Alumno/a: Lucas Manuel Pasquevich
Materia: Herramientas Computacionales Para Científicos
Curso: 2022

Trabajo Final

Clasificación de estrellas, galaxias y cuásares usando algoritmos de machine learning.

1. Introducción

En el presente trabajo se dsarrollarán algoritmos de machine learning, los parámetros para medir su performance y el peso en la predicción de los parámetros dados para la clasificación efectiva de fuentes luminosas basados en datos espectroscópicos, fotométricos y el redshift de cada fuente. En el mismo, evaluaremos el costo computacional de cada algoritmo pesando ventajas y desventajas de los mismos, para luego ver su desempeño en la clasificación.

2. Los Datos

Tenemos 10000 datos de fuentes luminosas, brindadas por SDSS (Sloan Digital Sky Survey). Tenemos 17 datos de las mismas, además de la columna "$Class$" que nos dice si es una galaxia, estrella o cuásar. Las otras 17 features se pueden dividir en tres grupos: Posicionales: Tenemos "$objid$" que es id identificador de la fuente, además de "ra" y "dec" que son las coordenadas de aserción recta y declinación. Espectroscópicas: Tenemos las features "r", "g", "u", "i", "z" que son las magnitudes de las fuentes en las 5 bandas del telescopio. Fotométricas: Las columnas "run", "$rerun$", "$camcol$" y "$field$" se refieren al campo del cielo donde SDSS tomo la imagen, un çampoçorrespondería a 2048x1489 píxeles. Los mismos identifican el escáner usado (run), el tipo de procesamiento de la imagen (rerun), identificador del 1 al 6 de la linea de exploración dentro del run (camcol). Complementarias: La columna "$specobjid$" es un identificador espectroscópico de la fuente, "$class$" nos indica que clase de objeto es (Star, Galaxy o QSO), el "$redshift$" de cada fuente, "$plate$" que es el numero de plato del SDSS, "$fiberid$" que es

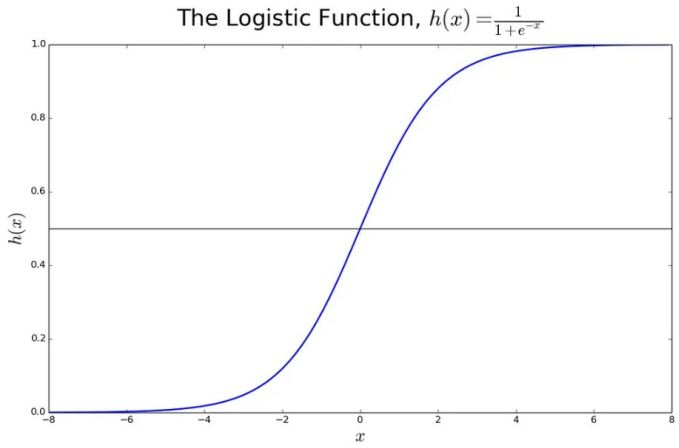

Figura 1: Representación gráfica de la función logística

el identificador de la fibra, y "mjd" que es la fecha de observación en días julianos.

Para la aplicación de los algoritmos, omitiremos las features de identificación, siendo irrelevantes para la clasificación de los objetos, además de la omisión de "$rerun$".

3. Algoritmos

Para la clasificación de las fuentes, vamos a hacer uso de dos diferentes tipos de algoritmo que difieren significativamente en su funcionamiento y complejidad.

3.1. Regresión Logística

El modelo de regresión logística trabaja de manera similar a la de una regresión lineal, asignando pesos a las features. La función logística es una función sigmoide (o Función de Respuesta Logística), la cual devuelve un valor entre 0 y 1.

$$\sigma(t) = \frac{1}{1 + e^{-t}} \tag{1}$$

La función logística calcula probabilidades donde, si un cierto elemento de nuestros datos supera cierto umbral, es asignado a una clase o a otra. Las funciones clave para esta regresión es la *Función de Respuesta Logística* y la *Función Logit*, en las cuales asignamos la probabilidad (que esta en

una escala de 0 a 1) a una escala más amplia congruente con el modelado lineal. Para que nuestra función exponencial no figure en el denominador consideramos "*oportunidades*" o *Odds* en lugar de probabilidades, estas *Odds* son la proporción de "*exitos*" (1) o "*noexitos*" (0).

$$Odds(Y = 1) = \frac{p}{1 - p} \tag{2}$$

Podemos obtener la probabilidad a partir de las oportunidades usando la función de oportunidades inversa

$$p = \frac{Odds}{1 + Odds} \tag{3}$$

Combinando lo anterior con la *Función de Respuesta Logística* obtenemos:

$$Odds(Y = 1) = e^{\beta 0 + \beta 1 x 1 + \beta 2 x 2 + ... + \beta n x n} \tag{4}$$

De esta forma, podemos definir la *Función Logit* como:

$$log(Odds(Y = 1)) = \beta 0 + \beta 1 x 1 + \beta 2 x 2 + ... + \beta n x n \tag{5}$$

Esta función, asigna la probabilidad de (0,1) a cualquier valor real. De esta forma, utilizamos un modelo lineal (la regresión lineal) para pronosticar una probabilidad que se puede asignar a una etiqueta o clase con alguna regla de corte. En la Fig. 1 podemos ver una representación gráfica de la función logística.

