

COMPARACIÓN DE ALGORITMO DE IA PARA PREDICCIÓN DE SERIES TEMPORALES

AI ALGORITHM COMPARISON FOR TIME
SERIES PREDICTION



TRABAJO FIN DE GRADO
CURSO 2024-2025

AUTOR
RAÚL GONZÁLEZ MUÑOZ
DIRECTOR
MIGUEL ÁNGEL ROJAS GÓMEZ

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

AGRADECIMIENTOS

A Miguel Ángel Rojas Gómez por aceptar ser mi tutor y apoyarme con ideas, recursos y contactos que han sido indispensables para un correcto desarrollo del trabajo. A José Manuel Velasco Cabo por introducirme a la regresión simbólica y al software de Atas.

RESUMEN

Comparación de algoritmo de IA para predicción de series temporales

Este trabajo consistirá en dar respuesta a dos preguntas, la primera es que algoritmo de inteligencia artificial es más adecuado para la predicción de series temporales, para ello utilizaré el precio de la criptomoneda Solana en el rango de las horas. La segunda pregunta es si el rendimiento de los algoritmos mejorará si además de los datos de precios utilizamos el índice de miedo-codicia en la predicción.

Para responder a estas preguntas implementaré cinco algoritmos de inteligencia artificial, RandomForest, XGBoost, redes neuronales, redes neuronales LSTM y regresión simbólica. Una vez implementados evaluaré la calidad de sus predicciones usando la diferencia porcentual entre el valor real y el valor predicho por el algoritmo. El algoritmo que minimice ese valor será el que consigue obtener mejores predicciones. El código asociado a este proyecto está en un repositorio de Git Hub al que se puede acceder a través de este enlace: <https://github.com/Raul-Gonzalez-M/TFG>

Palabras clave

RandomForest, XGBoost, redes neuronales, redes neuronales LSTM, regresión simbólica, índice de miedo-codicia.

ÍNDICE DE CONTENIDOS

Capítulo 1 - Introducción.....	1
1.1 Motivación	1
1.2 Objetivos.....	1
1.2.1 Subobjetivos	1
1.3 Plan de trabajo	2
1.4 Motivation	2
1.5 Objectives	2
1.5.1 Specific Objectives	3
1.6 Workflow	3
Capítulo 2 - Estado de la cuestión.....	5
2.1 Historia de la IA	5
2.2 Estado del Arte.....	6
Capítulo 3 - Explicación e Implementación de los Algoritmos.....	9
3.1 RandomForest.....	9
3.1.1 Implementación.....	9
3.2 XGBoost	12
3.2.1 Implementación.....	12
3.3 Redes Neuronales.....	15
3.3.1 Implementación.....	16
3.4 Redes Neuronales LSTM.....	19
3.4.1 Implementación.....	21
3.5 Regresión Simbólica	23

3.5.1 Implementación.....	24
Capítulo 4 - Evaluación de Resultados.....	35
4.1 Random Forest	35
4.1.1 Sin Índice de Miedo-Codicia	35
4.1.2 Con Índice de Miedo-Codicia.....	38
4.1.3 Conclusión	41
4.2 XGBoost	42
4.2.1 Sin Índice de Miedo-Codicia	42
4.2.2 Con Índice de Miedo-Codicia.....	46
4.2.3 Conclusión	50
4.3 Redes Neuronales	51
4.3.1 Sin Índice de Miedo-Codicia	51
4.3.2 Con Índice de Miedo-Codicia.....	55
4.3.3 Conclusión	59
4.4 Redes Neuronales LSTM.....	60
4.4.1 Sin Índice de Miedo-Codicia	60
4.4.2 Con Índice de Miedo-Codicia.....	64
4.4.3 Conclusión	68
4.5 Regresión Simbólica	69
4.5.1 Sin Índice de Miedo-Codicia ni Constantes	69
4.5.2 Sin Índice de Miedo-Codicia con Constantes	70
4.5.3 Con Índice de Miedo-Codicia sin Constantes	71
4.5.4 Con Índice de Miedo-Codicia con Constantes.....	73
4.5.5 Conclusión	73
Capítulo 5 - Conclusiones y trabajo futuro.....	77

5.1 Conclusiones	77
5.2 Trabajo Futuro	78
5.3 Conclusions	78
5.4 Future Work.....	80
Bibliografía.....	81

ÍNDICE DE FIGURAS

Figura 2-1. Línea temporal	5
Figura 2-2. Algoritmos genéticos	7
Figura 2-3. Redes Neuronales	8
Figura 3-1. Preparar datos RandomForest	10
Figura 3-2. Eval RandomForest	10
Figura 3-3. Train RandomForest depth	11
Figura 3-4. Train RandomForest.....	11
Figura 3-5. Llamada RandomForest	12
Figura 3-6. Create_df_n XGBoost	13
Figura 3-7. Create dataframes XGBoost.....	13
Figura 3-8. Preparar_datos XGBoost	13
Figura 3-9. Eval XGBoost.....	14
Figura 3-10. Train_XGB_depth XGBoost.....	14
Figura 3-11. TrainGlobalXGB XGBoost	15
Figura 3-12. Llamada XGBoost	15
Figura 3-13. Create_sequences redes neuronales	17
Figura 3-14. Preparar_datos redes neuronales.....	17
Figura 3-15. Eval redes neuronales.....	17
Figura 3-16. Estrategia redes neuronales	17
Figura 3-17. Opti_redes_densas redes neuronales	18
Figura 3-18. Opti_rd_h redes neuronales	19
Figura 3-19. Llamada redes neuronales.....	19
Figura 3-20. Create_sequences LSTM.....	21

Figura 3-21. Preparar_datos LSTM.....	21
Figura 3-22. Eval LSTM	22
Figura 3-23. Opti_redes LSTM	22
Figura 3-24. Opti_rLSTM_h LSTM.....	23
Figura 3-25. Llamada LSTM.....	23
Figura 3-26. Init regresión simbólica	25
Figura 3-27. Create regresión simbólica	25
Figura 3-28. Create_priv regresión simbólica	25
Figura 3-29. Fitness2 regresión simbólica.....	26
Figura 3-30. Evaluate regresión simbólica	26
Figura 3-31. Aplica_operaciones regresión simbólica	27
Figura 3-32. Display regresión simbólica	27
Figura 3-33. Mutate2 primera versión regresión simbólica.....	28
Figura 3-34. Mutate2 versión final regresión simbólica	28
Figura 3-35. Run regresión simbólica	29
Figura 3-36. Run segunda versión regresión simbólica	30
Figura 3-37. Run versión final regresión simbólica	31
Figura 3-38. Class Position regresión simbólica constantes	32
Figura 3-39. Class Constant regresión simbólica constantes	32
Figura 3-40. Cargar regresión simbólica constantes	32
Figura 3-41. Create_gen regresión simbólica constantes	33
Figura 3-42. AniadirNum regresión simbólica constantes.....	33
Figura 3-43. Display regresión simbólica constantes.....	33

ÍNDICE DE TABLAS

Tabla 4-1. Profundidad máxima 1 a 400 RandomForest	35
Tabla 4-2. Profundidad máxima 7 a 150 RandomForest	36
Tabla 4-3. Profundidad máxima 1 a 8 RandomForest	36
Tabla 4-4. Profundidad máxima 7 a 10 RandomForest	37
Tabla 4-5. Profundidad máxima 7 a 400 RandomForest	37
Tabla 4-6. Profundidad máxima 1 a 400 RandomForest IMC	38
Tabla 4-7. Profundidad máxima 7 a 150 RandomForest IMC	39
Tabla 4-8. Profundidad máxima 1 a 8 RandomForest IMC	39
Tabla 4-9. Profundidad máxima 7 a 10 RandomForest IMC	40
Tabla 4-10. Profundidad máxima 7 a 400 RandomForest IMC	40
Tabla 4-11. RandomForest IMC vs Normal.....	41
Tabla 4-12. Eta = 0,3 completa XGBoost	42
Tabla 4-13. Eta = 0,3, max depth 20 XGBoost.....	43
Tabla 4-14. Eta = 0,1 completa XGBoost	44
Tabla 4-15. Eta = 0,1 , max depth 20 XGBoost.....	44
Tabla 4-16. Eta = 0,01 completa XGBoost	45
Tabla 4-17. Eta = 0,01 , max depth 20 XGBoost.....	46
Tabla 4-18. Eta = 0,3 completa XGBoost IMC	47
Tabla 4-19. Eta = 0,3, max depth 20 XGBoost IMC.....	47
Tabla 4-20. Eta = 0,1 completa XGBoost IMC	48
Tabla 4-21. Eta = 0,1 , max depth 20 XGBoost IMC.....	48
Tabla 4-22. Eta = 0,01 completa XGBoost IMC	49
Tabla 4-23. Eta = 0,01 , max depth 20 XGBoost IMC.....	49

