UNIVERSIDAD DE GRANADA

AA

Aprendizaje Automático

Práctica Final

Autores:
Mario Carmona Segovia
Francisco José Aparicio Martos

Profesor: Nicolás Pérez de la Blanca Capilla

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Curso 2020 - 2021

Índice

Índice de figuras

Índice de tablas

1. Analizar y comprender el problema a resolver

Con la base de datos nuclear feature extraction for breast tumor diagnosis [1] se busca resolver el problema de diagnosticar tumores de mama con alta precisión.

Las medidas que se muestran en la base de datos han sido obtenidas de una serie de imágenes, tomadas a muestras de fluidos de los tumores. En la tareas de aprendizaje no se trabaja directamente con las imágenes si no que se usan características de las imágenes.

De cada imagen se extraen las siguientes características:

- Radio ← El radio de un núcleo individual se mide por la media de la longitud entre el centroide de la serpiente y los puntos individuales de la serpiente.
- Perímetro ← La distancia total entre los puntos de la serpiente
- Area ← El área nuclear es mide por el número de píxeles en el interior de la serpiente y añadiendo una mitad de los píxeles del perímetro.
- Compacidad ← El perímetro y área son combinados para medir la compacidad de los núcleos celulares usando la formula $perímetro^2/area$.
- Suavidad ← La suavidad de un núcleo se mide por la media de las diferentes longitudes de una línea radial y la media de la longitud que rodean al núcleo.
- Concavidad ← Medimos el número y la gravedad de las concavidades en un núcleo celular.
- Puntos cóncavos ← Es similar a la concavidad pero se mide sólo el número de concavidades.
- Simetría ← Se obtiene el eje de mayor tamaño o el acorde de mayor tamaño que pasa por el centro y se mide la distancia perpendicular al eje en ambas direcciones.
- Dimensión fractal ← La dimensión fractal de una celda es aproximada usando la aproximación de la linea de costa descrita por Mandelbrot.
- Textura ← La textura de una celda nuclear se mide por la varianza de la intensidad de la escala de gris de los píxeles.

Una vez calculadas estas características se obtiene el valor medio, el mayor valor y el error estándar de cada característica de la imagen. De este proceso resultan las 30 características que tiene nuestro problema, por lo que nuestra matriz de características se representaría de la siguiente forma X = 1 x R^{30}.

Con respecto a las clases tenemos un total de 2 clases, por lo tanto se trata de un problema de clasificación binaria. Una clase indica que un tumor es maligno y otra clase indica que un tumor es benigno.

La BBDD de la que disponemos contiene 569 ejemplos, teniendo cada uno 32 datos. Entre estos datos encontramos las 30 características, la clase a la que pertenece el ejemplo, y el id del ejemplo. Este id es eliminado nada más leer los datos, ya que no se utiliza para entrenar el

modelo. La clase del ejemplo es separada de las características para formar los datos de entrada y salida del problema.

Adicionalmente al leer los datos del archivo que los contiene, se añade una columna de valores nan al final de la matriz de datos, que es eliminada al empezar el programa después de la lectura de los datos.

Esta BBDD no ha sido descargada del repositorio UCI del problema [2], ya que los dataset que se encuentran en este repositorio no tenían parte de la información, como la característica a la que pertenecia cada columna de la BBDD. Por ello hemos descargado la BBDD del problema a través del siguiente enlace. Esta BBDD se encuentra en la carpeta datos dentro del proyecto, con el nombre de data.csv .

Un dato a destacar es que la BBDD no contiene datos pérdidos, por lo que no es necesario realizar este paso en el preprocesado de los datos.

1.1. Separación de los datos

Además otro aspecto a tener en cuenta en la resolución del problema es que sólo se dispone de un conjunto de datos, es decir, no se dispone en principio de un conjunto de datos para training y otro para test, por lo que será necesario que nosotros hagamos esa separación con los datos.

Al tratarse de un problema de clasificación debemos tener cuidado con mantener la representatividad de cada clase en ambos conjuntos de datos, es decir, se deben mantener las proporciones de las clases que hay en el conjunto de datos inicial en los dos nuevos conjuntos de datos.

El principal parámetro a decidir en esta separación de los datos es el porcentaje de datos que tendrá cada uno de los nuevos conjuntos. Dado que disponemos de un número limitado de datos, hay que decidir bien como se va ha realizar la separación, ya que si decidimos darle más cantidad de datos a test, tendremos un buen conjunto para validar el modelo entrenado, pero tendremos muy pocos datos para entrenar un buen modelo; y si decidimos darle más cantidad de datos a training, entrenaremos muy bien el modelo pero no tendremos suficientes datos como para poder validarlo de forma correcta. Por lo general se suele dividir en un 20 % de los datos para test y el restante 80 % para training, y esta es la separación que hemos decidido realizar.

Para realizar la separación hemos empleado la función train_test_split de la librería scikit learn, a la cual hay que indicarle el porcentaje de datos para test, en este caso el 20 %, y como se trata de un problema de clasificación en el que hace falta mantener la representatividad de cada clase, debemos pasar al parámetro stratify el conjunto de datos del que disponemos, para que pueda comprobar la proporción de cada clase en el conjunto de datos inicial.

A partir de la separación de los datos sólo trabajaremos con los datos training para entrenar el modelo, y los datos de test no serán usados hasta la estimación del error de generalización, para que sigan siendo insesgados.

En las siguientes tablas se muestra la separación realizada:

	Tamaño	**Porcentaje**
X	569	100.0 %
X_train	455	79.96 %
X_test	114	20.04 %

Tabla 1: Separación de los datos de entrada

	Tamaño	**Porcentaje**
Y	569	100.0 %
Y_train	455	79.96 %
Y_test	114	20.04 %

Tabla 2: Separación de los datos de salida

Clase	**Total**	**Training**	**Test**
B	357 (0.63)	285 (0.63)	72 (0.63)
M	212 (0.37)	170 (0.37)	42 (0.37)

Tabla 3: Número de ejemplos y proporción de las clases en los distintos conjuntos de datos

El valor entre paréntesis representa la proporción de esa clase dentro del conjunto de datos.

1.2. Requisitos para la ejecución del código

Para poder ejecuta el código sin problemas es necesario instalar la librería imbalanced-learn, para ello es necesario ejecutar el siguiente comando en el terminal para instalar la librería en Anaconda:

conda install -c conda-forge imbalanced-learn

2. Codificación de los datos

2.1. Datos de entrada

Ninguno de los datos de entrada necesita ser codificado, ya que todas son características numéricas y no categóricas.

2.2. Datos de salida

En cuanto a los datos de salida, en cualquier problema de clasificación es necesario codificar las clases. En el caso de los modelos no lineales no tendremos que hacerlo explícitamente en el código, ya que los algoritmos de aprendizaje de la librería scikit-learn que vamos a usar para este problema, internamente se encargan de codificar las clases.

Para comprobar que realmente estos algoritmos realizan internamente esta codificación, hemos comprobado experimentalmente que los resultados obtenidos con los valores por defecto y los valores codificados para las clases son los mismos. Para esta prueba hemos codificado las clases con las etiquetas -1 y 1.

Para realizar las comprobaciones utilizamos la técnica cross-validation para garantizar que los resultados obtenidos son independientes de la partición que hayamos hecho de training y test.

En nuestro caso hemos usado la variante 10-fold cross-validation, que consiste en dividir el conjunto de training en 10 partes iguales, y realizar 10 iteraciones que consisten en entrenar con 9 partes y validar el entrenamiento con la parte restante, usando en cada iteración una parte distinta para realizar la validación. Para calcular el error de cross-validation se hace la media de los resultados obtenidos en la validación en cada iteración.

En las funciones usadas en el experimento se han dejado los parámetros por defecto, ya que lo que interesa es comprobar si cambia el resultado o no al usar la codificación.

	Sin codificación	**Con codificación**
MLP	0.9699760938858301	0.9699760938858301
SVC	0.9651876267748479	0.9651876267748479

Tabla 4: Resultados del experimento con la codificación en los modelos no lineales

Como se puede comprobar los resultados obtenidos en el experimento no sufren ningún cambio al usar o no codificación, por lo que queda claro que estas funciones la realizan internamente.

Pero en el caso de los modelos lineales si hará falta realizar la codificación explícitamente porque la función que usamos no la realiza internamente. Nuevamente para comprobar que no la realiza, hemos realizado el mismo experimento, anteriormente usado con los modelos no lineales; y los resultados son los siguientes:

	Sin codificación	**Con codificación**
RL	0.9570124601564762	0.9670349174152421

Tabla 5: Resultados del experimento con la codificación en el modelo lineal

Como se puede comprobar en este caso si ha habido cambios en los resultados por lo que se puede deducir que hay que realizar la codificación de las clases antes de usar esta función del modelo lineal.

3. Selección de un subconjunto de las variables

En principio disponemos de 30 características, pero en el paper de la base de datos [1] han podido comprobar que con tan sólo 3 características su modelo obtiene un 97 % de accuracy realizando la técnica 10-fold cross-validation. Estas 3 características son: la textura media, la peor área y la peor suavidad.

Esta notable reducción de características podría simplificar en gran medida nuestra clase de funciones.