3.2. XGBoost

3.2.1. Modelos de *Árboles*

Los *Arboles de decisión* o CART (*Classification and Regression Trees*), son un algoritmo de clasificación y regresión de uso generalizado desarrollado por Leo Breiman y cia. en 1984. Los mismos son la base para el desarrollo de modelos de ensamble mucho más complejos que aprovechan las ventajas y aplacan las desventajas de los mismos (suelen sobre-ajustarse a los datos y dejar de aprender de los mismos para pasar a memorizarlos), como *Random Forest*, *Boosted Trees*, *Gradient Boosting*, entre otros. Un modelo de *Árbol* es un conjunto de reglas de "$si-entonces-sino$" que son sencillas de entender e implementar (Vemos en la Fig. 2 un esquema de los mismos). A diferencia de la *Regresión Logística* los modelos de *Árboles* tienen la capacidad de modelar patrones a priori desconocidos, originados de interacciones complejas entre los datos. Los mismos tienen varios hyperparametros que son optimizables

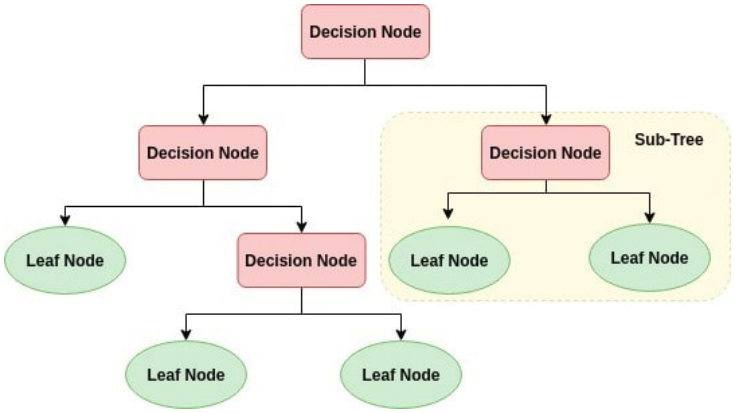

Figura 2: Esquema genérico de *Arboles de decisión*

para cada conjunto de datos, los mismos son tan variados como el tamaño del árbol (max-depth), parámetros de complejidad (cpp-alpha), etc.

3.2.2. Modelos de Boosting

Los modelos de Bagging o Boosting, y *XGBoost* en particular, son algoritmos mucho mas complejos y robustos estadísticamente que los modelos básicos de machine learning como los *Árboles*, *K-NN* o la propia *Regresión Logística*. Boosting es una técnica general para armar ensambles de varios modelos. Pensemos por ejemplo que en una regresión lineal examinamos los Residuos para ver como se puede mejorar el ajuste, Boosting aplica una lógica similar pero se revisan los propios modelos ensamblados para minimizar el error del modelo anterior. Seleccionamos una muestra aleatoria de datos, se ajusta a un modelo para luego entrenarlo secuencialmente (osea que cada modelo va a compensar las falencias del modelo anterior), de esta manera las reglas que son débiles para cada modelo individual se combinan para formar una regla de predicción fuerte (reduciendo el sesgo y/o la varianza de cada modelo por separado). Los algoritmos de boosting pueden diferir en la forma en que crean y agregan aprendices débiles durante el proceso secuencial. Una de las desventajas de este tipo de modelos es que se pierde interpretabilidad de los mismos, una de las ventajas que tenia usar modelos de *Árboles*.

3.2.3. Boosting de gradiente o *Gradient Boosting*

Es un modelo de ensamble basado en boosting, desarrollado por Leo Breimann y Jerome H. Friedman. Este modelo de ensamble trabaja agregando de manera secuencial predictores a un conjunto, donde cada uno va *corrigiendo* los sesgos del modelo anterior. El boosting de gradiente entrena los

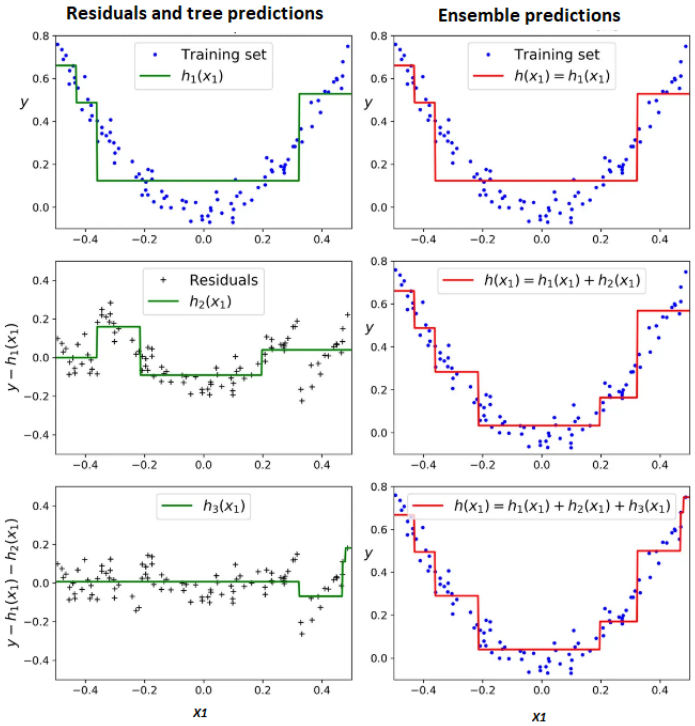

Figura 3: Evolución del *Gradient Boosting* a medida que se añaden modelos

errores residuales del predictor anterior para el siguiente. Esto lo podemos ver visualmente en la Fig 3.

3.2.4. Extreme Gradient Boosting (*XGBoost*)

Es un algoritmo desarrollador originalmente en C++ por Tianqi Chen y Carlos Guestrin en la universidad de Washington. Es un algoritmo paralelizable para clasificación o regresión, lo que nos permite usar muchos núcleos a la vez y le otorga una rapidez de computo muy alta. El mismo implementa algoritmos en el marco de *Gradient Boosting* para la optimización de los modelos y, como en el *Gradient Boosting*, *XGBoost* utiliza modelados de Árboles en paralelo. *XGBoost* funciona afín al método numérico Newton-Raphson en el espacio de funciones, a diferencia del *Gradient Boosting* que funciona según el método de Descenso de Gradiente en el espacio de funciones, donde se utiliza una aproximación de Taylor de segundo orden en la función de pérdida para establecer la conexión con el método Newton-Raphson. La ventaja de *XGBoost* por sobre *Gradient Boosting* es la optimización en el entrenamiento de los modelos utilizando los errores residuales de los modelos pasados, siendo un algoritmo muy utilizado para datasets grandes y complejos.

