.79	0.85	0.82	330
accuracy			0.82	677
macro avg	0.82	0.82	0.82	677
weighted avg	0.82	0.82	0.82	677



SCORE: 0.8153618906942393

3.4. Neural Network

Estos algoritmos están basados en los propios sistemas nerviosos biológicos:



Como podemos ver en la figura, cada señal que llega a las dendritas $\{x_1, x_2, \dots, x_n\}$ serían nuestros datos de entrada del modelo. Las sinapsis o puntos de conexión con otra neurona, serían los pesos $\{w_1, w_2, \dots, w_n\}$ que ponderan a las entradas: **positivo** es una sinapsis excitadora y **negativo** es una sinapsis inhibidora.

Después, la actividad interna de cada célula sería la sumatoria entre cada una de las conexiones anteriores, y por último, la **función de activación** que, dadas

estas entradas, define una salida para el modelo:

$$y_j = f(Net_j - \theta_j) = f(\sum w_{ji} \cdot x_i - \theta)$$

Para este trabajo, no vamos a implementar una red neuronal desde cero, ya que es una tarea bastante compleja. Sin embargo, vamos a utilizar un **MultiLayer Perceptron (MLP)**, que es una red neuronal compuesta por: una capa de entrada, capas ocultas (o *hidden layers*) y una capa de salida. Excepto los datos de entrada, cada nodo es una neurona que utiliza una función de activación no lineal (sigmoideal es la más común), y en el entrenamiento utiliza la técnica de *backtracking*

A la hora de implementarlo lo haremos de la siguiente forma:

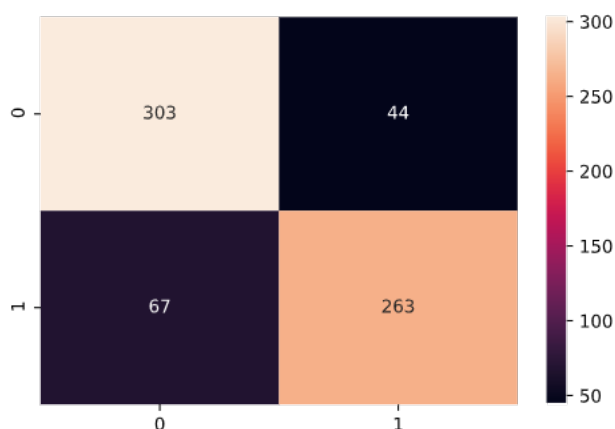
```
PerceptronClf = MLPClassifier(hidden_layer_sizes = 100, random_state = 0)
```

Aquí podemos ver la cantidad de capas ocultas que queremos que tenga nuestra red. Es el parámetro más importante, ya que cuanto más alto sea, más aprenderá de los datos que tenemos. Esto no quiere decir que tengamos que poner un valor muy alto, ya que puede causar sobreaprendizaje en nuestro modelo. El otro parámetro que aparece, al igual que en los otros clasificadores, nos sirve para seleccionar la semilla.

También existen otros parámetros importantes, como pueden ser: **activation**, con la que podemos seleccionar la función de activación a utilizar; **early_stopping**, que sirve para finalizar el entrenamiento si no se mejora durante un número X de iteraciones; o parámetros relacionados con los pesos como **solver** (para la optimización de los pesos), **learning_rate** (tasa de aprendizaje) ó **tol** (tolerancia).

Resultados obtenidos

	precision	recall	f1-score	support
0	0.82	0.87	0.85	347
1	0.86	0.80	0.83	330
accuracy			0.84	677
macro avg	0.84	0.84	0.84	677
weighted avg	0.84	0.84	0.84	677



SCORE: 0.8360413589364845

3.5. Support Vector Machine (SVM)

El último algoritmo que vamos a utilizar es uno de los más populares en cuanto a la clasificación binaria. Una SVM construye un hiperplano entre las dos clases de forma que la separación entre ellas se amplie al máximo. Se van transformando los datos de entrada en un espacio de características mediante el **producto interno (escalar)**.