Tabla 4-24. XGBoost IMC vs Normal	51
Tabla 4-25. Epoch 4 redes neuronales.....	52
Tabla 4-26. Epoch 6 redes neuronales.....	52
Tabla 4-27. Epoch 10 redes neuronales	53
Tabla 4-28. Epoch 14 redes neuronales	54
Tabla 4-29. Epoch 20 redes neuronales.....	54
Tabla 4-30. Epoch 40 redes neuronales	55
Tabla 4-31. Epoch 4 redes neuronales IMC	56
Tabla 4-32. Epoch 6 redes neuronales IMC	56
Tabla 4-33. Epoch 10 redes neuronales IMC	57
Tabla 4-34. Epoch 14 redes neuronales IMC	58
Tabla 4-35. Epoch 20 redes neuronales IMC	58
Tabla 4-36. Epoch 40 redes neuronales IMC	59
Tabla 4-37. Redes neuronales IMC vs Normal	60
Tabla 4-38. Epoch 4 redes neuronales LSTM.....	61
Tabla 4-39. Epoch 6 redes neuronales LSTM.....	61
Tabla 4-40. Epoch 10 redes neuronales LSTM.....	62
Tabla 4-41. Epoch 14 redes neuronales LSTM.....	63
Tabla 4-42. Epoch 20 redes neuronales LSTM.....	63
Tabla 4-43. Epoch 40 redes neuronales LSTM.....	64
Tabla 4-44. Epoch 4 redes neuronales LSTM IMC	65
Tabla 4-45. Epoch 6 redes neuronales LSTM IMC	65
Tabla 4-46. Epoch 10 redes neuronales LSTM IMC	66
Tabla 4-47. Epoch 14 redes neuronales LSTM IMC	67
Tabla 4-48. Epoch 20 redes neuronales LSTM IMC	67

Tabla 4-49. Epoch 40 redes neuronales LSTM IMC	68
Tabla 4-50. LSTM IMC vs Normal	69
Tabla 4-51. Regresión Simbólica sin constantes y sin IMC	70
Tabla 4-52. Regresión Simbólica con constantes y sin IMC	71
Tabla 4-53.. Regresión Simbólica sin constantes y con IMC completa.....	72
Tabla 4-54. Regresión Simbólica sin constantes, con IMC y valor < 1000	72
Tabla 4-55. Regresión Simbólica con constantes y con IMC	73
Tabla 4-56. Regresión Simbólica top 10 resultados.....	74
Tabla 5-1. Comparación de algoritmos	77
Tabla 5-2. Comparison of algorithms	79

Capítulo 1 - Introducción

1.1 Motivación

Durante los últimos años la inteligencia artificial ha experimentado un gran progreso, expandiendo sus posibilidades de aplicación a diferentes sectores y problemas. Uno de los problemas a los que se aplica la inteligencia artificial es la predicción de series temporales, como pueden ser la demanda de energía eléctrica, el precio de las acciones o las cosechas.

Una de las áreas que podría beneficiarse de estos algoritmos es la predicción del precio de las criptomonedas, tanto para operar con ellas como activos a corto plazo como para facilitar la operación con ellas como un medio de pago.

Por ello creo interesante explorar que algoritmos de inteligencia artificial obtienen mejores resultados en esta tarea usando como ejemplo específico el precio de la criptomoneda Solana, actualmente la quinta en cuanto a capitalización de mercado.

1.2 Objetivos

El objetivo de este TFG es la implementación, en Python, de varios modelos de machine learning cuyo propósito es la predicción del precio de la criptomoneda Solana en la franja horaria. Una vez implementados estos algoritmos compararemos la efectividad de cada uno a la hora de predecir el precio con el objetivo de discernir cuál es el más adecuado para esta tarea.

1.2.1 Subobjetivos

1. Implementar a través de librerías de Python el algoritmo Random Forest Regressor.
2. Implementar a través de librerías de Python el algoritmo XGBoost.
3. Implementar a través de librerías de Python una red neuronal básica.
4. Implementar a través de librerías de Python una red neuronal LSTM.
5. Programar en Python el algoritmo genético de regresión simbólica.
6. Entrenar y optimizar los algoritmos.
7. Comparar las predicciones de los 5 algoritmos para discernir cual es el más adecuado para esta tarea.

1.3 Plan de trabajo

En este TFG compararé varios modelos de inteligencia artificial para comprobar cual es capaz de predecir con mayor precisión el precio de la criptomonedra Solana.

Lo primero que haré será dar una breve explicación acerca del fundamento teórico y matemático de cada algoritmo, seguida de una descripción detallada de su implementación.

Tras los detalles teóricos y de implementación procederé a explicar los resultados después de entrenar cada modelo y realizar predicciones usándolo, primero lo haré con cada algoritmo individualmente, y finalmente haré una comparación entre todos los algoritmos para concluir cual o cuales aportan mejores resultados.

1.4 Motivation

In recent years, artificial intelligence has made significant progress, expanding its range of applications to various sectors and problems. One of the issues where artificial intelligence is applied is time series forecasting, such as predicting electricity demand, stock prices, or crop yields.

One area that could benefit from these algorithms is cryptocurrency price prediction, both for trading them as short-term assets and for facilitating their use as a means of payment.

For this reason, I find it interesting to explore which artificial intelligence algorithms achieve the best results in this task, using the price of the cryptocurrency Solana in the hour range as a specific example, currently the fifth-largest in terms of market capitalization.

1.5 Objectives

The objective of this project is to implement, in Python, several machine learning models aimed at predicting the price of the Solana cryptocurrency in the hour range. Once these algorithms are implemented, we will compare the effectiveness of each in predicting the price to determine which is most suitable for this task.

1.5.1 Specific Objectives

1. Implement the Random Forest Regressor algorithm using Python libraries.
2. Implement the XGBoost algorithm using Python libraries.
3. Implement a basic neural network using Python libraries.
4. Implement an LSTM neural network using Python libraries.
5. Program the symbolic regression genetic algorithm in Python.
6. Train and optimize the algorithms.
7. Compare the predictions of the five algorithms to determine which is best suited for this task.

1.6 Workflow

In this project, I will compare several artificial intelligence models to see which one is most capable of accurately predicting the price of the Solana cryptocurrency.

I will first briefly explain the theoretical and mathematical foundation of each algorithm, followed by a detailed description of its implementation.

After the theoretical and implementation details, I will proceed to explain the results after training each model and making predictions using it. I will first do this with each algorithm individually, and finally, I will compare all the algorithms to determine which one provides the best results.

Capítulo 2 - Estado de la cuestión

2.1 Historia de la IA

El campo de investigación de la inteligencia artificial se fundó como disciplina académica en **1956** cuando **John McCarthy** acuñó el término, seis años después de que **Alan Turing** publicara su célebre artículo “**Computing machinery and intelligence**” [1]. Durante las siguientes décadas se produjeron grandes avances como el primer programa de procesamiento de lenguaje natural. A pesar de esto no consiguieron cumplir con las, poco realistas, expectativas que se habían impuesto. Esto conllevo una retirada de la financiación, hasta que se retomó siete años después gracias a una iniciativa del gobierno japonés. Sin embargo, este periodo no duró mucho debido a que a finales de la década de los 80 se volvió a retirar la financiación y comenzó la etapa conocida como el **invierno de la IA** que duró hasta **mediados de los 2000** [2], aun así, en **1997** se alcanzó un hito importante cuando **Deep Blue** de IBM venció al entonces campeón mundial de ajedrez **Garry Kasparov**.