4. Medidas del Performance de los modelos

En los modelos de clasificación tenemos cuatro parámetros para medir su performance: "*accuracy*", "*precision*", "*recall*" y "$f1 - score$". Todos se basan en los conceptos de TP (verdadero positivo), TN (verdadero negativo), FP (falso positivo) y FN (falso negativo), donde los parámetros anteriores dan ponderaciones a distintos aspectos para la clasificación basados en estos conceptos.

- *Accuracy*: Es la proporción de elementos clasificados correctamente respecto a los casos totales (suele flaquear en datasets desbalanceados como es el caso).

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{6}$$

- *Precision*: Mide qué tan "*preciso*" es el clasificador al predecir las instancias positivas. Se calcula como el número de verdaderos positivos (TP) sobre todos los casos que son predichos positivos (TP+FP). Si su valor es bajo, es porque hay presencia de falsos positivos, por eso esta medida es sensible a los FP.

$$Precision = \frac{TP}{(TP + FP)} \tag{7}$$

- *Recall*: Mide la capacidad del modelo de detectar los verdaderos positivos (TP) sobre todos los casos que son positivos (TP+FN). Si su valor es bajo, es porque hay presencia de falsos negativos, por eso esta medida es sensible a los FN.

$$Recall = \frac{TP}{(TP + FN)} \tag{8}$$

- *F1-Score*: Es la media armónica de el Precision y el Recall. Para tener un "$F1 - Score$" alto, es necesario que tanto recall como precision sean altos, mientras que un "$F1 - Score$" bajo puede ser el resultado de un valor bajo en por lo menos una de estas métricas o en ambas a la vez.

$$F1 - Score = \frac{2 * Recall * Precision}{(Recall + Precision)} \tag{9}$$

Tambien utilizamos las curvas ROC y el índice AUC. Hay otra métrica que guarda equilibrio con el Recall que es la Specificity, mide qué tan

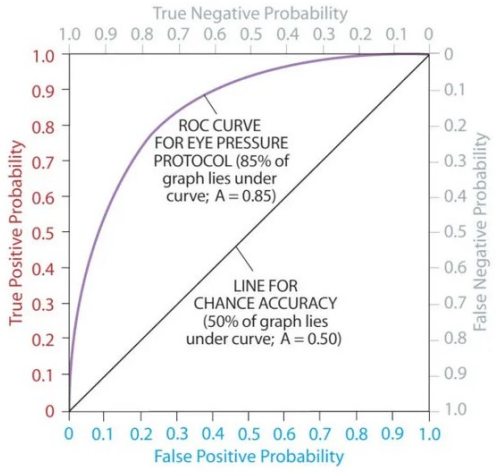

Figura 4: Esquema genérico de *Curva ROC* y *AUC*

"*específico*" es el clasificador al predecir las instancias positivas. Se calcula como el número de verdaderos negativos (TN) sobre todos los casos que son negativos (TN+FP):

$$Specificity = \frac{TN}{(TN + FP)} \tag{10}$$

Como dijimos antes, es evidente que guarda relación con el Recall. La métrica que captura este equilibrio entre los dos parámetros es la curva *Características operativas del receptor* (*Receiver Operating Characteristics*) o *Curva ROC*. Esta curva muestra el equilibrio entre el Recall y Specificity, a medida que cambie el corte para determinar como clasificar un registro, encontrándoce el Recall en el eje Y y tenemos en el eje x tenemos la Specificity - 1. En el gráfico veremos una linea diagonal punteada que representa a un clasificador que no es mejor que el azar, un clasificador extremadamente eficaz tendrá una *Curva ROC* que abraza la esquina superior izquierda. La *Curva ROC* no es una medida en si, sino que lo es el área bajo su curva, llamada *AUC* (area underneath the curve). Un *AUC* igual a 1 representa un clasificador perfecto y uno completamente ineficaz tendrá un área de 0.5. Estos conceptos los podemos ver representados en la Fig. 4.

5. Hiperparámetros

Los hiperparámetros son ciertas variables que pasamos como argumento a nuestros modelos, en la lista podemos ver hiperparámetros de la *Regresión*

Logística y de *XGBoost*.

Regresión Logística	*XGBoost*
penalty	booster
dual	verbosity
tol	validate parameters
C	nthread
fit intercept	disable default eval metric
intercept scaling	silent
class weight	eta
random state	min child weight
solver	max depth
max inter	max leaf nodes
multi class	gamma
warm start	max delta step
n jobs	subsample
l1 ratio	colsample bytree
	lambda
	alpha
	objective
	eval metric
	etc

Con esta tabla se puede apreciar la complejidad de *XGBoost* por sobre la *Regresión Logística*. Aunque ambos tengan varios hiperparámetros, solo vamos a setear los más relevantes. Siendo los hiperparámetros datos que tenemos que indicarle a mano al modelo hay combinaciones de los mismos que, dependiendo de los datos a trabajar, van a ajustar y predecir mas o menos efectivamente. Por ello existen varios métodos para hallar la mejor combinación de hiperparámetros posibles para maximizar los Scores mencionados anteriormente, estos son RandomSearchCV y GridSearchCV. Ambos, para cada combinación de valores de los hiperparámetros: aplica sobre el dataset de train, los evalúa con validación cruzada registra el score; al final de todas las búsquedas selecciona la combinación con más alto score y los aplica sobre los datos de train para luego predecir sobre los datos de test.