La siguiente gráfica muestra la visualización 3D de los datos con la reducción de características:

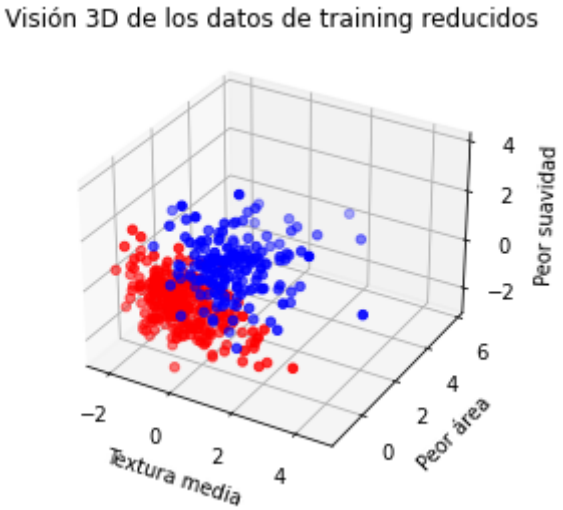

Figura 1: Visualización 3D de los datos de entrada con reducción

Si nos fijamos en la visualización de los datos podemos intuir que es posible utilizar un hiperplano para separar las clases.

Para comprobar que realmente sucede lo que se indica en el paper con los modelos que vamos a utilizar, se ejecutan todos los algoritmos con los mismos valores de los hiperparámetros que se prueban con todas las características, pero utilizando sólo las tres características que se indican en el paper.

Para la comparación volvemos a usar la técnica 10-fold cross-validation para garantizar que los resultados son independientes de la separación de los datos.

Los mejores resultados obtenidos en cada uno de los algoritmos utilizados son los siguientes:

	Todas las variables	**Variables reducidas**
MLP	0.9640321645899739	0.9582113880034772
SVC	0.9769957983193278	0.9559004636337294
RL	0.9717618081715447	0.9693893074471168

Tabla 6: Resultados de balanced accuracy del experimento con la reducción de variables

Observando los resultados obtenidos en la tabla [6] vemos como el utilizar todas la características da modelos con mayor balances accuracy que utilizando sólo las tres características.

Una vez visto los resultados, hemos decidido que es mejor elección entrenar el modelo con todas las características, ya que además de dar mejores resultados nos aseguramos de que al utilizar el modelo para predecir con nuevos datos en general no tengamos problemas con datos más complejos.

4. Necesidad de la normalización de los datos

La normalización de las características es un preprocesado de los datos necesario en algunos algoritmos, dado que mejoran drásticamente los resultados obtenidos. Esta normalización es muy importante en los algoritmos que vamos a usar para resolver el problema. Este es el único preprocesamiento que hemos realizado sobre los datos, aunque implícitamente en las funciones de los algoritmos se realiza otro, que es incluir el término del sesgo en los datos de entrada. Este término se debe incluir para poder entrenar el modelo de forma correcta, pero al realizarlo todas las funciones de los algoritmos que utilizamos, nos podemos olvidar de realizar ese preprocesamiento de los datos de entrada.

En nuestro caso hemos usado la estandarización para normalizar los datos, que consiste en medir la media y desviación estándar a cada característica, restar su media a cada una (se consigue la media 0), y dividir por su desviación estándar (se consigue la varianza 1).

El interés de cada algoritmo que vamos a usar en utilizar esta técnica es el siguiente:

4.1. Regresión logística

La estandarización de las característica es otra de las formas de preprocesamiento que ayudan a la SGD a tener un mejor desempeño pues puede que no restrinja los valores a un rango específico pero los transforma para que todas tengan media 0 y varianza 1.

4.2. Multi Layer Perceptron

El algoritmo MLP es sensible a la escala de las características, por lo que tal y como se indica en la guía de scikit-learn [3] es altamente recomendable escalar los datos. En esta guía se recomienda la estandarización como forma de normalizar los datos, usando la función StandarScaler de la librería scikit-learn. Además se advierte de que tanto los datos de entrenamiento como de test deben estar normalizados, pero sin olvidar que hay que tener cuidado con el data snooping, es decir, no contaminar los datos de validación con los datos de entrenamiento, para hecho estandarizamos los datos de test con la media y varianza extraida de los datos de entrenamiento.

4.3. Support Vector Classification

Al igual que pasa con los dos anteriores algoritmos es sensible a la escala de las características, por lo que tal y como se indica en la guía de scikit-learn [3] es altamente recomendable escalar los datos. En esta guía también se recomienda el uso de la estandarización como forma de escalar los datos, usando la función StandarScaler de la librería scikit-learn. Y al igual que pasa con el algoritmo MLP también se advierte que se debe realizar el escalado de los datos de test.

4.4. Visualización de la estandarización realizada

Para mostrar los resultados provocados por la estandarización se han generado las siguientes siguientes gráficas donde se muestra la media y la varianza de cada característica.

Figura 2: Datos de training sin estandarizar

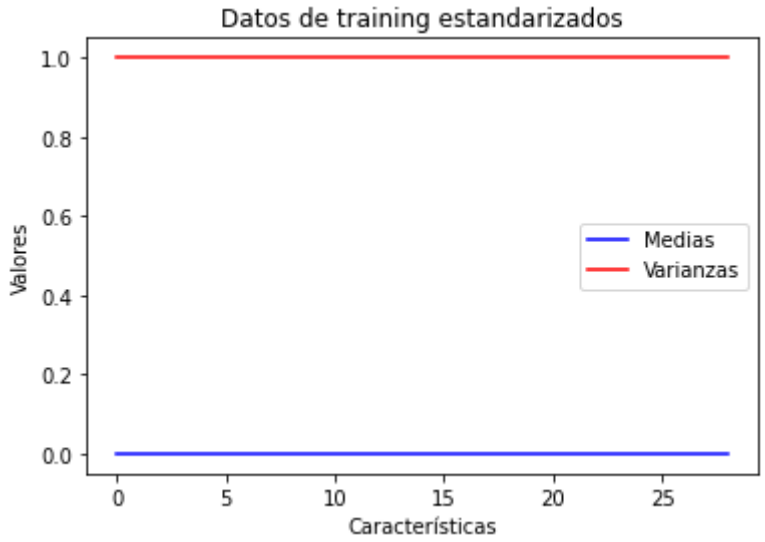

Figura 3: Datos de training estandarizados

Si observamos la figura [2], que tiene una escala logarítmica en el eje X, podemos ver como las distintas características tiene escalas distintas. Tras realizar la estandarización los resultados se pueden ver en la figura [3], donde la media a pasado a ser cero y la varianza uno en todas las características.

Cuando se vayan a usar los datos de test, también se deberán preprocesar, por lo que tendrán que ser estandarizados, pero en este caso no se vuelve a calcular la media y varianza, sino que se utiliza la media y varianza de los datos de training, ya que para el modelo entrenada se han estandarizado los datos con la media y varianza de training, por lo que para no realizar data

snooping y no contaminar los datos se debe hacer de esa forma la estandarización de los datos de test.

5. Funciones de pérdida usadas

Para los distintos algoritmos de aprendizaje usados se utilizan distintas funciones de pérdida.

5.1. Regresión logística

La regresión logística tiene el mismo funcionamiento que la regresión lineal, simplemente que ahora se usa una función de error distinta. En regresión nuestro objetivo era minimizar el error cuadrático medio que existía entre nuestras predicciones y los valores reales de predicción, ahora lo que queremos es conseguir la máxima verosimilitud de la muestra, es decir, que la probabilidad de asignar a una muestra su etiqueta real sea máxima.

$$L(Y|w_1, w_2, \ldots, w_n) = \prod_{n=1}^{N} \prod_{k=1}^{K} (\sigma(w^T x))^{y_{nk}}$$

Para poder usar descenso de gradiente estocástico tenemos que estar ante un problema de minimización, por lo tanto vamos a transformar esta maximización en una minimización. Para ello vamos a usar el logaritmo de la máxima verosimilitud, por lo que los valores pasan del intervalo [0,1] (probabilidades) al intervalo [-∞,0], pues el logaritmo de 0 es -∞ y el de 1 vale 0. Como queremos transformarlo en una minimización, vamos a multiplicar este resultado por -1 quedando finalmente el intervalo [0,∞].

$$E(w_1, w_2, \ldots, w_n) = -lnL(Y|w_1, w_2, \ldots, w_n) = \sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} ln\ t_{nk}$$

5.2. Multi Layer Perceptron

El algoritmo MLP utiliza distintas funciones de pérdida dependiendo del tipo de problema. Para los problemas de clasificación se utiliza como función de pérdida la entropía cruzada.

La pérdida de entropía cruzada calcula una puntuación en base a la probabilidad de la clase predicha y la salida deseada de la clase real. Esta función penaliza dependiendo de lo lejos que está del valor real. El objetivo de esta función es minimizar la pérdida.

Para casos de clasificación binaria, la función se define de la siguiente forma según la guía de scikit-learn [4]:

$$Loss(\hat{y}, y, W) = -y \ln \hat{y} - (1 - y) \ln (1 - \hat{y}) + \alpha ||W||_2^2$$

Donde $\alpha ||W||_2^2$ es el término de la regularización L2 que como se indica penaliza los modelos complejos, y α es un hiperparámetro que controla la intensidad de la regularización.

5.3. Support Vector Classification

El algoritmo SVC utiliza como función de pérdida la función hinge loss, llamada función de pérdida de bisagra. Esta función de pérdida es muy utilizada para entrenar clasificadores, especialemnte el SVM. La función se define de la siguiente forma:

$$Loss(\hat{y}, y, W) = max(0, 1 - y^i(x^i - b))$$

Donde b es el término de sesgo.