Desde mediados de los años 2000 hasta 2020, ha habido grandes avances destacando entre ellos: la integración de **Siri** en el iPhone en el año **2011**, la primera computadora **superó el test de Turing** en el año **2014** o la victoria de **AlphaGo** de DeepMind sobre el campeón mundial de **Go Lee Sedol** en el año **2016** [3].

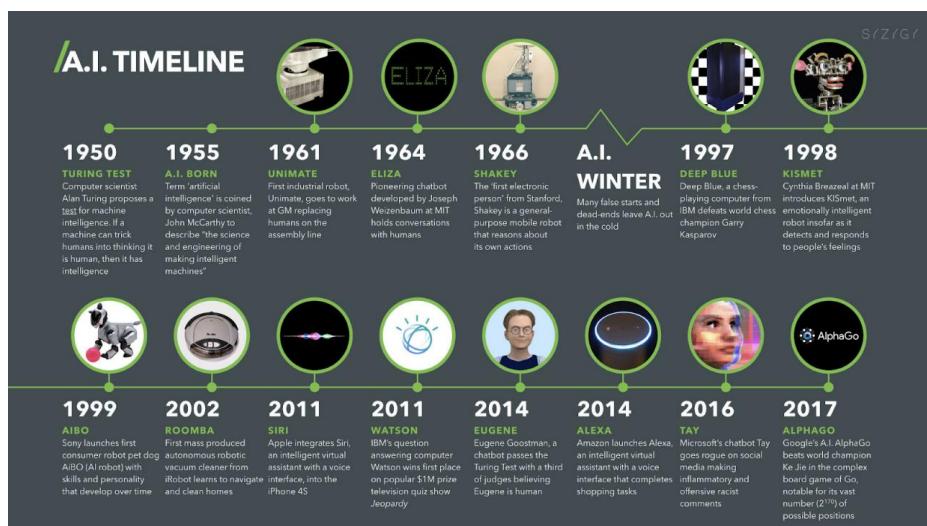


Figura 2-1. Línea temporal

En la actualidad se ha llegado a grandes avances en el campo de la inteligencia artificial, esto ha sido propiciado por varios factores como: el aumento en la cantidad y calidad de los datos y el aumento de los recursos técnicos. Algunos de los avances más significativos han sido: las inteligencias artificiales generativas, como pueden ser **ChatGPT**, lanzado al público en **2022**, o **LeonardoAI**, para generar texto y texto a imagen respectivamente, la conducción autónoma con el autopilot de Tesla y los múltiples softwares de visión por computadora y reconocimiento de voz. También se han producido avances en otras áreas como en la predicción del tiempo atmosférico, siendo capaces de predecir un huracán que afectó a Colorado permitiendo evacuar a la población a tiempo o en la medicina por ejemplo en el desarrollo de prótesis robóticas más avanzadas. La mayoría de estas inteligencias artificiales han sido desarrolladas utilizando algoritmos de machine learning.

2.2 Estado del Arte

La Inteligencia Artificial es el conjunto de algoritmos y técnicas que permiten a sistemas informáticos realizar tareas que normalmente requerirían inteligencia humana. Tradicionalmente se clasifica en dos tipos: **débil** y **general** [4]. La **inteligencia Artificial débil** se caracteriza por resolver un problema específico, es la única que existe en la actualidad. En contraposición, el propósito de la **Inteligencia Artificial general** es el de ser capaz de enfrentar cualquier tipo de problema, sin embargo, en la actualidad no se disponen de los medios técnicos para llevarla a cabo. Otra clasificación ampliamente utilizada es la que divide la **Inteligencia Artificial en simbólica** y **subsimbólica**. La **simbólica** se basa en representaciones de alto nivel y posee un enfoque deductivo por lo que es utilizada principalmente para la representación de conocimiento, por su parte la **subsimbólica** no requiere representaciones de alto nivel y posee un enfoque inductivo, actualmente es la más utilizada. Otra clasificación relevante es la proporcionada por **Stuart Russell** y **Peter Norvig** que dividen los sistemas de inteligencia artificial en cuatro grupos: **sistemas que piensan como humanos**, **sistemas que actúan como humanos**, **sistemas que piensan racionalmente** y **sistemas que actúan racionalmente** [5].

Para generar un sistema de inteligencia artificial se pueden utilizar un gran número de algoritmos y técnicas distintos, voy a explicar los que considero más relevantes:

Algoritmos de búsqueda, se utilizan cuando el problema se puede representar usando un árbol o grafo, pueden ser de **búsqueda no informada**, la búsqueda se basa en reglas impuestas previamente por el programador, o **informada o heurística**, que cuentan con funciones llamadas **heurísticas** que les permiten explorar el espacio de estados priorizando los nodos más prometedores. Algunos ejemplos de estos algoritmos son: **búsqueda de profundidad iterativa** y **búsqueda voraz**.

Algoritmos genéticos, se utilizan para resolver problemas de optimización complejos haciendo evolucionar una población de individuos que representan cada uno una solución. Esto se logra sometiéndolos a acciones aleatorias y seleccionándolos a través de una función de adaptación. Un ejemplo concreto de este tipo de algoritmos es la **regresión simbólica**.

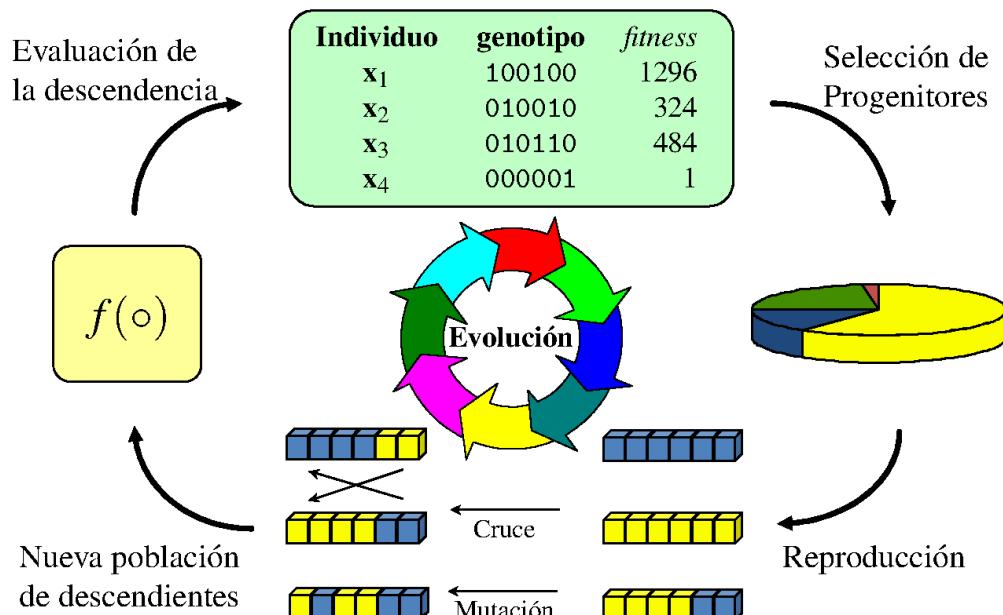


Figura 2-2. Algoritmos genéticos

Machine learning, engloba al conjunto de algoritmos que permite encontrar patrones en cantidades masivas de datos. Estos se dividen en: aprendizaje supervisado,

utiliza datos etiquetados, y aprendizaje no supervisado que no requiere datos etiquetados, algunos ejemplos son **Deep learning**, **k-means** o **XGBoost**. [6]