A la hora de implementarlo lo haremos de la siguiente forma:

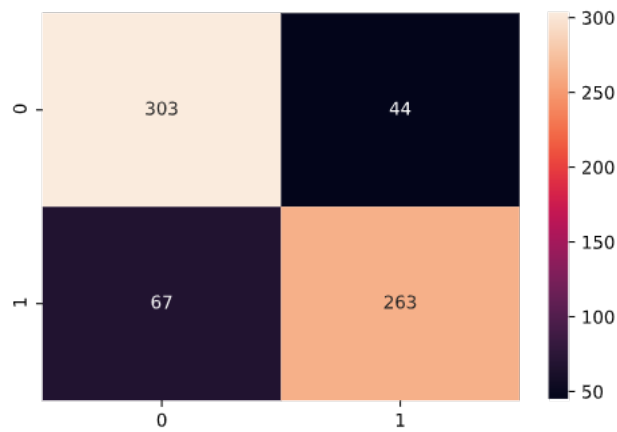
$$SvmClf = SVC(random_state = 0, gamma = 'auto')$$

El primer parámetro que tenemos aquí es la semilla, al igual que en los otros. El siguiente parámetro es para seleccionar el coeficiente del **kernel** (el cual también es otro parámetro que también se podrá modificar).

También tenemos otros parámetros como son la **C**, para la regularización; **tol** o tolerancia, como en las redes neuronales; o hasta un **max_iter** por si queremos poner un máximo de iteraciones.

Resultados obtenidos

	precision	recall	f1-score	support
0	0.81	0.88	0.84	347
1	0.86	0.78	0.82	330
accuracy			0.83	677
macro avg	0.83	0.83	0.83	677
weighted avg	0.83	0.83	0.83	677

**SCORE:** 0.8301329394387001

4. Análisis de los resultados

Esta es la tabla resumen con todos los resultados:

Algoritmo	precision	f1-score
kNN	0.79911	0.81
DecisionTree	0.7666	0.78
Naive-Bayes	0.8153	0.80
Neural Network	0.83604	0.84
SVM	0.83013	0.83

Podemos observar que, en todas las ejecuciones, hemos obtenido aproximadamente un 80 % de precisión, es decir, los modelos han acertado un 80 % de los casos que se le ha propuesto. El que mejor lo ha hecho de todos ha sido el **MultiLayer Perceptron**, con una precisión del 0.836041 de aciertos; y el que peor lo ha hecho ha sido el **Decision Tree** con un 0.7666174 de aciertos.

No obstante, podemos decir que todos los clasificadores que hemos escogido, lo han hecho bastante bien a priori. Que el clasificador con la menor precisión tenga aproximadamente un 75 % de acierto es muy buena señal, aunque ahora tenemos que mirar que la clasificación la ha hecho correctamente (por ejemplo, ha podido clasificarlos todos como 'maligno' y que una gran mayoría de los casos que le han llegado sean también 'maligno').

Para esto, vamos a mirar las matrices de confusión. Como podemos ver, los datos están perfectamente distribuidos, es decir, tenemos similares cantidades de datos tanto de una clase como de otra. Además, al tener tasas de acierto altas, se puede ver como los valores altos de la matriz corresponden a su diagonal (y que a su vez corresponden a los *True positives* y a los *True negatives*).

- **k-NN**. Pese a ser un modelo que no aprende, ya que consulta a los datos de entrenamiento cada vez que tiene que predecir una etiqueta, tiene una precisión alta. También hemos comentado lo simple que es, por lo que cambiar parámetros y probar otros métodos para ver si mejoramos la precisión es muy fácil. Por ejemplo, subiendo el k-valor a 100 se consigue mejorar la precisión hasta el $\approx 83\%$, lo que nos dice que el modelo es bastante sensible al ruido y a los datos desequilibrados.

A parte de que es sensible al ruido, es un algoritmo que depende de la distancia, por lo que un estudio más en profundidad de una función de distancia adecuada, también puede mejorar los resultados.

- **DecisionTree.** Este modelo, al igual que el anterior, es fácil de entender y utilizar. Además es eficiente y trabaja bien con datos con ruido, algo que le costaba más al algoritmo anterior.

Sin embargo, hemos obtenido peores resultados, lo cual se debe a varios factores. Lo primero es que no detectan correlación entre los atributos, es decir, que no tiene en cuenta si dos características van cambiando sistemáticamente para cada uno de los datos. Por otro lado, el mayor problema que les veo es el sobreaprendizaje y la división del dominio en regiones rectangulares, y a la hora de validar el modelo es donde más precisión pierde.