El cálculo de la pérdida consiste en obtener la predicción con $y^i(x^i - b)$, para después restarle uno a la predicción, y finalmente quedarme con el mayor valor entre el cero y el resultado de la resta.

6. Modelos seleccionados para la BBDD

La elección de modelos no lineales lo hemos realizado sin observar la base de datos, por lo que hemos elegido aquellos que nos permite conseguir hipótesis lo suficientemente complejas si el problema lo requiere.

6.1. Multi Layer Perceptron

Se ha elegido este algoritmo por las siguientes razones:

- Tiene la capacidad de aprender modelos no lineales lo que le permite ajustar cualquier tipo de muestra sin importar que tan compleja sea esta. La complejidad que pueda alcanzar la red neuronal será definida por la cantidad de capas ocultas y el número de neuronas que haya en cada una de estas.
- Se trata de un algoritmo por lo general bastante costoso computacionalmente, pero al tener una base de datos pequeña se puede usar sin tener grandes tiempos de entrenamiento.
- No se necesita especificar las transformaciones lineales, cosa que no ocurre con los modelos lineales. Con esto conseguimos reducir la cantidad de tiempo en lo que se conoce como feature engineer. La única forma que tenemos de manejar la complejidad de la hipótesis final es con el uso de regularización y ajustando la cantidad de capas y neuronas, que en este caso al ser fijada a 2 capas ocultas, tenemos menos combinaciones que probar.

6.2. Support Vector Machine

Las razones por las que se ha elegido este algoritmo son las siguientes:

- Con este algoritmo se buscan los hiperplanos separadores óptimos, es decir, aquellos hiperplanos cuya distancia con los puntos más cercanos sea máxima, provocando que la clase de funciones tenga menor varianza y por lo tanto sea mejor generalizando.
- Cuando se realiza la fase de entrenamiento no se usan todos los puntos de la muestra, solo los más cercanos a la frontera conocidos como los vectores soporte. Estos condicionarán la hipótesis que aprenderá la svm. Al no usar el conjunto de entrenamiento entero hace que sea eficiente en memoria.
- Es un algoritmo bastante versátil, ya que se puede modificar su comportamiento mediante el ajuste de hiperparámetros.
- Las hipótesis que se pueden llegar a aprender con la svm son bastante complejas, pudiendo llegar a usar infinitas características. Esto es posible con lo que se conoce como truco del kernel. Este método lo que hace es usar las funciones núcleo para expresar las distintas transformaciones que se quieren usar en la hipótesis, consiguiendo así que los cálculos sean independientes de la dimensionalidad de los datos y solo dependan de la cantidad de observaciones.

El modelo lineal seleccionado para ser comparado con los anteriores modelos no lineales es el siguiente:

6.3. Regresión logística estocástica

Las razones por las que hemos seleccionado este algoritmo son las siguientes:

- Es muy eficiente para entrenar modelos lineales de clasificación, pues usa gradiente descendente estocástico para la actualización de los pesos.
- Es un algoritmo fácil de ajustar, ya que tiene pocos hiperparámetros que fijar.
- Tiene un comportamiento bastante bueno cuando se tratan problemas de alta dimensionalidad.

7. Regularización usada

El objetivo principal a la hora de crear un modelo de aprendizaje es conseguir una hipótesis que sea buena generalizando, es decir, una vez tengamos la hipótesis elegida, poder aplicar dicha hipótesis en otros conjuntos de datos y conseguir buenos resultados. Para ello buscamos obtener un buen error de generalización. El gran problema existente en las tareas de aprendizaje es que nuestras observaciones no son perfectas, hay ciertos errores en la toma de medidas conocido como error estocástico. A este error le tenemos que sumar el propio error que introducimos nosotros mismos en la tarea de aprendizaje, el error/ruido determinístico. Este ruido determinísticos se divide en dos partes: sesgo y varianza. El sesgo trata sobre que tan alejada esta nuestra clases de funciones de contener la función f buscada, que en algunos casos es inexistente. Mientras que la varianza mide la variedad de funciones que contiene nuestra clase. Si tenemos una clase con mucha varianza un ligero cambio en los datos producirán cambios muy drásticos en nuestra hipótesis, mientras que si esta es muy simple no se podrá adaptar bien a los datos. Es por esto que surge la regularización. La idea principal es reducir el tamaño de nuestra clase de funciones haciendo uso de una penalización sobre el vector de pesos, con esto se consigue disminuir la varianza de nuestra clase a cambio de sacrificar una cierta cantidad de sesgo que al final nos hará conseguir errores de generalización notoriamente superiores.

Las dos regularizaciones que he considerado para esta práctica son la regularización Ridge y Lasso. La regularización Ridge provoca una reducción proporcional del valor de los coeficientes pero sin llegar a 0, con esto se consigue reducir la influencia de los predictores menos relacionados con la variable respuesta, para ello incluye la suma de los cuadrados de los coeficientes regularizados por un parámetro lambda $E_{in}(w) = E_{in}(w) + \lambda \sum_{j=1}^{n} w_j^2$. Lasso por su parte en vez de usar los coeficientes al cuadrado usa su valor absoluto, consiguiendo que los coeficientes tiendan a 0 y por consecuente anular aquellas características que sean irrelevantes en la tarea de predicción, $E_{in}(w) = E_{in}(w) + \lambda \sum_{j=1}^{n} |w_j|$. La diferencia entre estas dos regularizaciones es que Lasso realiza una selección de predictores mientras que Ridge no excluye a ninguno. En escenarios donde no todos los predictores son necesarios Lasso tiene una mayor ventaja pero si existen correlaciones provoca que sea muy inestable.

En el caso de los modelos lineales tenemos la regresión logística, que de por sí no tiene ninguna regularización asociada, por lo que para ver qué regularización es la mejor, elegiré l1 o l2 según los resultados experimentales.

En el caso de los modelo no lineales tenemos las siguientes elecciones:

- En el caso de MLP es usual usar la regularización ridge, ya que lo que se busca es reducir el tamaño de los pesos para que le sea difícil a la red aprender de datos que son ruido, es decir, que por el efecto de unos pocos datos cambie mucho el aprendizaje; ya que con pesos muy grandes una pequeña modificación en el comportamiento puede hacer que la red aprenda del ruido y el modelo adquiera mucho error.
- En el caso de la SVC lo que busca es encontrar aquel hiperplano cuya distancia a los vectores soporte sea máxima, para ello lo que se tiene que minimizar es la norma de los pesos, es decir, si nos damos cuenta esto no es más que la regularización ridge. El propio modelo tiene una regularización asociada, así que por motivo obvios esta será la usada.

8. Algoritmos de aprendizaje y sus hiperparámetros

Como se ha comentado en el apartado 6 todos los algoritmos de aprendizaje usados en esta práctica pueden ser modificados haciendo uso de sus hiperparámetros. Para encontrar la mejor configuración de estos se ha realizado un estudio mediante 10-fold cross-validation haciendo uso de la función de scikit learn gridSearchCV. Para que se pueda realizar este estudio se define previamente un grid de parámetros que se le pasará a la función junto al predictor a utilizar. Tras haber aplicado cross-validation a todas las configuraciones posibles con los parámetros que se la ha pasado se obtiene el modelo con mejor score. En este caso aquel cuyo balanced accuracy sea más alto. A continuación vamos a exponer las distintas configuraciones estudiadas en la práctica.

8.1. Multi Layer Perceptron

Los distintos parámetros que se pueden modificar para el perceptrón multicapa son los siguientes, cabe destacar que la función se ha usado con la función BalancedBaggingClassifier de la librería imbalanced-learn ya que la implementación de sklearn no soporta clases desbalanceadas y con esta función se corrige:

- hidden_layers_sizes: número de neuronas por capa oculta.
- activation: función de activación usada en cada neurona.
- solver: algortimo usado en la red neuronal, usaremos sgd al igual que se enseño en teoría.
- alpha: parámetro que multiplica el término de regularización, su tamaño indica la intensidad de la regularización aplicada. batch_size: tamaño de las batch usados en el gradiente descendente estocástico.
- learning_rate: tipo de comportamiento del learning rate a lo largo de la ejecución.
- learning_rate_init: valor inicial del learning rate, si es muy grande se tiende a diverger y si es muy pequeño se tarda en converger.
- power t: se usa solo cuando $learningrate = invsclaing$, por lo tanto no lo usamos.
- max_iter: número de épocas máxima que puede completar el objetivo antes de su finalización.
- shuffle: indica si se barajan los datos para hacer el gradiente descendente estocástico.
- random_state: semilla para la generación aleatoria de números.
- tol: tolerancia de la optimización. Cuando se han realizado n iter no change iteraciones sin haber conseguido un cambio mayor a tol, entonces el algoritmo finaliza.
- verbose: indica si queremos que el modelo ofrezca más o menos información.
- warm_start: se indica si se va a iniciar con un solución previa.

- momentum: parámetro que multiplica el término de momentum.
- nevsterovs_momentum: indica si usamos el momementum en el aprendizaje, este se basa en usar los pasos tomados previamente (inercia) para actualizar el vector de pesos. Se suelen conseguir buenos resultados en la práctica.
- early_stopping: indicamos si se usa un porcentaje de la muestra para usarla en el proceso de validación en el early stopping.
- validation_fraction: tamaño del conjunto de validación que se usará para la parada temprana.
- beta1, beta2, epsilon no lo usamos ya que son exclusivos del solver adam.
- n_iter_no_change: número de épocas máximo en la que no se consigue una actualización mayor a la indicada por tol.
- max_fun: no se usa en el solver sgd.