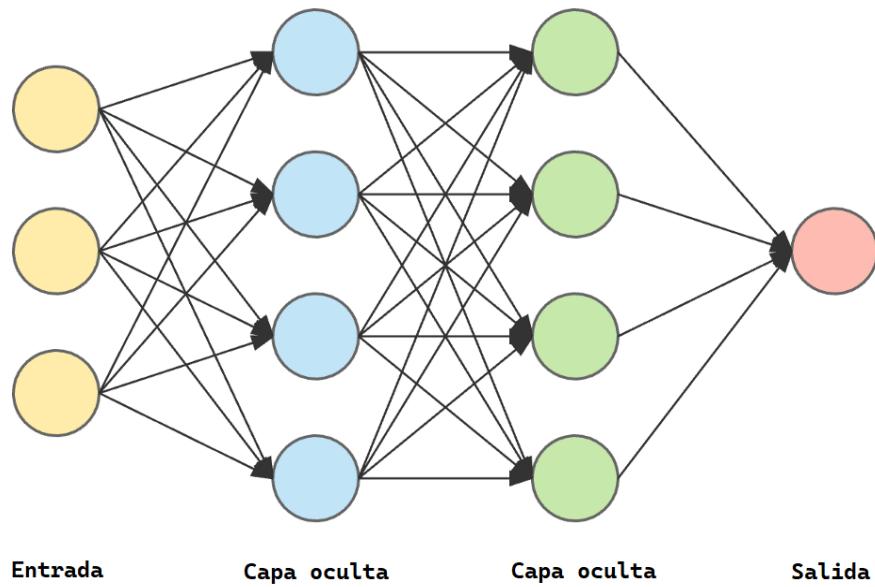


Figura 2-3. Redes Neuronales

Capítulo 3 - Explicación e Implementación de los Algoritmos

3.1 RandomForest

El algoritmo RandomForest fue desarrollado por Leo Breiman utilizando como referencias trabajos previos que había realizado y la selección aleatoria de atributos propuesta por Tin Kam Ho [7].

El funcionamiento de este algoritmo se basa en promediar los resultados de muchos árboles de decisión. Estos árboles se entrena cada uno usando solo un subconjunto de los casos de entrenamiento, usando el resto de los casos de entrenamiento para estimar el error. Cada nodo de estos árboles utiliza un subconjunto “m”, elegido aleatoriamente, de las variables en el clasificador y calcula la mejor partición del conjunto de entrenamiento utilizando esas “m” variables. La partición se calcula probando todos los umbrales posibles para todas las variables de “m” y eligiendo la combinación variable-umbral que minimiza la suma de las varianzas en las particiones resultantes. Esto continúa hasta alcanzar algún criterio de parada, como una profundidad máxima. Los nodos terminales o hojas contienen un conjunto de valores continuo. [8]

Las principales ventajas que presenta este algoritmo son: un menor riesgo de sobreajuste que los árboles de decisión, flexibilidad ya que puede usarse como regresor y clasificador y también es eficaz cuando faltan parte de los datos, y la facilidad para evaluar la importancia de cada una de las características. Los principales desafíos que enfrenta este algoritmo son su mayor complejidad que los árboles de decisión y su consumo de tiempo y recursos computacionales. [7]

3.1.1 Implementación

Para implementar RandomForest utilicé la librería RandomForestRegressor de sklearn. Lo primero que hice fue transformar los datos de un dataframe de panda a el formato requerido por las funciones fit y predict proporcionadas por la librería. Para ello cree una función con el nombre preparar_datosRandomForest cuyos parámetros son un

dataset df y el número de horas, numhoras, que se utilizan para hacer la predicción. El formato utilizado es X, una lista de listas, cada una de las listas contiene las características con las que se entrena el árbol e y, una lista con los resultados que tiene que obtener el árbol por cada entrada de X. Es decir que el primer miembro de y es el resultado que se debe obtener de predecir usando las características del primer miembro de X.

```
def preparar_datosRandomForest(df, numhoras):
    X = []
    y= []
    for i in range(0, df.shape[0] - numhoras):
        auxy = df.iloc[i + numhoras] #Seleccionamos la fila del precio que debe predecir
        y.append(auxy.close) #Agregamos el precio de cierre de la fila seleccionada a y
        aux1 = []
        for e in range(0, numhoras):
            aux = df.iloc[i + e] #Seleccionamos las filas de las que utilizaremos las características
            for r in range(1, aux.size):
                aux1.append(aux[r]) #Agregamos cada característica a la lista
        X.append(aux1) #Agregamos la lista a X
    return (X, y)
```

Figura 3-1. Preparar datos RandomForest

Tras esto implementé una función que permite evaluar el rendimiento de un modelo de RandomForest, evalRandomForest, cuyos parámetros son una lista con los valores reales y otra con los valores predichos. La función calcula el error porcentual medio de las predicciones y ese es el rendimiento del modelo, por lo tanto, cuanto más cercano a 0 más preciso es el modelo.

```
def evalRandomForest(Testrpr, predictT):
    suma = 0
    n = len(Testrpr)
    for i in range(0,n): # Suma el error relativo de todas las predicciones.
        suma = abs(predictT[i] - Testrpr[i])/Testrpr[i] + suma
    error_medio = suma/n # Divido la suma entre el número de predicciones para calcular la media
    emp = error_medio*100 # Multiplico por 100 para obtener el error porcentual medio
    return emp
```

Figura 3-2. Eval RandomForest

Tras esto implementé dos funciones para entrenar el algoritmo, la primera se llama train_randomForestdepth y recibe como argumentos: una lista de profundidades, un X y un y con datos de entrenamiento, un X y un y con datos para test y el numhoras a los que se corresponden esos datos. Su objetivo es, dada la lista de profundidades, entrenar un modelo de RandomForest utilizando cada una como profundidad máxima y evaluar

su rendimiento. Estos rendimientos se guardan en una lista que se devuelve junto con el mejor resultado y la profundidad a la que se obtuvo el mejor resultado.

```
def train_randomForestdepth(d_array, Xtr, ytr, Xvtaux, yvtaux, numhoras):
    resultados = []
    posbest = 0
    best = float('inf')
    for i in d_array:
        regri = RandomForestRegressor(max_depth=i, random_state=0) # Creo el RandomForest
        regri.fit(Xtr, ytr) # Entreno el RandoForest utilizando los X,y de entrenamiento
        predictI = regri.predict(Xvtaux) # Hago una predicción usando el modelo entrenado y el Xvtaux, que es el X reservado para testear
        valor = evalrandomForest(yvtaux, predictI) # Evalúo el rendimiento comparando la predicción con el valor real, que se encuentra en el yvtaux
        resultados.append({'numhoras': numhoras,'max_depth': i, 'valor': valor}) # Guardo los resultados del modelo
        if valor < best:
            best = valor
            posbest = i
            if valor < 0.8: # Si el rendimiento del modelo es mejor de un 0,8% guardo el modelo
                cadena = "Modelos/random_forest_model_h" + str(numhoras) + "_d" + str(i) + ".pkl"
                joblib.dump(regri, cadena)
    return(posbest, best, resultados)
```