- **Naive-Bayes.** Como ya pudimos preveer al describir el algoritmo, ha dado unos resultados muy competitivos. Lo más probable es que la falta de precisión (la poca que hay, no es que el modelo sea malo) sea debida a la falta de dependencia entre las variables, lo cual es complicado de retocar en este modelo. Debido a esto yo preferiría indagar e investigar más otros clasificadores a este.

- **Neural Network.** Es el que mejor resultado nos ha dado, principalmente por su eficacia antes los valores con ruido o irregulares, su eficacia para evaluar nuevos casos y, sobre todo, que aprende a medida que le van llegando nuevos datos (y puede seguir aprendiendo después de entrenar con los actuales).

Puede ser bastante complejo el modificar una red neuronal, aunque modificar el MLP que hemos propuesto es más factible que, por ejemplo, el modelo anterior. También podemos crear una red desde 0, adaptada nuestro conjunto de datos. En mi caso, yo me quedaría este modelo por su variabilidad de propuestas para resolverlo, a parte de que ha sido el que mejor porcentaje de acierto nos ha dado.

En contra tenemos que es un modelo con un coste en tiempo muy alto, y que gran parte del entrenamiento es por ensayo y error, debido a que es muy difícil de interpretar.

- **SVM.** Este modelo es bueno en comparación a otros ya que la complejidad computacional se reduce. También tiene parámetro de regularización, por lo que se puede evitar el sobreajuste; y con el parámetro del kernel puede ir aprendiendo. De ahí que haya obtenido un gran resultado.

Lo que más ha podido afectar al clasificador son los datos con ruido, y en un futuro, también le puede afectar que no es adecuado para conjuntos de datos grandes (al contrario que le le pasa a las redes neuronales, que siguen aprendiendo).

5. Interpretación de los resultados y conclusión

Como ya he dicho, me voy a quedar con el **MultiLayer Perceptron** de las redes neuronales debido a la gran flexibilidad que tiene y a las posibilidades que nos ofrece para cambiar de modelo. Como también hemos visto que ha sido el modelo que mejor ha funcionado, podremos utilizarlo para intentar mejorar el preprocesado de los datos, y que no sea simplemente eliminar los datos que son nulos.

Por ejemplo, podemos probar a utilizar PCA sobre el conjunto de datos. Esto lo que hace es un análisis de los componentes principales y reduce la dimensionalidad de los datos, de forma que los descompone en valores singulares (busca las variables no correlacionadas) y, posteriormente, los elimina del conjunto. El resultado ha sido el siguiente:

	precision	recall	f1-score	support
0	0.81	0.87	0.84	347
1	0.85	0.79	0.82	330
accuracy			0.83	677
macro avg	0.83	0.83	0.83	677
weighted avg	0.83	0.83	0.83	677

SCORE: 0.8286558345642541

Podemos observar que el resultado apenas cambia, por lo que podemos deducir que el analizador no ha encontrado datos no correlacionados y tampoco los ha modificado mucho. Por lo tanto, sólo seguiremos utilizando el *StandardScaler* que ya teníamos.

A continuación, probaremos modificando parámetros de nuestro modelo elegido. Para ello, voy a realizar varias ejecuciones modificando el número de capas ocultas, el solver para la optimización de pesos y el parámetro "*early_stopping*" por si hay excesivo sobreajuste en el modelo. El resultado lo mostraré en la siguiente tabla:

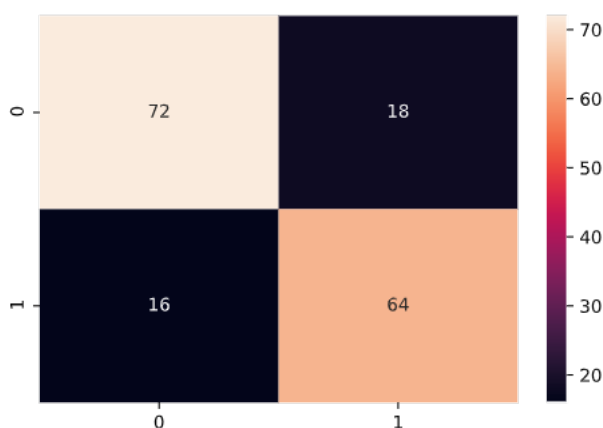
hidden_layer_sizes	25	50	50	100	200	200
solver	adam	adam	adam	adam	adam	adam
early_stopping	False	False	True	False	False	True
Score	0.82865	0.82570	0.79025	0.83604	0.83161	0.80945

hidden_layer_sizes	25	50	50	100	200	200
solver	sgd	sgd	sgd	sgd	sgd	sgd
early_stopping	False	False	True	False	False	True
Score	0.82127	0.81979	0.78581	0.82274	0.82570	0.81388