En el guión se nos especifica que el modelo que se use con el perceptron multicapa deberá de tener $3capas = 2ocultas + salida$ y las capas ocultas tendrán que tener un número de neuronas comprendido en el rango 50-100. Es por eso que los valores que hemos elegido para realizar el estudio son los siguientes:

- hidden_layers_sizes: [(100,50,),(50,100,),(50,50,),(100,100,)].
- activation: ['relu'].
- solver: ['sgd'].
- alpha: [0.1,0.01,0.001].
- batch_size: por defecto.
- learning_rate: ['constant'].
- learning_rate_init: [0.1,0.01,0.001].
- power t: no lo usamos pues solo se usa cuando learning_rate = invscaling.
- max_iter: [1000].
- shuffle: [True].
- random_state: [24].
- tol: [10^{-4}].
- verbose: no nos interesa en esta práctica.
- warm_start:[False].

- momentum: [0.9].
- nevsterovs_momentum: [True].
- early_stopping: [True].
- validation_fraction: [0.1].
- n_iter_no_change: [10].
- max_fun: no se usa en el solver sgd.

Con el primer valor expresamos la estructura de nuestra red neuronal fijando así la expresividad que esta puede llegar a tener. El criterio usado ha sido el dictado en el guión por lo que se han usado 2 capas ocultas en las cuales hemos combinado las posibles configuraciones con el mínimo y máximo número de neuronas permitidas por capa.

Con respecto a la función de activación de las capas intermedias hemos elegido la conocida como relu (rectified linear unit) [5] pues es la más ultilizada en los modelos mlp, de hecho es el parámetro por defecto en la librería scikit learn. Esto se debe principalmente a que evita el gran problema de las funciones tanh y sigmoidal, tender a saturar cuando los valores son grandes y como consecuencia no se consigue propagar la información hacia las capas anteriores con el backpropagation. Además no aplica ninguna transformación lineal por lo que su entrenamiento es más rápido, pero como consecuencia se reduce la potencia de expresividad de la red neuronal.

En cuanto al solver usado nos hemos decantado por el algoritmo que se ha visto en clase, el gradiente descendente estocástico, si recordamos este algoritmo es una versión mejorada del clásico batch gradient descent ya que en cada actualización del vector de pesos no se usa la muestra completa si no que se usa parte de esta para evitar que puntos que se contradigan anulen el gradiente provocando que los pesos no se actualicen correctamente.

El learning rate lo dejamos constante pues es el que hemos usado en prácticas anteriores consiguiendo buenos resultados, además es el valor por defecto que trae la propia librería. Con respecto a los valores iniciales del learning rate hemos usado los valores más comunes que se suelen emplear, estos no son muy grandes pues si recordamos valores grandes de learning rate hace que el algoritmo diverja y no pueda encontrar soluciones óptimas debido a que los pasos que toma son demasiado grandes. En el caso contrario si se usan valores muy pequeños, el tiempo que necesita el algoritmo para converger es demasiado alto pues los pasos que va tomando son muy cortos.

El número máximo de iteraciones que hemos usado ha sido el que mejores resultados nos ha dado en prácticas anteriores, pues son suficientes para que el algoritmo converja. El parámetro shuffle se ha fijado a true pues al usar gradiente descente estocástico se necesita barajar las observaciones para obtener los mejores resultados y no siempre recorrer la base de datos en el mismo orden. El nivel de tolerancia lo hemos dejado tal cual recomienda la librería pues es el más utilizado en entrenamientos de redes neuronales. No se usa warm_start ya que no vamos a partir de soluciones anteriores en las fases de training.

Con respecto al momentum, hemos decidido usarlo pues tiene en cuenta pasos anteriores en el aprendizaje permitiendo así salir de óptimos locales y encontrar mejores resultados. El factor

que multiplica a este nuevo componente hemos decidido dejar el recomendado por la librería, pues este es el que más se suele utilizar.

Los últimos parámetros tiene que ver con el criterio de early stopping. Para esta práctica hemos decidido usarlo ya que las redes neuronales tienden a sobreajustar por lo que hacer early stopping es una buena práctica para evitar este gran inconveniente de las redes neuronales. La proporción que se usa para la validación cruzada es la recomendada por la propia librería.

Muchas de las configuraciones que hemos dejado por defecto se debe a que los valores que usa la librería de scikit learn son los valores que mejor funcionan por término general, por lo que para ahorrar tiempo en la selección de modelos hemos decidido no tocar parámetros que suelen ser más generales y manipular aquellos con los que más estamos familiarizados y conocemos cuales son sus efectos en los algoritmos de aprendizaje.

Una vez se han fijado los parámetros simplemente dejamos que gridSearchCV haga el estudio con todas las posibles combinaciones en la muestra con todas las características y en la que hemos dejado solo las características recomendadas por el paper. Tras 185 segundos obtenemos los siguientes resultados:

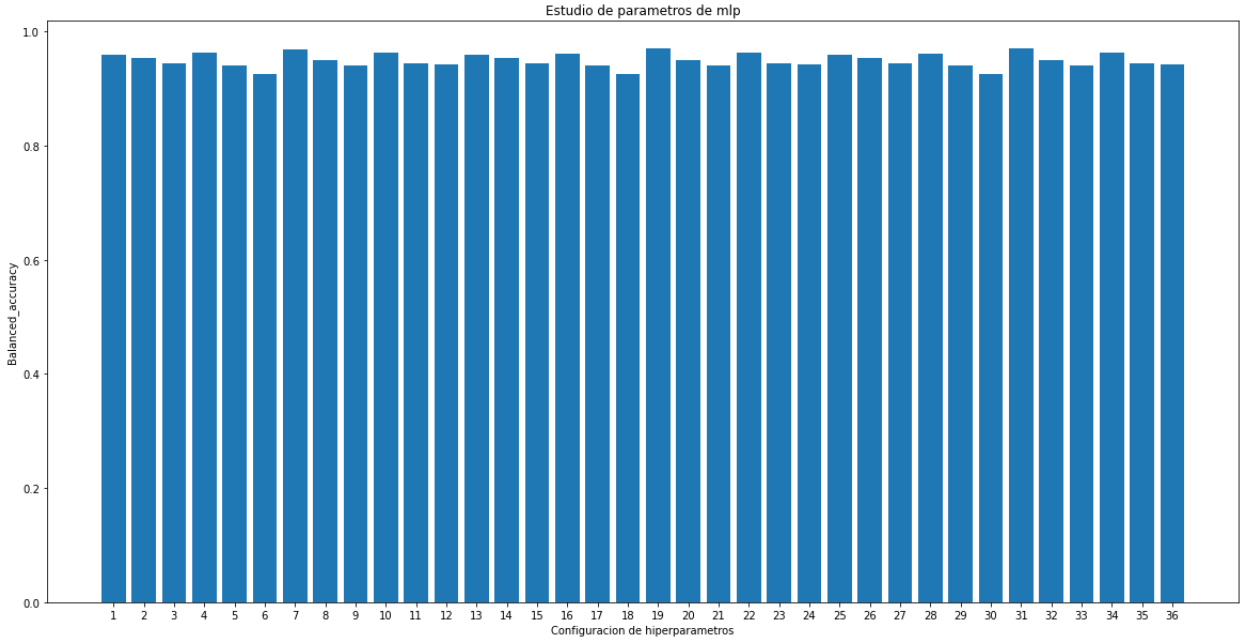

Figura 4: Estudio de parámetros con todas las características

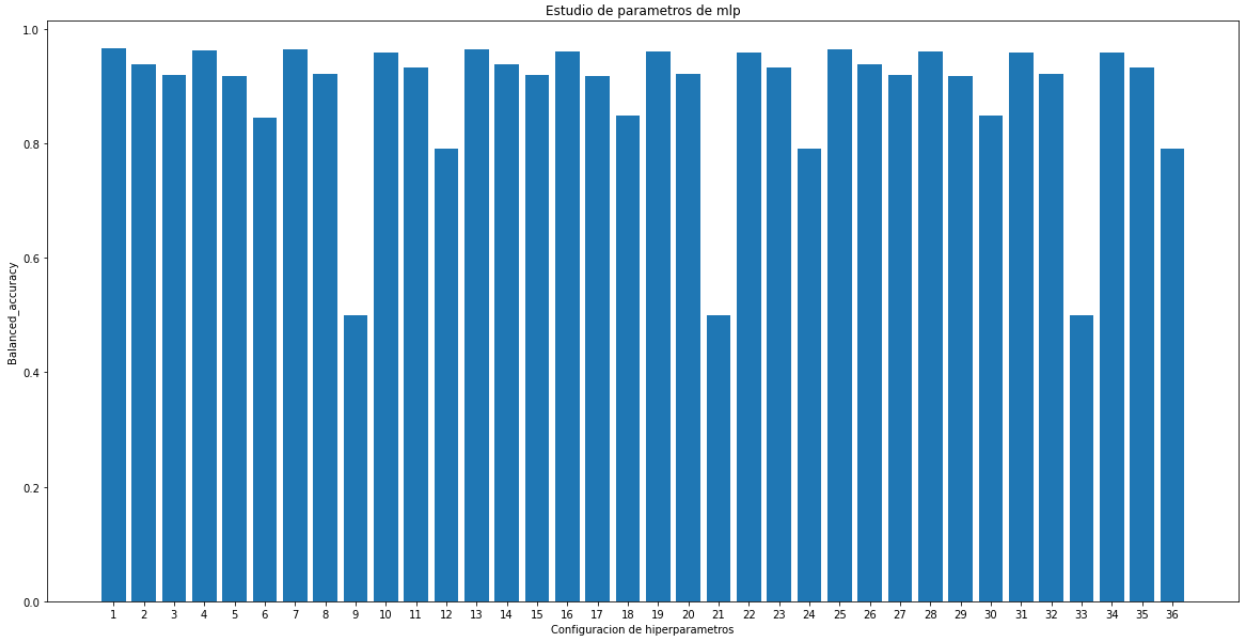

Figura 5: Estudio de parámetros con las características recomendadas por el paper

Vemos que en la gráfica donde se usan todas las características no hay mucha diferencia entre todas las configuraciones, cosa que no ocurre en el segundo estudio en la que se usan solo las características especificadas por el paper. Las mejores configuraciones para cada caso son las siguientes:

Para la muestra con todas la características:

- activation: relu.
- alpha: 0.01.
- early_stopping: True.
- hidden_layer_sizes: (50, 50).
- learning_rate: constant.
- learning_rate_init: 0.1.
- max_iter: 1000.
- momentum: 0.9
- n_iter_no_change: 10.
- nesterovs_momentum: True.
- random_state: 24.