Figura 3-3. Train RandomForest depth

La segunda función se llama `train_randomForest` y recibe como argumentos una lista de horas y una lista de profundidades, el objetivo de esta función es entrenar los modelos usando una cantidad de horas diferente cada vez, para ello crea los X e y correspondientes a la cantidad de horas de esa iteración y llama a `train_randomForestdepth` pasándoselos como argumento. Una vez finalizadas todas las iteraciones guarda los resultados en un dataframe, este se guarda con formato csv. La función devuelve el mejor resultado y la profundidad a la que se obtuvo.

```
def train_randomForest(h_array, d_array):
    resultados = []
    posbest = 0
    best = 100
    for i in h_array:
        Xtrain, ytrain = preparar_datosRandomForest(df_train, i) # Crear el X e y de entrenamiento para la cantidad de horas con las que se va a entrenar esta iteración
        Xvtaux, yvtaux = preparar_datosRandomForest(df_valitest, i) # Crear el X e y de test para la cantidad de horas con las que se va a entrenar esta iteración
        valores = train_randomForestdepth(d_array, Xtrain, ytrain, Xvtaux, yvtaux, i) # Llamada a la función train_randomForestdepth que devuelve los resultados para esa hora
        valor = valores[1]
        resultados.extend(valores[2]) # Añado los resultados obtenidos en esta iteración a los de las iteraciones anteriores
        print(str(i)+" "+str(valores[0])+" "+str(valor))
        with open('optimizaciónRandomForestIMC.txt', 'a') as archivo:
            # Escribo en el archivo la cantidad de horas, el mejor rendimiento obtenido y la profundidad a la que se ha obtenido
            archivo.write("Número de horas: "+str(i)+" Profundidad: "+str(valores[0])+" Valor de emp obtenido: "+str(valor) + "\n")
        if valor < best:
            best = valor
            posbest = valores[0]
    df_resultados = pd.DataFrame(resultados) # Transformo los resultados en un dataframe
    cadena = "Dataframes/resultados_randomForest.csv"
    df_resultados.to_csv(cadena, index=False) # Guardo el dataframe en formato csv
    return(posbest, best)
```

Figura 3-4. Train RandomForest

Finalmente cree la llamada a la función, la lista de horas que elegí fue: [1,2,3,4,5,6,7,8,9,10,14,18,20,25,30,40,50,75,100], y la lista de profundidades fue: [1,2,3,4,5,7,10,15,20,25,30,40,50,75,100,200,300,450,600].

```
tuple = train_randomForest([1,2,3,4,5,6,7,8,9,10,14,18,20,25,30,40,50,75,100], [1,2,3,4,5,7,10,15,20,25,30,40,50,75,100,200,300,450,600])
```

Figura 3-5. Llamada RandomForest

3.2 XGBoost

El algoritmo XGBoost fue desarrollado por Tianqi Chen como un proyecto de investigación, este se inspiró en el trabajo de Friedman. XGBoost supuso una versión de mejorada de los árboles de regresión con potenciación de gradiente (GBRT) de Friedman, los cuales desarrolló en 2001. XGBoost aumentó mucho su popularidad por ser el algoritmo elegido por varios equipos que obtuvieron victorias en competiciones de aprendizaje automático.

XGBoost fue originalmente desarrollado para C++, pero debido a su popularidad cuenta con implementaciones en los lenguajes mayoritariamente utilizados para el tratamiento de datos, como: Python, Java, R y Julia.

El algoritmo XGBoost es una evolución del Gradient Boosting, XGBoost combina árboles de decisión de forma secuencial, donde cada nuevo árbol corrige los errores del anterior. Por lo tanto, el objetivo de los árboles es reducir una función pérdida. Esta función pérdida crece en función del número de árboles y la complejidad de estos para así evitar un sobre ajuste. El algoritmo va añadiendo un árbol en cada iteración con el objetivo de minimizar la función de pérdida, XGBoost utiliza una expansión de Taylor de segundo orden para aproximar la función de pérdida facilitando así el cálculo de la ganancia o pérdida en cada iteración [9].

3.2.1 Implementación

Para implementar XGBoost utilice la librería de XGBoost disponible para Python. Lo primero que hice fue transformar mis datos de un dataframe de panda a el formato requerido por las funciones fit y predict proporcionadas por la librería. Para ello creo un conjunto de funciones, al primera de ellas se llama create_df_n, su objetivo es crear un dataframe acorde a la cantidad de horas que se utilizan en cada iteración.

```

def create_df_n(df, n): # Creo un nuevo dataframe con un formato adecuado para el XGBoost
    df_aux = df.copy()
    for i in range(1, n):
        # Añado en el dataframe que he creado los valores de hace n - 1 horas como columnas
        df_aux['open_before' + str(i)] = df_aux['open'].shift(+i) # Por ejemplo el valor de apertura de hace 2 horas se añade como open_before2
        df_aux['high_before' + str(i)] = df_aux['high'].shift(+i)
        df_aux['low_before' + str(i)] = df_aux['low'].shift(+i)
        df_aux['close_before' + str(i)] = df_aux['close'].shift(+i)
        df_aux['value_before' + str(i)] = df_aux['value'].shift(+i)
    df_aux['close_next'] = df_aux['close'].shift(-1) # Añado en el dataframe el valor de cierre de la siguiente hora
    df_aux = df_aux.dropna() # Elimino todas las filas que no tengan todos los valores
    return df_aux

```

Figura 3-6. Create_df_n XGBoost

El siguiente paso es dividir el dataframe en entrenamiento, validación y test, creando un dataframe para cada una de las tres cosas. Eso lo realizo a través de estas tres funciones.

```

def createdftrain(df_aux):
    tamano_aux = df_aux.shape[0] # Obtengo el tamaño del dataframe
    return df_aux.copy().iloc[0:int(tamano_aux*0.7)] # Selecciono el 70% inicial para ser el dataframe para training

Run Cell | Run Above | Debug Cell
# %%
def createdfvali(df_aux):
    tamano_aux = df_aux.shape[0] # Obtengo el tamaño del dataframe
    # Selecciono los datos entre el 70% y el 90% para ser el dataframe para validación
    return df_aux.copy().iloc[int(tamano_aux*0.7 + 1):int(tamano_aux*0.9)]

Run Cell | Run Above | Debug Cell
# %%
def createdftest(df_aux):
    tamano_aux = df_aux.shape[0] # Obtengo el tamaño del dataframe
    return df_aux.copy().iloc[int(tamano_aux*0.9 + 1):tamano_aux] # Selecciono el 10% final para ser el dataframe para test

```

Figura 3-7. Create dataframes XGBoost

El último paso prepara los tres dataframe en el formato adecuado para las funciones fit y predict y convertirlo en una DMatrix. Para ello cree la función llamada preparar_datosXGBoost.

```

def preparar_datosXGBoost(df_aux):
    X = df_aux.drop(['date', 'close_next'], axis=1) # Elimino la columna date y la columna close_next
    y = df_aux['close_next'] # Asigno a y los valores de la columna close_next
    return (xgb.DMatrix(data=X, label=y), y) # Convierto X en una matriz

```

Figura 3-8. Preparar_datos XGBoost

Tras preparar los datos, implementé una función que permite evaluar el rendimiento de un modelo de XGBoost, llamada evalXGB, cuyos parámetros son una lista con los valores reales y otra con los valores predichos. La función calcula el error porcentual medio de las predicciones y ese es el rendimiento del modelo, por lo tanto, cuanto más cercano a 0 más preciso es el modelo.

```

def evalXGB(Test_xgb, predict_xgb_test):
    suma = 0
    n = len(Test_xgb)
    for i in range(0,n):    # Suma el error relativo de todas las predicciones.
        suma = abs(predict_xgb_test[i] - Test_xgb[i])/Test_xgb[i] + suma
    error_medio = suma/n    # Divido la suma entre el número de predicciones para calcular la media
    emp = error_medio*100   # Multiplico por 100 para obtener el error porcentual medio
    return emp