5.1. GridSearchCV

Es uno de los algoritmos mas simples para buscar hiperparámetros con técnicas de validación cruzada, al mismo se le brinda el modelo utilizar, un diccionario con los hiperparámetros y el número de conjuntos en los que se

divide los datos para la validación cruzada. GridSearchCV busca la mejor combinación de hiperparámetros dentro de una grilla (grid) especificada previamente. La búsqueda es exhaustiva para cada valor de la grilla. Tengamos en cuenta que no hay forma de saber de antemano cuáles son los mejores valores para los hiperparámetros, por lo que lo ideal es probar todos los valores posibles para conocer los valores óptimos. Hacer esto manualmente podría llevar una cantidad considerable de tiempo y recursos, por lo que utilizamos GridSearchCV para automatizar el ajuste de los hiperparámetros. GridSearchCV prueba todas las combinaciones de los valores introducidos en el diccionario y evalúa el modelo para cada combinación utilizando el método de validación cruzada. Por lo tanto, después de usar esta función obtenemos la precisión/pérdida para cada combinación de hiperparámetros y podemos elegir la que tenga el mejor rendimiento. Pero hay que tener en cuenta que puede ser extremadamente costosa desde el punto de vista computacional y puede llevar mucho tiempo pues busca exhaustivamente la mejor combinación de hiperparámetros.

5.2. RandomSearchCV

RandomSearchCV consiste en generar y evaluar entradas aleatorias a la función objetivo. Este algoritmo selecciona en forma aleatoria un subset de los hiperparámetros. Es interesante usar estos métodos porque no asumen nada sobre la estructura de la función objetivo. Esto puede ser beneficioso pues suele evitar preconcepciones que pueden influir o sesgar la estrategia de optimización, lo que permite descubrir soluciones no intuitivas. Definimos distribuciones para cada hiperparámetro que pueden definirse uniformemente o con un método de muestreo. La diferencia clave con GridSearchCV es que en RandomSearchCV no se prueban todos los valores y los valores probados se seleccionan al azar, de esta manera volviéndolo una alternativa con un costo computacional mucho menor pero menor rigurosidad.

6. Performance de los modelos

Para la mayor eficiencia de los modelos, estandarizamos nuestros datos y definimos a las clases con números, tal que "$STAR : 0$", "$GALAXY : 1$" y "$QSO : 2$".

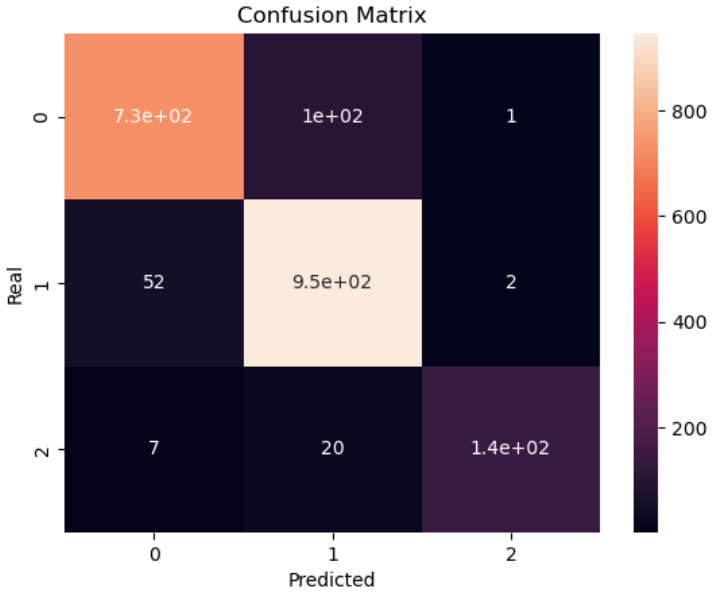

Figura 5: Matriz de Confusión de la Regresión Logística como hipótesis nula.

6.1. Regresión Logística

6.1.1. Hipótesis Nula

Corremos el modelo para nuestros datos sin setear ningún hiperparámetro. Este modelo tiene un tiempo computacional prácticamente nulo, con la siguiente performance:

Performance de la hipótesis nula de la Regresión Logística.			
Objeto/Score	Precision	Recall	F1-Score
Star (0)	0.92	0.88	0.90
Galaxy (1)	0.89	0.95	0.91
Cuásar (2)	0.98	0.84	0.91

El accuracy de este modelo es de 0.91.

La matriz de confusión de la predicción podemos verla en la Fig. 5.

6.1.2. GridSearchCV en Regresión Logística

Ahora utilizamos GridSearchCV para determinar los hiperparámetros óptimos, en los que pasamos los valores:

- *Penalty*: Es la norma de las sanciones (o pesos) a los parámetros. Se le pasaron las opciones de "*none*", "*l*1", "*l*2" y "*elasticnet*".

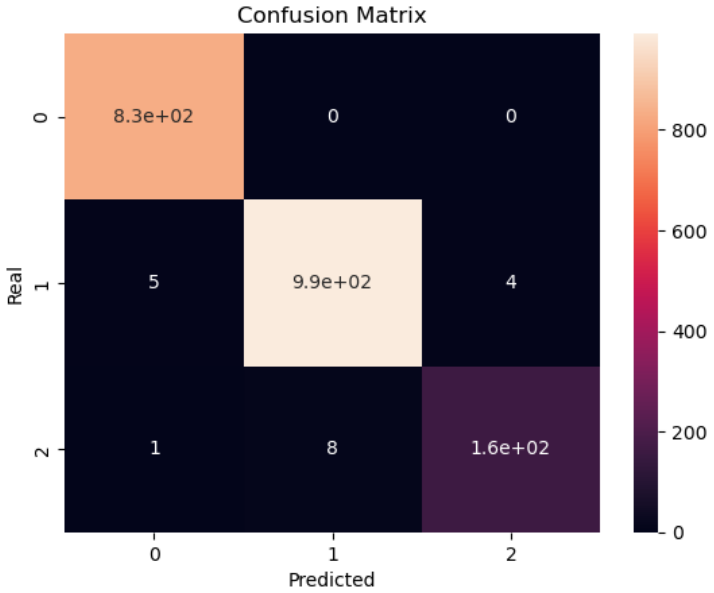

Figura 6: Matriz de Confusión de la Regresión Logística con GridSearchCV.