- shuffle: True.
- solver: sgd.
- tol: 0.0001.
- validation_fraction: 0.1.
- warm_start: False.

Consiguiendo un balanced accuracy en validación cruzada de 0.9640321645899739.

Para la muestra con las características reducidas:

- activation: relu.
- alpha: 0.01.
- early_stopping: True.
- hidden_layer_sizes: (100, 50).
- learning_rate: constant.
- learning_rate_init: 0.1.
- max_iter: 1000.
- momentum: 0.9
- n_iter_no_change: 10.
- nesterovs_momentum: True.
- random_state: 24.
- shuffle: True.
- solver: sgd.
- tol: 0.0001.
- validation_fraction: 0.1.
- warm_start: False.

Consiguiendo un balanced accuracy en validación cruzada de 0.9582113880034772.

Por lo tanto el mejor modelo en este caso es aquel que usa todas las características, por lo que será el modelo que usaremos en la comparación final entre los tres algoritmos. Con respecto a los parámetros cabe destacar que la arquitectura usada para la muestra con todas las características es la más simple posible, 50 neuronas en cada capa oculta.

8.2. Support Vector Machine

Los parámetros que se pueden ajustar para el modelo de svm de scikit learn son los siguientes:

- C: constante que multiplica a los errores.
- kernel: kernel que se va a aplicar.
- degree: grados del polinomio usado en el kernel polinomial.
- gamma: el coeficiente para los kernels rbd, polinomial y sigmoidal.
- coef0: termino independiente de la función kernel.
- shrinking: indica si se usa la heurística shrinking.
- probability: indicamos si se quiere hacer el calculo de las probabilidades estimadas.
- tol: tolerancia de nuestro algoritmo, valor a partir del cual se considera que no hay cambios significativos.
- cache_size: se indica la cantidad de memoria cache que usará el algoritmo.
- class_weight: se indica si las clases estan desbalanceadas.
- verbose: opción para que el resultado contenga más o menos detalles.
- max_iter: número máximo de iteraciones de nuestro algoritmo.
- decision_function_shape: forma en la que se realiza la clasificación de las distintas clases, como es clasificación binaria se ignora.
- break_ties: solo se usa en clasificación con más de 2 clases.
- random_state: semilla que se usará en la generación de número aleatorios, solo se usa si se realiza el calculo de probabilidades.

Los valores que hemos estudiado para cada uno de los parámetros los mostramos a continuación:

- C: [2,1]
- kernel: [poly,rbf].
- degree: [2,3,5].
- gamma: [scale,auto].
- coef0: [0,1].
- shrinking: [True,False].

- probability: [False].
- tol: [0.001].
- cache_size: [500].
- class_weight: ['balanced'].
- verbose: por defecto.
- max_iter: [-1].

El primero de los parámetros que podemos ajustar para este modelo es la constante C, si nos acordamos esta constante multiplica los términos de error de nuestro modelo, cuanto más grande es C más pequeños tienen que ser los errores y cuando C es pequeña se pueden producir más errores de clasificación. Con este parámetro simplemente estamos regulando la cantidad de errores que podemos cometer, por lo que hemos elegido dos valores, el 2 y el 1. El 1 es el valor por defecto y el recomendado por la librería por lo que para comparar con otro modelo que ajusta más los datos hemos elegido el valor 2, de esta forma tenemos unos modelos que intentan tener menor Ein que otro. Después del análisis veremos cuales son mejores generalizando.

Con respecto al kernel se han estudiado los dos propuestos en el guión, tanto el kernel polinomial como el kernel Gaussiano. Ambos permiten aprender una hipotésis no lineal que pueda ajustarse bien a los datos. El kernel polinómico permitirá aprender hipótesis cuyo grado será el indicado en el parámetro degree, mientras que el kernel rbf podrá aprender funciones tan complejas como desee pues su espacio de características es infinito.

Otro de los parámetros más importantes usados en las svm es el gamma. Este valor marca el rango de influencia de cada dato. Si es alto el alcance es menor y si es bajo el alcance será mayor. Por lo tanto para usar un gamma que se ajuste a nuestro problema hemos estudiado dos configuraciones, la auto y la scale. La auto es la inversa del número de características mientras que scale es igual a $\frac{1}{n\_features * X.var()}$. De esta forma gamma será proporcional a nuestro problema.

En cuanto al coef0 los valores que hemos estudiado han sido el 0 y el 1. Este valor es simplemente el término independiente del polinomio usado en el kernel polinomial. Con respecto al motivo que hemos elegido estos valores es simplemente porque el por defecto y recomendado por sklearn es 0 mientras que el visto en clase usaba el valor 1. Veremos tras el estudio cual es el que mejor se adapta a nuestro problema.

El parámetro shrinking indica si queremos usar la heurística shiriking o no. Esta heurística lo que hace es acortar el tiempo de entrenamiento. En la documentación de sklearn nos indica que esta puede funcionar o no dependiendo del problema, es por eso que al no ser una respuesta seguro probamos los valores True y False y veremos cuales ofrecen mejores resultados.

Como para este problema no vamos a calcular la probabilidad de pertenecer a cada clase el parámetro probability lo dejaremos a False. En la tolerancia usada será la misma que en los otros modelos pues es la recomendada y suele dar buenos resultados.

La cantidad de cache la hemos fijado realizando pruebas de velocidad de ejecución ya que su modificación no afecta a la calidad de las respuestas solo a los tiempo de ejecución, es por eso que tras varios experimentos hemos decidido fijarla a 500 MB.

Al estar en un problema en donde las clases estan desbalanceadas es oblgatorio usar el parámetro class_weiht y fijarlo a balanced para que en la fase de aprendizaje se intente clasificar bien cada clase, para ello la función asignará un peso a cada una de las clases en la función de error que use para compensar la diferencia de ejemplos de una clase con la otra.

Hemos decidido no poner límite de ejecuciones al algoritmo para no reducir la potencialidad del algoritmo de encontrar el mejor separador posible. En cuanto a los otros parámetros, no hemos especificado sus valores ya que no afectan al estudio que hemos realizado.

Una vez explicado todos los parámetros mostramos los resultados obtenidos:

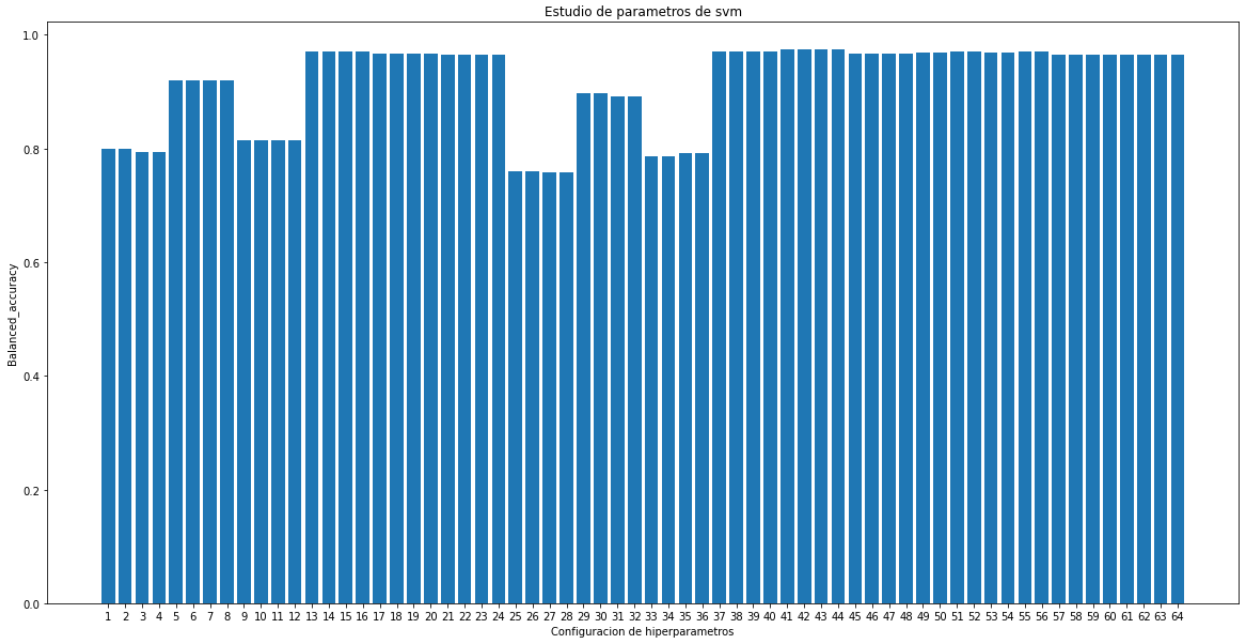

Figura 6: Estudio de parámetros con todas las características

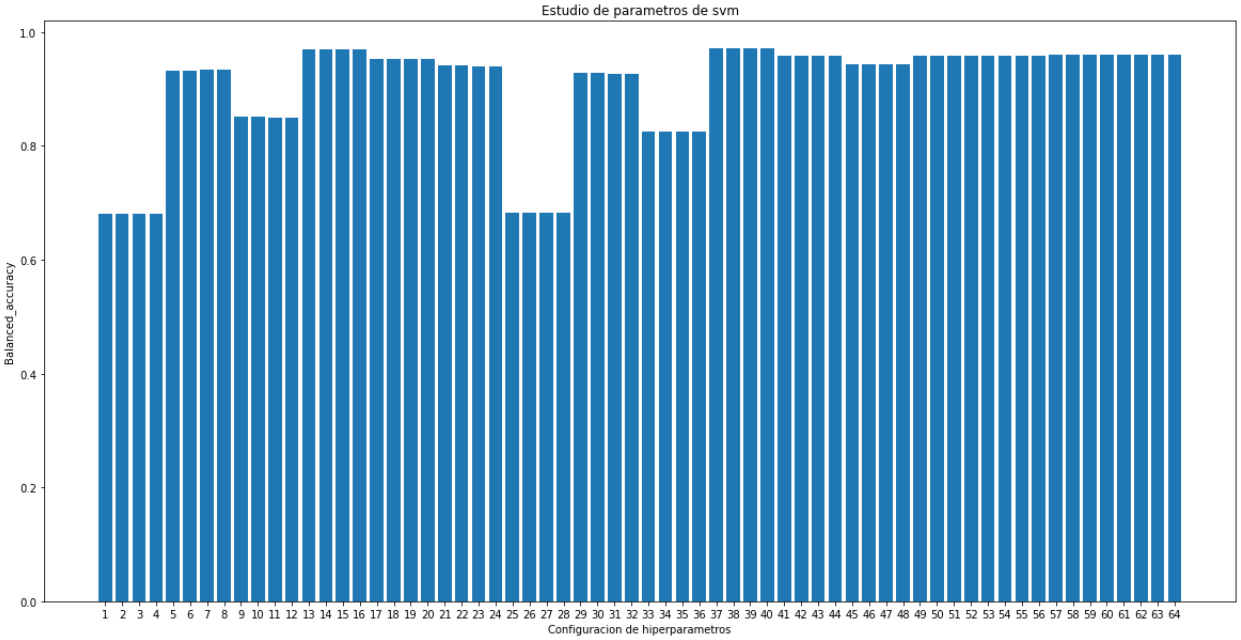

Figura 7: Estudio de parámetros con las características recomendadas por el paper

El mejor modelo entrenado en la muestra con todas las características consigue una balanced accuracy de 0.9769957983193278, mientas que el que ha sido entrenado en la muestra con menos características ha conseguido una balanced accuracy de 0.9559004636337294. Por lo tanto podemos concluir que para este modelo es mejor usar todas las característica al igual que ocurría en las redes neuronales. En este caso para ambas muestras el mejor modelo tiene los siguientes parámetros:

- C: [1]
- kernel: [poly].
- degree: [2].
- gamma: [scale].
- coef0: [1].
- shrinking: [True].
- probability: [False]
- tol: [0.001].
- cache_size: [500].
- class_weight: ['balanced'].
- verbose: por defecto.

- max_iter: [-1].

Como podemos ver el que mejor desempeño da es el kernel polinomial de grado 2, por lo que podemos intuir que los datos no son muy complicados de separar. Además vemos que la heurística shrinking es útil en este problema pues ofrece buenos resultados y consigue resultados rápidos. Con respecto a la constante C el valor final a usar es el 1, es decir, que para este problema es mejor tener un hiperplano que permite clasificar erróneamente una cierta cantidad de puntos con el objetivo de poder generalizar mejor.

8.3. Regresión Logística Estocástica

Los parámetros que se pueden ajustar para este modelo son los siguientes:

- loss: función de error a usar.
- penalty: función de regularización a usar, l1 es la regularización Lasso y l2 la ridge.
- alpha: constante que multiplica al término de regularización, indicamos si queremos una regularización más o menos intensa.
- fit_intercept: se indica si se quiere calcular el término de sesgo.
- max_iter: número máximo de iteraciones.
- shuffle: indicamos que en cada época se barajen las observaciones para no recorrerlas siempre en el mismo orden
- n_jobs: indica el número de CPU's a usar.
- random_state: fijamos la semilla de ejecución, para que siempre obtengamos los mismos resultados entre ejecución y ejecución.
- learning_rate: comportamiento del learning rate durante la ejecución.
- eta0: valores de learning rate a probar, cuanto más grande sea este más grande serán los pasos que se den en cada actualización de los datos, nos puede llevar a diverger, si eta0 es más pequeño entonces los pasos son más cortos y se tarda más en converger.
- early_stopping: indicamos si queremos usar early stopping o no.
- class_weight: en esta variable expresamos si las clases se encuentran balanceadas.
- warm_start: indicamos que si queremos inicializar el vector de pesos con una solución anterior.
- average: se indica si se quiere devolver como vector de pesos la media del historial.

Los valores que se han estudiado son los siguientes:

- loss: [log].
- penalty:[l1,l2].
- alpha:[0.1,0.01,0.001].
- fit_intercept: [True].
- max_iter: [10**3,10**4].
- shuffle: [True].
- n_jobs: [1], por defecto.
- random_state: [24].
- learning_rate: [constant].
- eta0: [0.1,0.01,0.001].
- early_stopping: [True,False].
- class_weight: [balanced].
- warm_start: [False].
- average: [False].

La función de error al estar resolviendo por el algoritmo de la regresión logística es la función 'log'. Con respecto a las penalizaciones hemos probado tanto la Lasso como la Ridge para ver cual de las dos se comporta mejor para este problema. Con respecto al valor de alpha al igual que en la SVM hemos probado los valores más típicos entre 0.1 y 0.001 para ver cual de estos es el más adecuado para nuestro problema.

Con respecto al parámetro fit intercept lo dejamos a true ya que queremos que se calcule el término de sesgo. Para el número máximo de iteraciones hemos elegido dos valores 1000 (como en los anteriores modelos) y 10000, para ver si un aumento de iteraciones puede mejorar nuestro modelo.

El parámetro shuffle lo hemos dejado como es lógico a True pues estamos usando el método de descenso del gradiente estocástico por lo que es necesario barajar las observaciones con el objetivo de conseguir buenos resultados. Asociado a este parámetro está la semilla de generación de números aleatorios el cual hemos dejado a 24. El números de CPU a usar los hemos dejado por defecto pues este algoritmo es rápido y para el tamaño de la base de datos usada hemos considerado que no es necesario darle más recursos.

Los valores de eta0 que hemos usado para el estudio son los mismos que en el caso de la SVM, pues son los valores más comunes que suele tener los learning rate por lo que hemos decidido probar con estos tres valores y ver cual rinde mejor.

Con respecto al early stopping a diferencia de los otros modelos este al solo ajustar un modelo lineal no tiene tanta capacidad de sobreajustar, de hecho dependiendo de la complejidad

del problema puede tender a no ajustar lo suficiente por lo que se produce underfitting. Como no sabemos si va a ser necesario para nuestro problema hemos decidido probar los valores True y False y ver que resultados obtenemos.

Finalmente al igual que en los otro modelos necesitamos fijar el parámetro de class_weight a balanced para que el algoritmo tenga en cuenta que no hay la misma cantidad de observaciones de una clase que de otra, por lo que tendrá que aplicarle una ponderación a cada una. El parámetro de warm_start no lo usamos ya que no queremos partir de soluciones anteriores y como no queremos que los pesos devueltos sean la media del historial fijamos average a False.