```

Figura 3-9. Eval XGBoost

Tras esto implementé dos funciones para entrenar el algoritmo, la primera se llama train_XGB_depth y recibe como argumentos: una lista de profundidades, las DMatrix de entrenamiento, validación y test y el valor que debería predecirse al usar la DMatrix de test. Su objetivo es, dada la lista de profundidades, entrenar tres modelos de XGBoost, uno con eta 0,3, otro 0,1 y el último 0,01, utilizando cada una como profundidad máxima y evaluar su rendimiento. Estos rendimientos se guardan en una lista que se devuelve junto con el mejor resultado y la profundidad a la que se obtuvo el mejor resultado.

```

def train_XGB_depth(d_array, dtrainf, dvalif, dtestf, ytest):
    resultados = []
    best = float('inf') # Variable en la que guardo el mejor resultado
    best_depth = 0 # Variable en la que guardo la profundidad que genera el mejor resultado
    for i in d_array: # Genero un modelo para todas las profundidades
        etaAux = [0.3 , 0.1, 0.01] # Diferentes parámetro eta que vamos a utilizar
        for e in etaAux: # Genero un modelo para todas las etas
            param = { 'max_depth': i, 'eta': e, 'objective': 'reg:squarederror'} # Asigno los parámetros para el modelo
            evals = [(dtrainf, 'train'), (dvalif, 'validacion')] # Asigno los datos para entrenar y validar el modelo
            esr = int(1/e) # Calculo el parámetro early stopping rounds en función del valor de eta que este usando
            if(esr < 10): # Si el parámetro early stopping rounds es menor que 10 lo cambio a 10
                esr = 10
            # Entreno el modelo
            bstaux = xgb.train(param, dtrainf, num_boost_round=int(100/e), evals=evals, early_stopping_rounds=esr, verbose_eval=100)
            predict_xgb_test = bstaux.predict(dtestf) # Utilizo el modelo que acabo de entrenar para predecir usando los datos para test
            valor = evalXGB(ytest, predict_xgb_test) # Evalúo el rendimiento del modelo usando la predicción y los datos reales
            resultados.append({'max_depth': i, 'eta': param['eta'], 'valor': valor}) # GUARDO LOS RESULTADOS DEL RENDIMIENTO DE ESTE MODELO
            if(valor < best): # Si el rendimiento del modelo es menor que el mejor rendimiento hasta ahora lo asigno como el mejor
                best = valor
                best_depth = i # Guardo la profundidad del mejor modelo hasta el momento
                if best < 0.75: # Si el rendimiento es menor que 0.75 guardo el modelo
                    cadena = "Modelos/modelo_xgb_v.json" + str(valor)+ "_d" + str(i) + "_eta" + str(param.get("eta")) + ".json"
                    bstaux.save_model(cadena) # Guardo el modelo
    return (best_depth, best, resultados)

```

Figura 3-10. Train_XGB_depth XGBoost

La segunda función se llama trainGlobalXGB y recibe como argumentos una lista de profundidades y una lista de horas, el objetivo de esta función es entrenar los modelos usando una cantidad de horas diferente cada vez, para ello crea las DMatrix e y correspondientes a la cantidad de horas de esa iteración y llama a train_XGB_depth pasándoselos como argumento. Una vez finalizadas todas las iteraciones guarda los resultados en un dataframe, este se guarda con formato csv. La función devuelve el mejor resultado y la profundidad a la que se obtuvo.

```

def trainGlobalXGB(d_array, h_array):
    best = float('inf') # Variable en la que guardo el mejor resultado
    best_depth = 0 # Variable en la que guardo la profundidad que genera el mejor resultado
    for i in h_array:
        df_aux = create_df_n(df, i) # Creo el dataframe acorde al número de horas de esta iteración, i
        dtrain_aux = createdftrain(df_aux) # Creo el dataframe que voy a usar para el entrenamiento
        dvali_aux = createdfvali(df_aux) # Creo el dataframe que voy a usar para la validación
        dtest_aux = createdftest(df_aux) # Creo el dataframe que voy a usar para el test
        dtrain_prep = preparar_datosXGBoost(dtrain_aux) # Función con la que preparo los datos que voy a usar para el entrenamiento
        dvali_prep = preparar_datosXGBoost(dvali_aux) # Función con la que preparo los datos que voy a usar para la validación
        dtest_prep = preparar_datosXGBoost(dtest_aux) # Función con la que preparo los datos que voy a usar para el testeo
        # Entreno los modelos usando los datos preparados con la cantidad de horas correcta
        values = train.XGB_depth(d_array, dtrain_prep[0], dvali_prep[0], dtest_prep[0], dtest_prep[1].values)
        # Imprimo por pantalla el número de horas, la mejor profundidad de la iteración y el mejor rendimiento de la iteración
        print(str(i)+" "+str(values[0])+" "+str(values[1]))
    df_resultados = pd.DataFrame(values[2]) # Convierto los resultados del modelo en un dataframe
    cadena = "Dataframes/resultados_xgboost_h" + str(i) + ".csv"
    df_resultados.to_csv(cadena, index=False) # Guardo el dataframe
    with open('OptimizaciónXGBoostIMC.txt', 'a') as archivo:
        archivo.write("Número de horas: "+str(i)+" Profundidad: "+str(values[0])+" Valor de emp obtenido: "+str(values[1]) + "\n")
    if(values[1] < best): # Si el rendimiento obtenido durante el entrenamiento es menor que el mejor rendimiento hasta ahora lo asigno como el mejor
        best = values[1]
        best_depth = values[0] # Guardo la profundidad del mejor modelo hasta el momento
    return best, best_depth

```

Figura 3-11. TrainGlobalXGB XGBoost

Para finalizar cree la llamada a la función trainGlobalXGB, la lista de horas que elegí fue: [1,2,3,4,5,6,7,8,9,10,14,18,20,30,40,50,75,100], y la lista de profundidades fue: [1,2,3,4,5,6,7,8,9,10,15,20,30,40,50,75,100].

```
# Ejecuto la función principal trainGlobalXGB, la primera lista es la lista que contiene las profundidades y la segunda lista es la lista que contiene las horas
data = trainGlobalXGB([1,2,3,4,5,6,7,8,9,10,15,20,30,40,50,75,100],[1,2,3,4,5,6,7,8,9,10,14,18,20,30,40,50,75,100])
```

Figura 3-12. Llamada XGBoost

3.3 Redes Neuronales

En 1943 Warren McCulloch y Walter Pitts crearon un modelo informático para redes neuronales llamado lógica umbral, este modelo dividió la investigación de redes neuronales en dos enfoques distintos, uno enfocado en los procesos biológicos del cerebro y el otro en la aplicación a la inteligencia artificial.

Otro hito importante se produjo en 1940, psicólogo Donald Hebb formuló una hipótesis de aprendizaje basado en el mecanismo de plasticidad neuronal conocida como aprendizaje de Hebb. En 1948 se comenzó a aplicar a la computación, obteniéndose así una máquina de Turing de tipo B. [10]

En 1958 Frank Rosenblatt creó el perceptrón, lo que le permitió crear un algoritmo de clasificación, usando adición y sustracción simples. Parte de la circuitería diseñada por Frank Rosenblatt, circuito o-exclusiva, no pudo ser implementado hasta que en 1974 Paul Werbos creó el algoritmo de propagación hacia atrás. [11] Este algoritmo es una aplicación eficaz de la regla de la cadena a redes de nodos diferenciables, que fue derivada por Gottfried Wilhelm Leibniz en 1673.

El funcionamiento de las redes neuronales depende de la estructura y capacidades de las neuronas que las componen, una neurona básica de una capa intermedia cuenta con la siguiente estructura: una o más entradas las cuales son recibidas en un vector que llamaremos x , sus respectivos pesos también en un vector que llamaremos w y un sesgo que llamaremos b . La neurona efectúa una suma ponderada, z , usando estos elementos: $z = (w \cdot x + b)$. La salida de la neurona será el resultado de aplicar una función de activación [12], ϕ , al resultado de la suma ponderada. El propósito de la función de activación es introducir no linealidad para permitir el aprendizaje de patrones complejos que no sean lineales. Algunos ejemplos de estas funciones serían la función sigmoide, softmax o unidad lineal rectificada, ReLU.

Las neuronas se estructuran en capas, la primera capa de la red se llama capa de entrada, la última se llama capa de salida y las que están entre ambas se denominan capas ocultas. El número de capas ocultas varía desde 0 hasta varios miles de capas. El número de neuronas en cada capa también varía desde 1 hasta varios miles.

El proceso de entrenamiento del modelo consiste en proporcionar una o varias entradas, calcular el error entre la salida que proporciona el modelo y el valor real y ejecutar el algoritmo de propagación hacia atrás para propagar el error y así actualizar el vector de pesos. Esto se hace cada x número de entradas, este número se denomina `batch_size`, y se hace con todos los datos n veces, este n se denomina `num_epoch`.