- *Solver*:Algoritmo a utilizar en el problema de optimización. Se le pasaron las opciones de "$lbfgs$", "$liblinear$", "$newton-cg$", "$newton-cholesky$", "sag", "$saga$".
- C: Inverso de la fuerza de regularización; debe ser un valor flotante positivo. Como en las máquinas de vectores soporte, los valores más pequeños especifican una regularización más fuerte. Se le pasaron 20 valores posibles para C.

La ejecución de GridSearchCV tardó 2.4 minutos, y nos dice que la mejor combinación de hiperparámetros son C: 1e-05, penalty: none, solver: newton-cg.

La performance del modelo para estos hiperparámetros es:

Performance de la Regresión Logística con GridSearchCV.			
Objeto/Score	Precision	Recall	F1-Score
Star (0)	0.99	1.00	1.00
Galaxy (1)	0.99	0.99	0.99
Cuásar (2)	0.98	0.95	0.96

El accuracy de este modelo es de 0.99.

Podemos ver en la Fig. 6 la matriz de confusión.

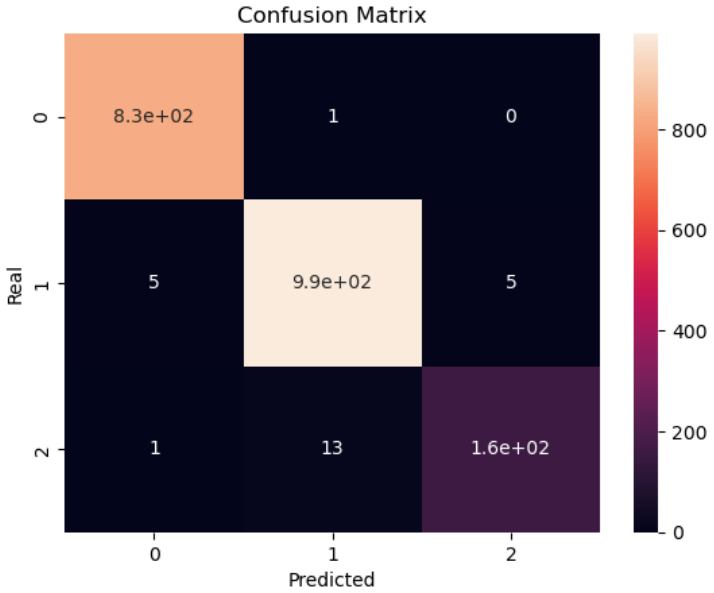

Figura 7: Matriz de Confusión de la Regresión Logística con RandomSearchCV.

6.1.3. RandomSearchCV en Regresión Logística

Se le pasaron los mismos hiperparámetros para la optimización, brindándonos los hiperparámetros solver: liblinear, penalty: l1, C: 263665089.87. Este algoritmo necesitó un tiempo de ejecución de 8 segundos. La performance de este modelo fue:

Performance de la Regresión Logística con RandomSearchCV.			
Objeto/Score	Precision	Recall	F1-Score
Star (0)	0.99	1.00	1.00
Galaxy (1)	0.99	0.99	0.99
Cuásar (2)	0.97	0.92	0.94

El accuracy de este modelo es de 0.99.

Podemos ver en la Fig. 7 la matriz de confusión.

6.2. XGBoost

6.2.1. Hipótesis Nula

Igual que en el caso del modelo anterior, primero implementamos $XGBoost$ con los hiperparámetros con default. Como vemos a continuación, el la per-

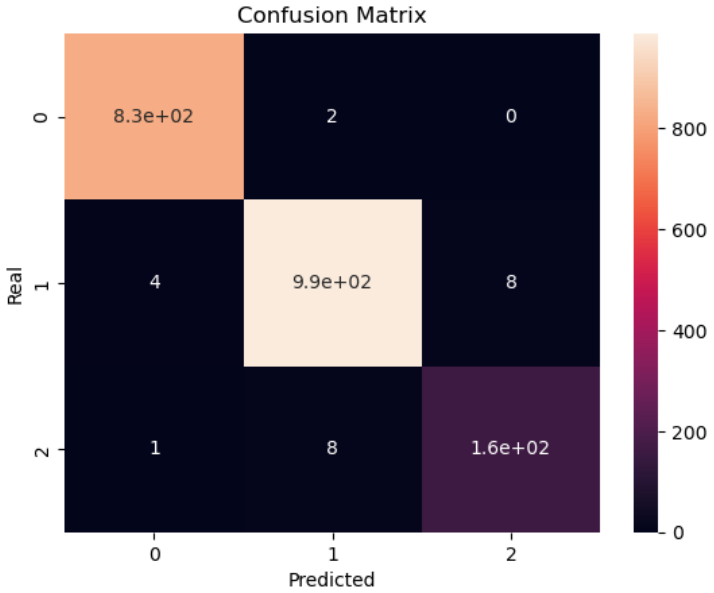

Figura 8: Matriz de Confusión de la hipótesis nula con *XGBoost*.

formance de la hipótesis nula de *XGBoost*, es mejor predictor que el caso de la *RegresionLogistica*, teniendo un coste computacional similar.