Explicado los parámetros fijados pasamos a mostrar los resultados del modelo para la muestra con todas las características y la muestra con las características reducidas:

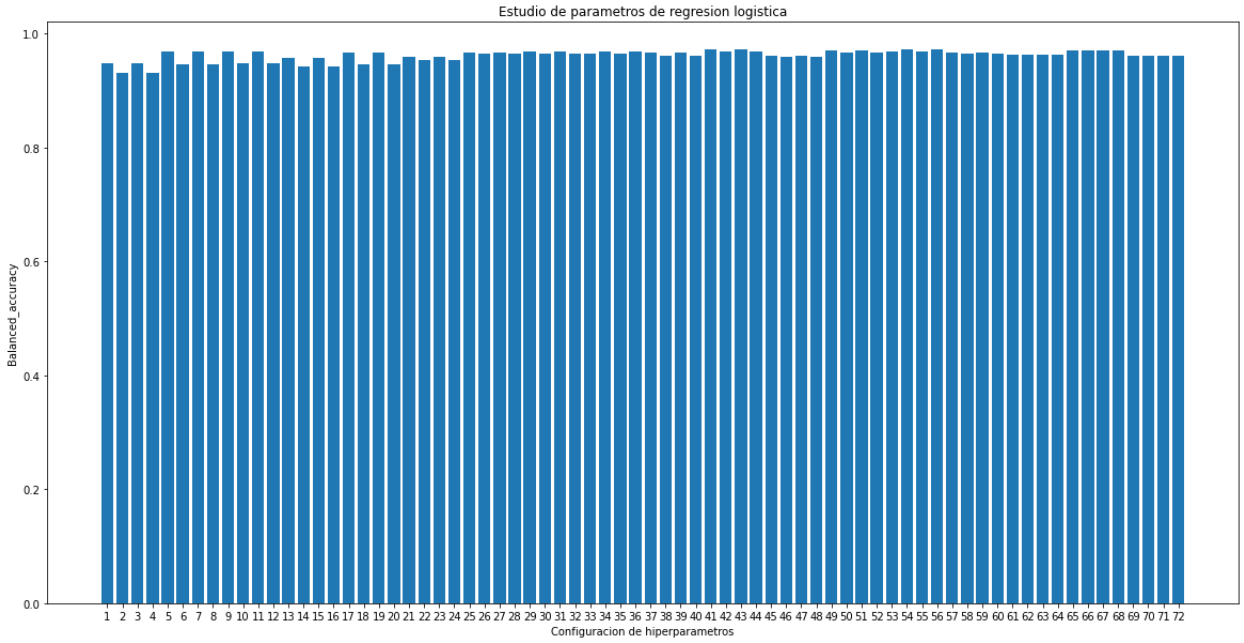

Figura 8: Estudio de parámetros con todas las características

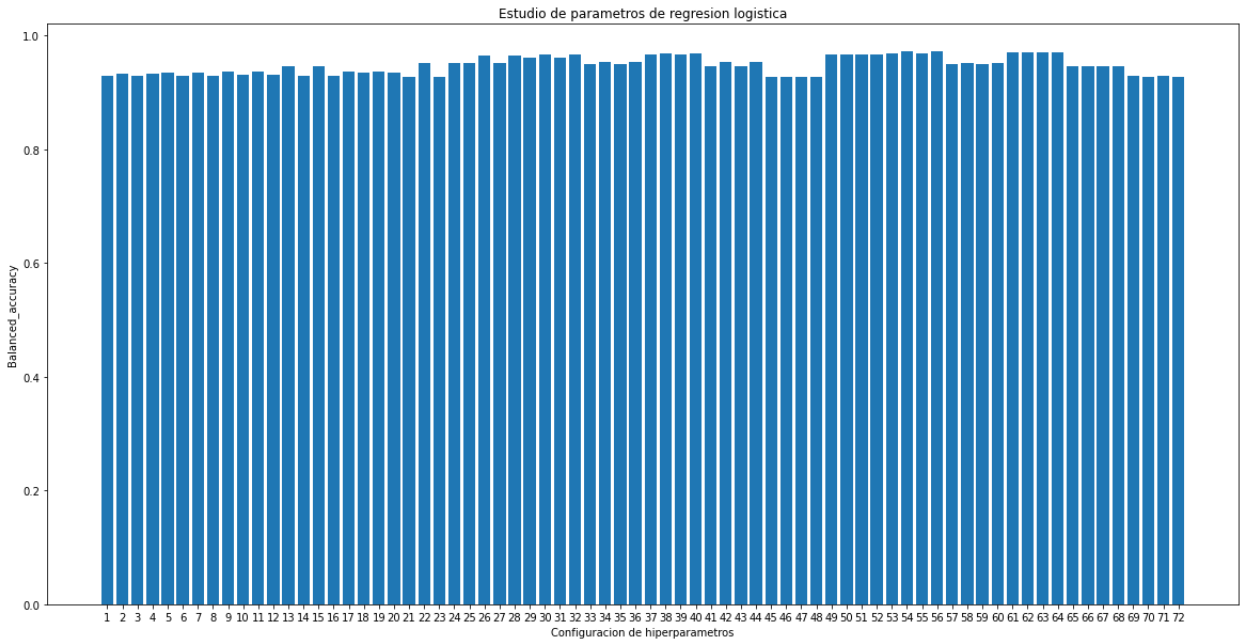

Figura 9: Estudio de parámetros con las características recomendadas por el paper

El modelo entrenado en la muestra con todas las características ha conseguido un error de validación de 0.9717618081715447 mientras que el que se ha entrando en la muestra con características reducidas se ha obtenido un error de validación de 0.9693893074471168. Por lo que como ocurría en los anteriores modelos este modelo tiene un mejor desempeño cuando se entrena con todas las características. El mejor modelo para ambas muestras tiene los siguientes parámetros:

- loss: [log].
- penalty:[l1].
- alpha:[0.001].
- fit_intercept: [True].
- max_iter: [10**3].
- shuffle: [True].
- n_jobs: [1].
- random_state: [24].
- learning_rate: [constant].
- eta0: [0.01].
- early_stopping: [False].

- class_weight: [balanced].
- warm_start: [False].
- average: [False].

Para este problema con la regresión logística la mejor regularización que se puede usar es la Lasso que si recordamos lo que hace es dejar a 0 las características más irrelavantes para la tarea de predicción. Con respecto al alpha y eta0 tienen los valores 0.001 y 0.01 respectivamente lo que indica que se usa la regularización más permisiva de las que le hemos pasado y la distancia de los pasos que toma es un 1 % el valor del gradiente.

Con respecto al early stopping vemos que no es necesario para este problema puesto que el modelo no tiene suficiente expresividad como para poder sobreajustar con los datos de entrenamiento que tenemos.

9. Selección de la mejor hipótesis

El criterio que hemos usado para elegir el mejor modelo ha sido aquel que tenga el menor error de validación, pues nuestro objetivo es encontrar el modelo que mejor generalice, es decir, que para datos que no ha visto nunca obtenga un número muy reducido de fallos, siendo en el mejor de los casos 0. Es por eso que el método que hemos llevado a cabo para la elección tanto de los mejores parámetros como para hacer la comparación de modelos es la validación cruzada, ya que con el Eval estamos realizando una estimación de este error fuera de la muestra. Este método hay que usarlo con cuidado porque el Eval que calculamos para estimar Eout es un estimador optimista. No lo podemos usar como cota sobre el error de generalización de nuestro modelo ya que se encuentra contaminado debido a su uso durante el proceso de selección.

Para la estimación de estos Eval hemos usado el método de 10-fold cross-validation, el cual consiste en dividir la muestra de entrenamiento en 10 trozos, 9 lo usaremos para entrenar y usaremos 1 no empleado durante el entrenamiento para estimar el Eval. Este proceso se repetirá 10 veces y el Eval final será la media de los Eval obtenidos en las 10 ejecuciones.

Los resultados obtenidos son los siguientes:

Modelo	**Balanced accuracy**
MLP sin reducción	0.9640321645899739
MLP con reducción	0.9582113880034772
SVM sin reducción	0.9769957983193278
SVM con reducción	0.9559004636337294
RL sin reducción	0.9717618081715447
RL con reducción	0.9693893074471168

Como podemos ver finalmente los modelos que mejores Eval tienen son aquellos que usan todas las características, de estos los mejores modelos son por orden: SVM, RL y MLP. Efectivamente como habíamos visto en teoría uno de los mejores modelos para clasificación son las SVM y en este problema se sigue cumpliendo. Cabe destacar que el kernel usado es el polinomial de grado 2 por lo que la separación que debe de existir entre las dos clases no debe de ser muy compleja. Otro punto que nos confirma esto es que el segundo mejor modelo de la práctica es la regresión logística la cual recordemos ajusta un modelo lineal que tiene mucha menos expresividad que modelos no lineales. En tercer lugar nos queda el perceptron multicapa aunque no muy lejos de los otros dos modelos. Esto es debido a que es un modelo que tiende a sobreajustar debido a su gran expresividad, por lo que en problemas más simples como el nuestro puede conseguir buenos resultados, pero debido a la complejidad de las clases de funciones su generalización es mucho que peor que modelos más simples, como en el caso de la regresión logística.

Elegido ya nuestro mejor modelo vamos a calcular la estimación del Eout conseguido, para ello primero entrenamos nuestro modelo con toda la muestra de training y después calculamos el Etest que tiene. Tras esto aplicaremos el término de la desigualdad de Hoeffding que penaliza la complejidad de las clases de funciones para finalmente calcular la cota de Eout de nuestro modelo. Al usar una hipótesis prefijada, elegida en el proceso de training, el número de funciones de nuestra clase tendrá solo 1 elemento, por lo que la expresión de nuestra cota quedará así:

$E_{out}(g^*) \leq E_{test}(g^*) + \sqrt{\frac{1}{2N} \ln \frac{2}{0.05}}$. Con esta desigualdad podemos decir que el error de generalización de nuestro modelo es menor igual a $\sqrt{\frac{1}{2N} \ln \frac{2}{0.05}}$ con un 0.95 % de confianza. Tenemos que acordarnos que la métrica que usamos nos dice el porcentaje de datos bien clasificados por lo que tenemos que quedarnos con el contrario 1- Etest para ver el porcentaje de datos mal clasificado y entonces aplicar la desigualdad, si no hacemos esto estaríamos diciendo que se tiene una mayor precisión cuando se generaliza cosa que no tiene mucho sentido.

Aplicado lo anterior obtenemos el siguiente resultado: $E_{out}(g^*) \leq 0.13910262293778755$ por lo que nuestro modelo clasificará mal el 13 % de los datos fuera de la muestra. O lo que es lo mismo la precisión del modelo será mayor o igual a 86 %. Y todo esto lo confirmamos con un nivel de confianza del 95 %.