Tras finalizar el proceso de entrenamiento, para hacer una predicción simplemente hay que proporcionar una entrada al modelo, la salida será la predicción.

3.3.1 Implementación

Para implementar la red neuronal utilicé la librería de keras de Tensorflow. Lo primero que hice fue transformar los datos de un dataframe de panda a el formato requerido por las funciones `fit` y `predict` proporcionadas por la librería. Para ello cree dos funciones una con el nombre `preparar_datos` cuyos parámetros son un dataset `df` y el número de horas, `numhoras`, que se utilizan para hacer la predicción y otra llamada `create_sequences`. El formato utilizado es `X`, una lista de listas, `e` y una lista de valores, cada entrada de `e` se corresponde con el valor que se debe predecir usando esa misma entrada de `X`.

```

def create_sequences(data, n_steps):
    X, y = [], []
    for i in range(len(data) - n_steps):
        X.append(data[i:i+n_steps])      # Añado a X las listas de precios
        y.append(data[i+n_steps, 3])     # Añado a y el precio de cierre de la siguiente hora
    return np.array(X), np.array(y)      # Los transformo en arrays de numpy

```

Figura 3-13. Create_sequences redes neuronales

```

def preparar_datos(df, numhoras):
    data = df[['open', 'high', 'low', 'close', 'value']].values # Selecciono y transformo las columnas del dataframe en un array
    # Genero una matriz de listas, cada entrada de la matriz contiene n listas del tipo [open, high, low, close, value] siendo n = numhoras
    X, y = create_sequences(data, numhoras)
    X_aux = []
    for i in X: # Transformo la matriz de listas en una lista de listas, juntando las n listas de cada entrada en una
        aux = []
        for r in range(0, numhoras): # Transformo las n listas de esa entrada en una
            for elem in i[r]:
                aux.append(elem)
        X_aux.append(aux) # Añado la lista transformada a la lista global
    X_aux = np.array(X_aux) # Transformo la lista en un array de numpy
    return X_aux, y

```

Figura 3-14. Preparar_datos redes neuronales

Tras preparar los datos, implementé una función que permite evaluar el rendimiento de un modelo de redes neuronales, llamada evalRedDensa, cuyos parámetros son una lista con los valores reales y otra con los valores predichos. La función calcula el error porcentual medio de las predicciones y ese es el rendimiento del modelo, por lo tanto, cuanto más cercano a 0 más preciso es el modelo.

```

def evalRedDensa(ytest, y_pred):
    y_pred = y_pred.flatten() # Transforma una lista de listas en una lista de valores
    suma = 0
    n = len(y_pred) # Obtengo el tamaño de y
    for i in range(0,n): # Suma el error relativo de todas las predicciones.
        suma = abs(y_pred[i] - ytest[i])/ytest[i] + suma
    error_medio = suma/n # Divido la suma entre el número de predicciones para calcular la media
    emp = error_medio*100 # Multiplico por 100 para obtener el error porcentual medio
    return emp

```

Figura 3-15. Eval redes neuronales

Una vez había preparado los datos, definí la estrategia de entrenamiento que iba a usar la red.

```

strategy = tf.distribute.MirroredStrategy() # Replica tu modelo en todas las GPUs disponibles
print(f"Número de GPUs detectadas: {strategy.num_replicas_in_sync}")

```

Figura 3-16. Estrategia redes neuronales

Tras esto, implementé dos funciones para entrenar el algoritmo, la primera se llama opti_redes_densas_multi_gpu y recibe como argumentos: una lista de epoch, una lista de batch_size, el número de horas y las listas X e y de entrenamiento, validación y test. Su objetivo es, dada la lista de epoch y la lista de batch_size, entrenar quince redes

neuronales por cada posible par de valores que se obtiene de combinar ambas listas y evaluar su rendimiento. Estos rendimientos se guardan en una lista que se devuelve junto con el mejor resultado y la profundidad a la que se obtuvo el mejor resultado.

```

def opti_redes_densas_multi_gpu(epoch_array, batch_array, numhoras, X_train, y_train, X_vali, y_vali, X_test, y_test):
    best = float('inf') # Asigno a best un valor infinito de tipo float
    epoch_best = 0
    batch_best = 0
    best_model = None
    training_results = []

    for e in epoch_array: # Genero modelos con todos los epoch pasados como parámetro en el array
        for b in batch_array: # Genero modelos con todos los batch_size pasados como parámetro en el array
            best_value_of_the15 = float('inf') # Asigno a best_value_of_the15 un valor infinito de tipo float
            best_model_of_the15 = None
            with tf.device('/CPU:0'):
                for m in range(15): # Genero 15 modelos con cada de par epoch, batch_size
                    with strategy.scope(): # El código debajo de esta línea se ejecuta usando la estrategia que hay en el scope
                        model = Sequential() # Declaro el modelo como secuencial
                        # Creo la capa de entrada con 128 neuronas, definiendo la forma de la entrada y cuya función de activación es Rectified Linear Unit
                        model.add(Dense(128, activation='relu', input_shape=(numhoras + 5,)))
                        # Creo la capa interna con 64 neuronas y cuya función de activación es Rectified Linear Unit, que selecciona el másximo entre 0 y el valor de la neurona
                        model.add(Dense(64, activation='relu'))
                        # Creo la capa de salida
                        model.add(Dense(1))
                        # Compilo la red neuronal usando adam como optimizador y mape, Mean Absolute Percentage Error, como función que debe minimizar
                        model.compile(optimizer='adam', loss='mape')

                    history = model.fit(X_train, y_train, epochs=e, batch_size=b, validation_data=(X_vali, y_vali), shuffle=False) # Entreno el modelo
                    y_pred = model.predict(X_test) # Realizo una predicción usando los datos de test
                    valor = evalRedDensa(y_test, y_pred) # Evalúo el rendimiento del modelo

                    if valor < best_value_of_the15: # Si el rendimiento que obtengo es el mejor de esta iteración hasta ahora lo sustituyo y guardo el modelo
                        best_value_of_the15 = valor
                        best_model_of_the15 = model

            print(f"Epoch: {e}, Batch size: {b}, Value: {best_value_of_the15}") # Imprimo por pantalla la epoch, el batch_size y el mejor rendimiento obtenido para estos
            training_results.append({"epoch": e, "batch_size": b, "hours": numhoras, "value": best_value_of_the15})

    with open('pasosdados.txt', 'w') as archivo:
        archivo.write("epoch: " + str(e) + ", batch_size:" + str(b)) # Escribo en un archivo de texto la epoch y el batch_size para los que acabo de entrenar el modelo
    if best_value_of_the15 < best: # Si el rendimiento que he obtenido en la iteración es el mejor hasta ahora lo sustituyo
        best = best_value_of_the15
        epoch_best = e # Guardo su epoch
        batch_best = b # Guardo su batch_size
        best_model = best_model_of_the15 # Guardo el modelo en la variable
        if best < 0.75: # Si el rendimiento es mejor que 0,75, guardo el modelo
            cadena_guardado = f"ModelosDensosOptiMultiGPU{m}_mi_modelo_densoIMC_Opti_e{e}_b{b}_v{round(best, 3)}_nh{numhoras}"
            best_model.save(cadena_guardado + ".keras") # Guardo el modelo

    results_df = pd.DataFrame(training_results) # Transformo los datos guardados en un dataframe
    cadena = "densasSH" + str(numhoras) + ".csv"
    results_df.to_csv(cadena, index=False) # Guardo el dataframe con formato csv
    print("Resultados guardados en 'densas.csv'") # Imprimo por pantalla un mensaje que indica que el dataframe ha sido guardado
    return epoch_best, batch_best, best, best_model