Performance de la Regresión Logística con RandomSearchCV.			
Objeto/Score	Precision	Recall	F1-Score
Star (0)	0.99	1.00	1.00
Galaxy (1)	0.99	0.99	0.99
Cuásar (2)	0.95	0.95	0.95

El accuracy de este modelo es de 0.99.

Podemos ver en la Fig. 7 la matriz de confusión.

6.2.2. GridSearchCV en XGBoost

Como vimos anteriormente, XGBoost tiene gran cantidad de hiperparámetros. La 1ra vez que se pasaron los valores por el algoritmo tardó mas de 25 horas, por lo que se acotaron los valores por los siguiente:

- *min-child-weight*: Suma mínima de peso de instancia (hessian) necesaria en un hijo. Si el paso de partición del árbol da como resultado un nodo hoja con la suma del peso de instancia menor que min-child-weight, entonces el proceso de construcción renunciará a seguir particionando.

En la tarea de regresión lineal, esto corresponde simplemente al número mínimo de instancias necesarias en cada nodo. Cuanto mayor sea min-child-weight, más conservador será el algoritmo. Se le pasó los valores de 0, 1 y 2.

- *Gamma*: Reducción mínima de pérdidas necesaria para hacer una partición más en un nodo hoja del árbol. Cuanto mayor sea gamma, más conservador será el algoritmo. Se le pasó los valores de 0, 0.1 y 0.2.

- *Subsample*: Proporción de submuestreo de las instancias de entrenamiento. Si se establece en 0,5, XGBoost muestreará aleatoriamente la mitad de los datos de entrenamiento antes de hacer crecer los árboles, lo que evitará el sobreajuste. El submuestreo se realizará una vez en cada iteración de boosting. Se le pasó los valores de 0.8, 0.9, 1.0, 1.1.

- *colsample-bytree*: colsample-bytree es la proporción de submuestreo de columnas al construir cada árbol. El submuestreo se realiza una vez por cada árbol construido. Se le pasó los valores de 0.7, 0.8, 0.9.

- *reg-alpha*: Término de regularización L1 de los pesos. Aumentar este valor hará que el modelo sea más conservador. Se le pasó los valores de 1e-5, 1e-2, 0.1, 1, 3.

- *max-depth*: Profundidad máxima de un árbol. Aumentar este valor hará que el modelo sea más complejo y más propenso a sobreajustarse. 0 indica que no hay límite en la profundidad. Tengamos en cuenta que XGBoost consume mucha memoria cuando entrena un árbol profundo. Se le pasó los valores de 3, 4, 5.

- *n-estimators*: Es la cantidad de modelos de Arbol máxima que se puede desarrollar. Se le pasó los valores de 160, 175, 185, 200.

- *eta*: La reducción del tamaño del paso se utiliza en la actualización para evitar el sobreajuste. Después de cada paso de refuerzo, podemos obtener directamente los pesos de las nuevas características, y eta reduce los pesos de las características para que el proceso de refuerzo sea más conservador. Se le pasó los valores de 0.1, 0.2, 0.3, 0.4, 0.5, 0.6.

Con estos hiperparámetros, GridSearchCV, tardó 461 minutos en hallar la mejor combinación. La cual fue: colsample-bytree: 0.7, eta: 0.5, gamma: 0.2, max-depth: 5, min-child-weight: 1, n-estimators: 175, reg-alpha: 0, subsample: 0.9. Además de ofrecernos una performance levemente superior que si versión sin optimización de hiperparámetros:

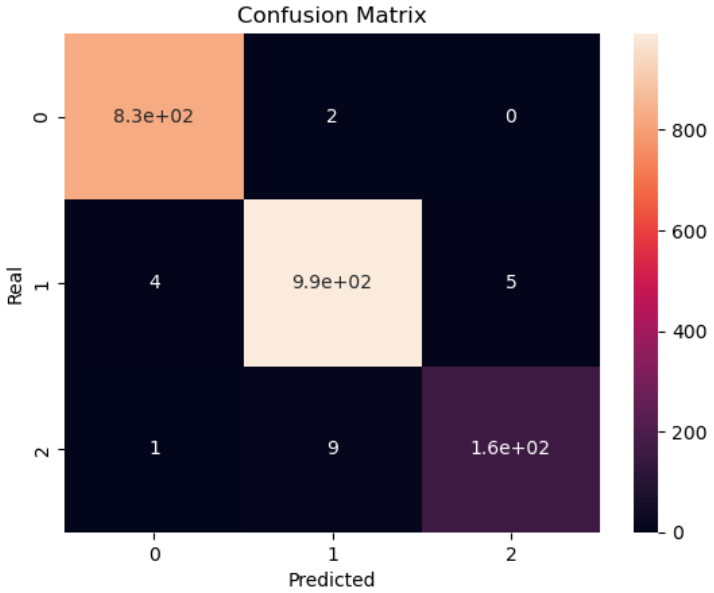

Figura 9: Matriz de Confusión de *XGBoost* optimizado con GridSearchCV.

Performance de XGBoost con GridSearchCV.			
Objeto/Score	Precision	Recall	F1-Score
Star (0)	0.99	1.00	1.00
Galaxy (1)	0.99	0.99	0.99
Cuásar (2)	0.97	0.94	0.96

El accuracy de este modelo es de 0.99.

Podemos ver en la Fig. 8 la matriz de confusión.

6.2.3. RandomSearchCV en XGBoost

Realizamos el mismo procedimiento con RandomSearchCV, el cual tardó solamente 12 segundos para darnos el mismo resultado que GridSearchCV, pero con diferente cominacion de hiperparámetros: subsample: 0.8, reg-alpha: 0.01, n-estimators: 200, min-child-weight: 0, max-depth: 3, gamma: 0.2, eta: 0.1, colsample-bytree: 0.9.