Visto los resultados comprobamos que efectivamente los errores calculados durante el periodo de validación eran optimistas como dijimos anteriormente.

10. Valoración de los resultados

10.1. Métrica de error

Al encontrarnos en un problema de clasificación binaria donde las clases están desbalanceadas, la métrica que he decidido usar es la balanced accuracy. Esta medida nos indica el porcentaje de clasificaciones correctas que hemos realizado. Tiene la siguiente fórmula:

$$BalancedAccuracy = \frac{Sensitivity + Specificity}{2}$$

La balanced accuracy se calcula como la media entre la sensitivity y la specificity. La sensitivity, conocida como la tasa de los verdaderos positivos o recuperación; es la métrica que mide los casos positivos que detecté correctamente. La specificity, conocida como la tasa de verdaderos negativos; es la métrica que mide los casos negativos que detecté correctamente.

Ahora bien, cuando vaya a calcular las cotas no se puede usar la balanced accuracy ya que esta en sí no es una medida de error como tal, se puede usar para comparar modelos, pero para la estimación teórica se necesita el porcentaje de datos mal clasificados, por lo que a la hora de calcular las cotas al mejor balanced accuracy que hemos conseguido se lo restamos a 1 para conseguir el porcentaje mal clasificado y poder calcular así las cotas correctamente. Si usasemos el balanced accuracy entonces al sumar los términos que penalizan la complejidad de la clase de funciones estaríamos mejorando el balanced accuracy cuando se supone que es una penalización.

10.2. Matriz de confusión

La matriz de confusión es una matriz donde se relacionan los valores verdaderos con los valores predichos. Esta matriz nos permite ver si el clasificador se está confundiendo al predecir ciertas clases. Con los valores de esta matriz se puede calcular el valor de ciertas métricas como accuracy, specificity, etc.

La matriz de confusión obtenida con nuestra mejor hipótesis es la siguiente:

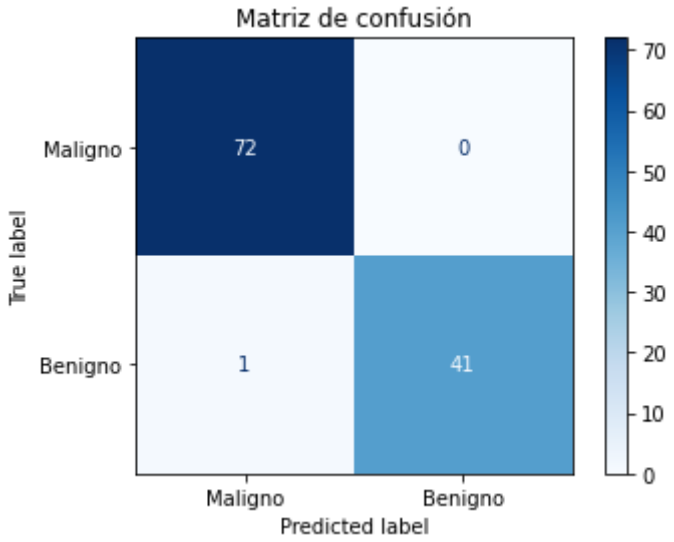

Figura 10: Matriz de confusión de la mejor hipótesis

Observando los valores de la matriz de la figura [10] vemos como el clasificador distingue perfectamente las dos clases que hay en el problema, teniendo un ratio de falsos positivos y falsos negativos diminuto.

Dado que los datos están desbalanceados, nos hemos asegurado de que el algoritmo lo tuviese en cuenta, ya que si utilizamos un clasificador con datos desbalanceados y no lo solucionamos, vamos a tener un clasificador con mucho sesgo, y puede ocurrir que ese clasificador nos de unos resultados muy buenos, pero al estar sesgados no podemos confiar en que sea un buen modelo, ya que si hay una proporción muy alta de una clase respecto de la otra, el modelo al aprender de los datos desbalanceados normalmente dará como predicción la clases con más proporción y en la matriz de confusión habrá poco ratio de falsos negativos o falsos positivos, ya que en su mayoría los datos son de una clase y justamente esa clase es la que predice el modelo, por lo que en principio veremos que es un buen clasificador, pero realmente no lo es, ya que el modelo tendrá un enorme error de sesgo, que se verá reflejado al obtener el error fuera de la muestra.

10.3. Curva de ROC

La curva de ROC es una gráfica que representa al ratio de verdaderos positivos frente al ratio de falsos positivos. La curva de ROC nos permite comparar distintos clasificadores.

Para saber como de bueno es un clasificador se tiene en cuenta el área que forma la curva del clasificador respecto de la diagonal que va desde el punto (0,0) hasta el puntos (1,1). Esta diagonal representa al clasificador aleatorio, que es el clasificador que peores resultados nos puede dar, por lo tanto nuestro modelo debe estar siempre por encima de este clasificador para que empiece a ser bueno, ya que si está por debajo de la diagonal, quiere decir que es mejor opción elegir el clasificador aleatorio.

Dado que la calidad de un clasificador se mide por el área que forma con la diagonal, el

mejor clasificador será aquel que cuando se tenga un ratio de falsos positivos igual a 0 ,tenga un ratio de verdaderos positivos igual a 1, y que este ratio de verdaderos positivos se mantenga para todos los valores del ratio de falsos negativos.

La curva de ROC obtenida con nuestra mejor hipótesis es la siguiente:

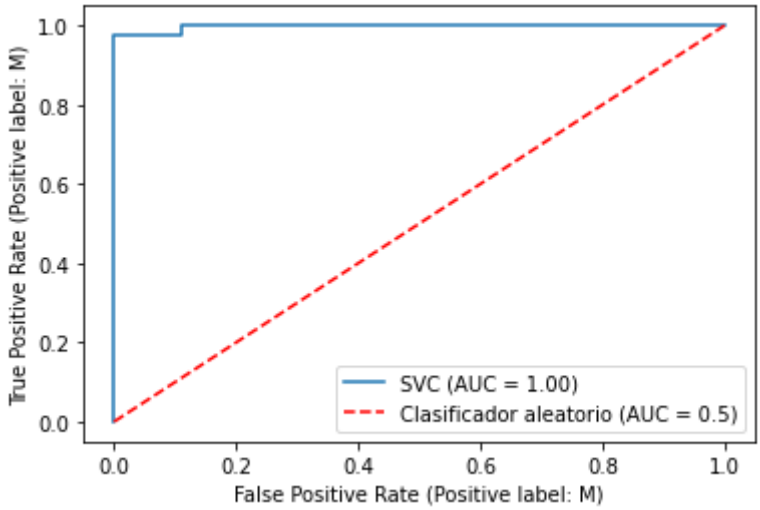

Figura 11: Curva de ROC de la mejor hipótesis

Como se puede ver en la figura [11], la mejor hipótesis tiene un área cercana a 1, por lo tanto es un clasificador casi perfecto, y mucho mejor que el clasificador aleatorio, que lo que hace es predecir la clase de forma aleatoria.

11. Argumentación de la mejor hipótesis

LLegamos al final de la práctica y nos toca justificar por qué todo el proceso que hemos seguido para la selección de modelos es el adecuado y el que nos ha permitido elegir el modelo que mejor generaliza. El criterio seguido para la selección es el conocido como SRM (structural risk minimization), este a diferencia del seguido en prácticas anteriores (ERM) tiene como objetivo no solo disminuir el error dentro de la muestra si no que también busca reducir la complejidad de la clase de funciones. Esto se debe principalmente a que al usar modelos no lineales nuestras clases empiezan a tener complejidades muchos más altas, incluso infinitas, por lo que tenemos que de alguna forma reducir esta complejidad añadida. Es por eso que usamos la regularización con el objetivo de disminuir esta penalización, reduciendo la varianza de nuestro modelo a costa de ganar un poco de sesgo. El resultado que se obtiene por lo tanto es el mejor generalizador posible.

Este proceso lo hemos seguido haciendo uso de la validación cruzada. La fase de entrenar cada configuración del modelo lo que hace es minimizar el error dentro de la muestra, con esto se minimiza la parte de la cota asociada al error dentro de la muestra E_{in}. Es en el proceso de la comparación de errores de validación donde seleccionamos la regularización a usar y la mejor combinación de parámetros donde se hace el proceso de minimización de la complejidad de la clase de funciones. Pues estamos eligiendo aquel que consigue menor error de E_{val}, que es nuestra estimación optimista de E_{out}. Al fin y al cabo con esto estamos eligiendo el que mejor generaliza, es decir, aquel cuyo equilibrio entre el error dentro de la muestra y penalización de la complejidad de la clase de funciones sea óptimo.

Es por ello que la selección por validación cruzada nos garantiza que aquel que minimice el E_{val} será el mejor modelo posible de los estudiados.

Referencias

[1] W. Nick Street, W. H. Wolberg y O. L. Mangasarian. "Nuclear feature extraction for breast tumor diagnosis". En: 1905 (1993). Ed. por Raj S. Acharya y Dmitry B. Goldgof, págs. 861 -870. URL: https://doi.org/10.1117/12.148698.

[2] *Repositorio UCI del problema.* URL: http://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(diagnostic).

[3] *Tips de la guía de Scikit-learn: MLP.* URL: https://scikit-learn.org/stable/modules/neural_networks_supervised.html#tips-on-practical-use.

[4] *Función de pérdida de la guía de Scikit-learn: MLP.* URL: https://scikit-learn.org/stable/modules/neural_networks_supervised.html#mathematical-formulation.

[5] *How to Choose an Activation Function for Deep Learning.* URL: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/.