Podemos ver que el resultado apenas cambia entre unas ejecuciones y otras. Por lo general, los mejores resultados se obtienen sobre las 100 capas ocultas, siendo el solver *'adam'* (el que trae por defecto) ligeramente mejor que el *'sgd'*. En cuanto al parámetro *"early_stopping"* podemos ver que tampoco cambia demasiado los resultados, por lo que podemos que apenas teníamos sobreajuste en los datos.

A continuación, viendo que las pruebas no mejoran demasiado el modelo, procederé a ejecutar el conjunto de test con el modelo que más precisión ha obtenido: 100 capas ocultas con *'adam'* y sin *'early_stopping'*. Este ha sido el resultado final:

	precision	recall	f1-score	support
0	0.82	0.80	0.81	90
1	0.78	0.80	0.79	80
accuracy			0.80	170
macro avg	0.80	0.80	0.80	170
weighted avg	0.80	0.80	0.80	170



SCORE: 0.8

Como podemos observar, el resultado ha sido lo que esperábamos, y es que ha obtenido prácticamente los mismos valores que los modelos anteriores. Que haya obtenido un 80 % exacto nos puede hacer pensar que está mal hecho, sin embargo, al tener tan pocos datos, es posible que se haya dado esto (es más, si miramos la matriz de confusión y sumamos los verdaderos positivos y negativos, podremos comprobar que es el 80 % exacto del total de ejemplos).

Todas estas ejecuciones y, viendo los resultados que hemos obtenido, lo que nos lleva a pensar es que el modelo está *'atascado'* en este punto. Es decir, que en cuanto

a entrenamiento no vamos a obtener muchos mejores resultados con los datos que actualmente (obviando que podemos crear un modelo que esté sobreajustado de más, algo que tampoco nos conviene). Un conjunto con 961 datos, de los cuales unos 120 aproximadamente contienen atributos nulos o faltantes, y que a su vez tenemos que dividirlo en datos para entrenamiento y test; pues es un dataset que se nos queda bastante corto.

Pese a todo esto, con los modelos analizados obtenemos unos porcentajes sobre 80 % de acierto, que es un porcentaje muy bueno. Con esto quiero decir que, con un dataset más grande y con una o dos características más que sean relevantes para decir si el tumor es maligno o benigno (aunque esto son temas que no puedo solucionar, ni sé si se puede hacer algo más) pienso que podríamos obtener unos resultados muy buenos y una tasa de acierto realmente alta.

Referencias

- [1] Scikit-Learn. *SVC*
<https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC>
- [2] Scikit-Learn. *RandomForestClassifier*
<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier>
- [3] Scikit-Learn. *MLPClassifier*
https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier
- [4] Scikit-Learn. *StandardScaler*
<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>
- [5] Scikit-Learn. *PCA*
<https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html>
- [6] Scikit-Learn. *plot_learning_curve*
https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html#sphx-glr-auto-examples-model-selection-plot-learning-curve-p
- [7] Scikit-Learn. *recall_score*
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html
- [8] Scikit-Learn. *precision_score*
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html
- [9] *Should I normalize/standardize/rescale the data?*
<http://www.faqs.org/faqs/ai-faq/neural-nets/part2/>
- [10] Wikipedia. *Sensitivity and specificity*
https://en.wikipedia.org/wiki/Sensitivity_and_specificity
- [11] DataFlair. *Kernel Functions-Introduction to SVM Kernel & Examples*
<https://data-flair.training/blogs/svm-kernel-functions/>
- [12] Isaac Changhau. *Loss Functions in Neural Networks*
https://isaacchanghau.github.io/post/loss_functions/

- [13] MathWorks *Support Vector Machine*
<https://es.mathworks.com/discovery/support-vector-machine.html>
- [14] Scikit-Learn. *cross_val_score*
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html
- [15] Scikit-Learn. *train_test_split*
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
- [16] Scikit-Learn. *Confusion Matrix*
https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html