Performance de XGBoost con RandomSearchCV.			
Objeto/Score	Precision	Recall	F1-Score
Star (0)	0.99	1.00	1.00
Galaxy (1)	0.99	0.99	0.99
Cuásar (2)	0.97	0.95	0.96

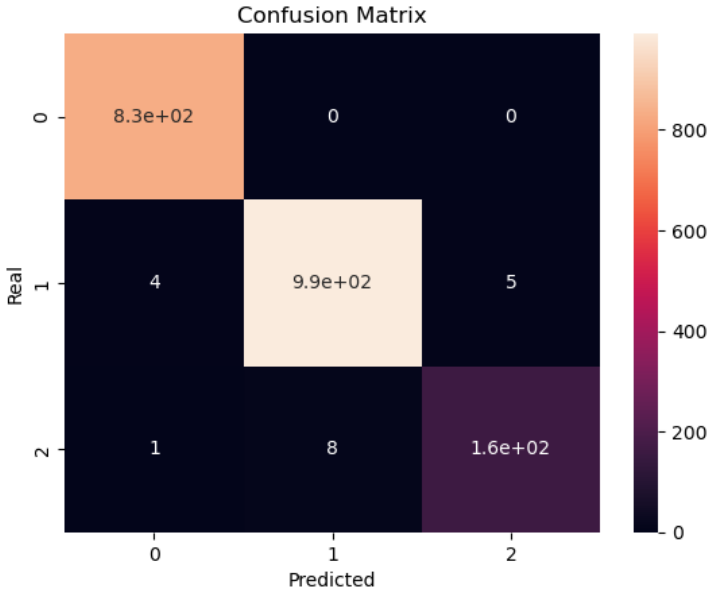

Figura 10: Matriz de Confusión de *XGBoost* optimizado con RandomSearchCV.

El accuracy de este modelo es de 0.99.

Podemos ver en la Fig. 10 la matriz de confusión.

7. Curvas ROC y AUC

Para la mejor versión de los modelos tenemos *curvasROC* y *AUC* casi perfectas, salvo los Cuásars, ya que es una clase desbalanceada a comparación de las otras dos, no tiene tantas muestras para entrenar. Las mismas las podemos ver en las Fig. 11 y 12.

8. Feature Importance

Podemos sacar más información valiosa a partir de nuestros modelos, como saber cuáles features fueron más determinantes para la discriminación de las clases. Las mismas las podemos ver para el mejor modelo de *XGBoost* en la Fig. 13 respectivamente.

Como podemos ver, con diferencia, las features más importantes en la predicción fueron el "$redshift$" y el parámetro fotométrico "*plate*". Esto es importante desde el punto de vista de que, para intentar hacer modelos

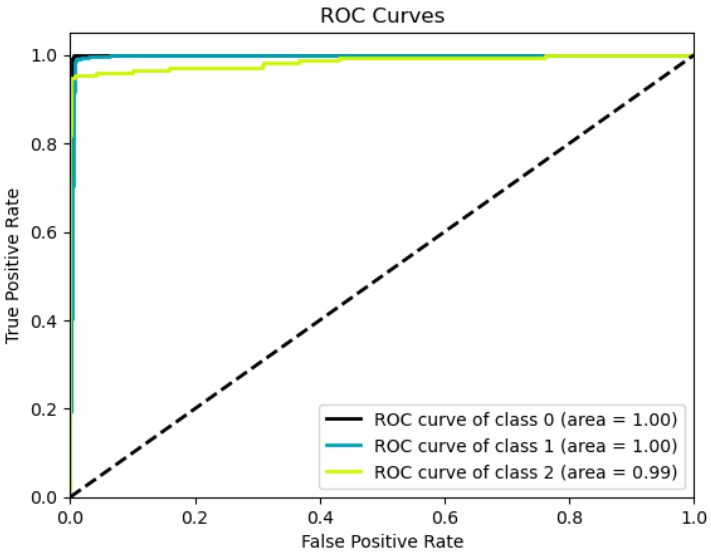

Figura 11: Curvas ROC de *XGBoost*.

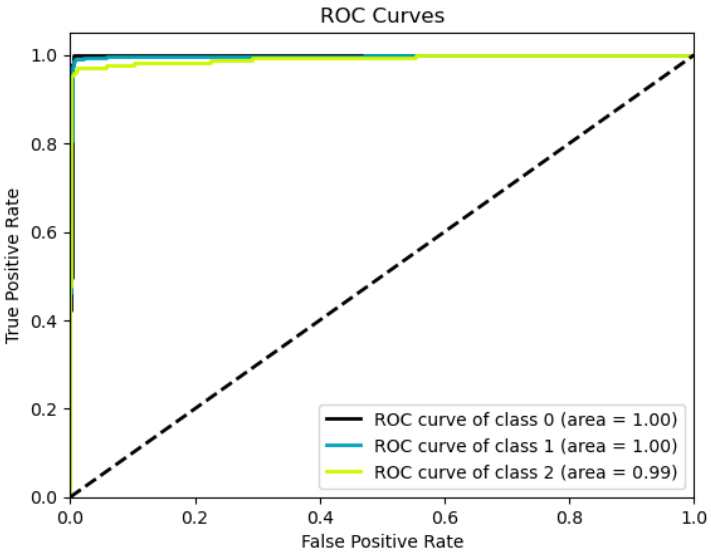

Figura 12: Curvas ROC de *XGBoost*.

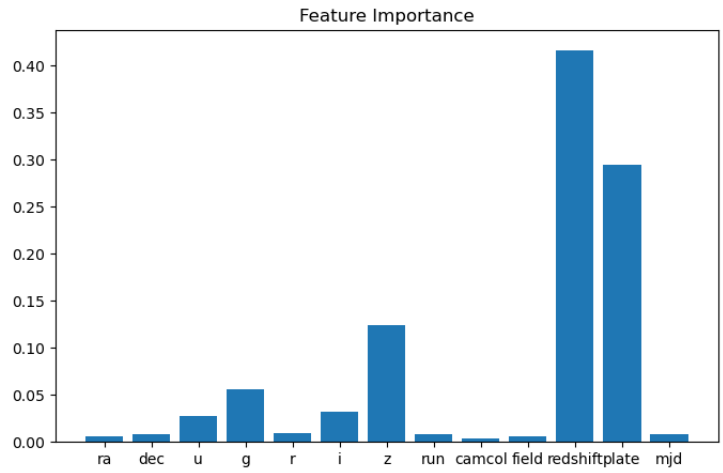

Figura 13: Feature Importance.

más precisos, mientras mejor calculado sea el "$redshift$" más certeros serán nuestros modelos. Algo interesante es ver que ocurre si omitimos la feature "$redshift$" de la predicción, y ver cuanto empeorará nuestro modelo. Recordemos que el redshift no es un dato directo, sino que el mismo debe ser calculado ya sea tradicionalmente o con otras técnicas de machine learning.

Si omitimos el redchift como variable predictora, el modelo nos da una performance:

Performance de XGBoost con GridSearchCV omitiendo el Redshift.			
Objeto/Score	Precision	Recall	F1-Score
Star (0)	0.93	0.93	0.93
Galaxy (1)	0.95	0.95	0.95
Cuásar (2)	0.90	0.88	0.89

El accuracy de este modelo es de 0.94.

Podemos ver en la Fig. 14 la matriz de confusión, y en la Fig. 15 podemos ver la *curva ROC*.

Vemos que el modelo empeoró, aunque no sustancialmente, por lo que con los datos fotométricos y espectroscópicos son bastante buenos a la hora de predecir por si mismos el tipo de fuente.

9. Conclusión

Como pudimos ver, a la hora de optimizar hiperparámetros es mas conveniente utilizar algoritmos e hiperparámetros más simples a lo referente a

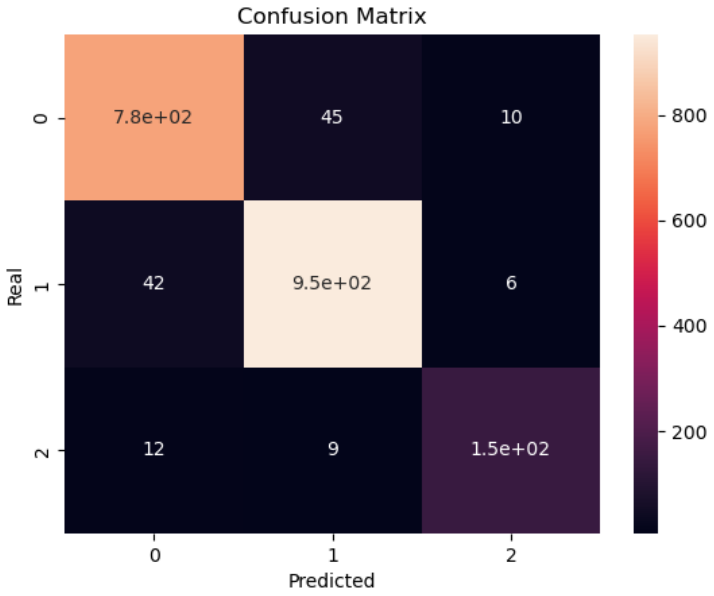

Figura 14: Matriz de Confusión de *XGBoost* optimizado omitiendo la feature "$redshift$".

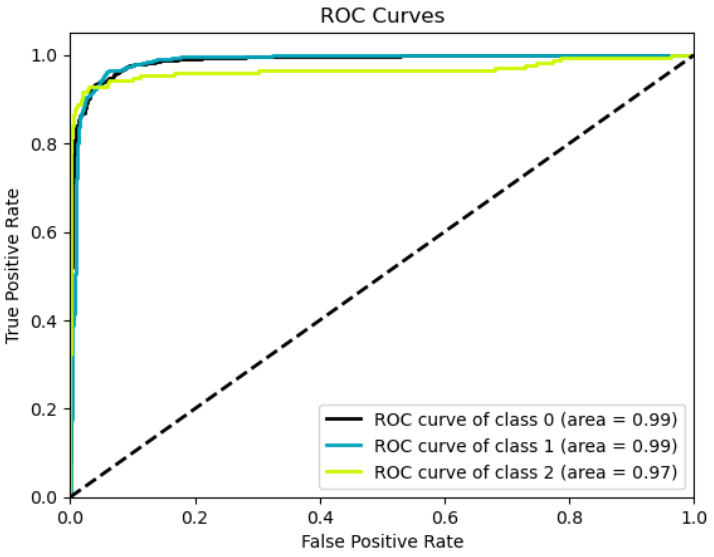

Figura 15: Curva ROC de *XGBoost* optimizado omitiendo la feature "$redshift$".

los tiempos de computo. Para nuestro set de datos de solo 10000 fuentes y con una proporción baja de cuásares, vemos que en general todos los modelos implementados son muy buenos predictores. Además de comprobar, mas que nada en la Regresión Logística, que aunque RandomSearchCV sea mucho más rápido, puede darnos una combinación de hiperparámetros que no sea la mejor a comparación de los dados por GridSearchCV. A la hora de usar datasets mucho mas grandes o tener datos que no sean tan precisos vale la pena utilizar algoritmos de ensamble como XGBoost para tener la mayor precisión posible so pena de los tiempos de computo. En este trabajo desarrollamos varios algoritmos bastante existosos a la hora de hacer una clasificación de fuentes, incluso tomando datos directamente del SDSS que no incluyan el redshift.