```

Figura 3-17. Opti_redes_densas redes neuronales

La segunda función se llama opti_rd_h y recibe como argumentos una lista de horas, una lista de epoch y una lista de batch_size, el objetivo de esta función es entrenar las redes neuronales usando una cantidad de horas diferente cada vez, para ello crea las listas X e y correspondientes a la cantidad de horas de esa iteración y llama a opti_redes_densas_multi_gpu pasándoselos como argumento. La función devuelve el mejor resultado, el epoch, batch_size y número de horas con los que se obtuvo y el modelo que lo produjo.

```

def opti_rd_h(h_array, epoch_array, batch_array):
    best = float('inf') # Asigno a best un valor infinito de tipo float
    epoch_best = 0
    batch_best = 0
    h_best = 0
    best_model = None
    for i in h_array: # Entreno modelos con el numero de horas en el array de horas
        Xtrain, ytrain = preparar_datos(df_train, i) # Preparo los datos de entrenamiento
        Xvali, yvali = preparar_datos(df_vali, i) # Preparo los datos de validación
        Xtest, ytest = preparar_datos(df_test, i) # Preparo los datos de test
        # Entreno los modelos usando los datos preparados con la cantidad de horas correcta
        valores = opti_redes_densas.multi_gpu(epoch_array, batch_array, i, Xtrain, ytrain, Xvali, yvali, Xtest, ytest)
        if valores[2] < best: # Si el rendimiento que he obtenido en la iteración es el mejor hasta ahora lo sustituyo
            best = valores[2]
            epoch_best = valores[0] # Guardo su epoch
            batch_best = valores[1] # Guardo su batch_size
            h_best = i # Guardo el número de horas
            best_model = valores[3] # Guardo el modelo en la variable
            cadena_guardado = "ModelosDensosOptiMoreDataIMCBest/mi_modelo_densoIMC_Opti_e"+str(epoch_best)+"_b"+str(batch_best)+"_h"+str(i)+"_v"+str(round(best, 3))+"_nh"+str(i)
            best_model.save(cadena_guardado+".keras") # Guardo el modelo
            with open('pasosdoshoras.txt', 'w') as archivo:
                archivo.write("horas: "+str(i)+"\n") # Escribo en un archivo de texto la hora para la que acabo de entrenar los modelos
    return best, epoch_best, batch_best, h_best, best_model # Devuelvo el mejor rendimiento, epoch, batch_size, hora y modelo

```

Figura 3-18. Opti_rd_h redes neuronales

Para finalizar cree la llamada a la función opti_rd_h, la lista de horas que elegí es [1, 3, 5, 7, 10, 12, 14, 18, 21], la lista de epoch es [4, 6, 10, 14, 20, 40] y la lista de batch_size es [4, 8, 12, 16, 32, 64, 128, 256]. Al finalizar imprime el resultado obtenido y un mensaje indicando que ha terminado.

```

# Llamo a la función de entrenamiento principal, la primera es la lista con las horas, al segunda es la lista con las epoch y la tercera es la lista de los batch_size
data = opti_rd_h([1, 3, 5, 7, 10, 12, 14, 18, 21], [4, 6, 10, 14, 20, 40], [4, 8, 12, 16, 32, 64, 128, 256])
print(data) # Imprimo lo que devuelve la función
print("Ha terminado") # Imprimo un mensaje indicando que la ejecución ha concluido

```

Figura 3-19. Llamada redes neuronales

3.4 Redes Neuronales LSTM

Las redes LSTM fueron inventadas por Sepp Hochreiter en colaboración con Jürgen Schmidhuber en el año 1995. Previamente en 1991, Hochreiter, en su tesis, formalizó el problema de colapso del gradiente, vanishing gradient, y en 1993 desarrolló la primera red muy profunda, precursora de las LSTM, en la que introdujo conexiones residuales recurrentes para evitar el colapso del gradiente.

Sin embargo, las redes LSTM de 1995 aún no tenían la misma estructura de las modernas, esta arquitectura fue desarrollada por Felix Gers en colaboración con Schmidhuber, la innovación que fue introducida fue el incluir una puerta de olvido.

A partir del año 2006 las redes LSTM comenzaron a superar al resto de modelos en reconocimiento de voz y a partir del año 2014 comenzaron a dominar en campos como el procesamiento de lenguaje y la descripción de imágenes. [13]

El proceso de entrenamiento es similar al ya explicado para las redes neuronales clásicas.

La principal diferencia entre las redes neuronales clásicas y las redes neuronales LSTM es la arquitectura de sus neuronas. En el caso de las neuronas LSTM, estas están compuestas de una memoria central y tres puertas, que trabajan conjuntamente para controlar el flujo de información.

La puerta de olvido se encarga de decidir qué información se descarta de la memoria a largo plazo. Para ello utiliza esta fórmula: $f_t = \sigma (W_f \cdot x_t + U_f \cdot h_{t-1} + b_f)$, la salida f estará comprendida entre 0 y 1, si la salida está más próxima a 0 se olvida la información, si está más próxima a 1 se conserva. W se corresponde con los pesos de entrada y U con el vector de pesos de las conexiones recurrentes, h con el estado oculto, x con la entrada, b con el sesgo y σ con la función de activación sigmoide. [14]

La puerta de entrada se encarga de decidir qué parte de la información se almacena. Para ello se utilizan dos ecuaciones, la primera para controlar la capa de entrada: $i_t = \sigma (W_i \cdot x_t + U_i \cdot h_{t-1} + b_i)$, la salida i estará comprendida entre 0 y 1, si la salida está más próxima a 0 no se actualiza la información, si está más próxima a 1 se actualiza. W se corresponde con los pesos de entrada y U con el vector de pesos de las conexiones recurrentes, h con el estado oculto, x con la entrada, b con el sesgo y σ con la función de activación sigmoide. La segunda controla la activación de la entrada de la célula: $\sim c_t = \sigma (W_c \cdot x_t + U_c \cdot h_{t-1} + b_c)$, la salida c está comprendida entre -1 y 1, si la salida es 0 la información es irrelevante si es 1 o -1 es relevante. La nomenclatura es la misma que para la ecuación anterior. [14]

La puerta de salida se encarga de decidir qué parte de la información es transmitida $o_t = \sigma (W_o \cdot x_o + U_o \cdot h_{t-1} + b_o)$, la salida o estará comprendida entre 0 y 1, si la salida está más próxima a 0 no se permite la salida de la información, si está más próxima a 1 se permite. La nomenclatura es la misma que en las ecuaciones anteriores. Otra ecuación de la puerta de salida es la ecuación del vector del estado oculto: $h_t = (o_t \odot \sigma_h \cdot c_t)$, la salida h está comprendida entre -1 y 1, si la salida es 0 la información es irrelevante si es 1 o -1 es relevante. El símbolo \odot hace referencia al producto de Hadamard. [14]

La memoria central también posee una ecuación asociada, esta ecuación calcula el vector estado de la neurona: $c_t = (f_t \odot c_{t-1} + i_t \odot \sim c_t)$, la salida se encontrará dentro de los reales. [14]

3.4.1 Implementación

Para implementar la red neuronal utilicé la librería de keras de Tensorflow. Lo primero que hice fue transformar los datos de un dataframde de panda a el formato requerido por las funciones fit y predict proporcionadas por la librería. Para ello cree dos funciones una con el nombre preparar_datos cuyos parámetros son un dataset df y el número de horas, numhoras, que se utilizan para hacer la predicción; y otra llamada create_sequences que recibe como parámetros los datos en un array np y el número de horas. El formato utilizado es X, una lista de listas de listas, e y una lista de valores, cada entrada de y se corresponde con el valor que se debe predecir usando esa misma entrada de X.

```
def create_sequences(data, n_steps):
    X, y = [], []
    for i in range(len(data) - n_steps):
        X.append(data[i:i+n_steps])      # Añado a X las listas de precios
        y.append(data[i+n_steps, 3])      # Añado a y el precio de cierre de la siguiente hora
    return np.array(X), np.array(y)      # Los transformo en arrays de numpy
```

Figura 3-20. Create_sequences LSTM

```
def preparar_datosLSTM(df, numhoras):
    # Llamo a la función create_sequences con los valores del df en un array de numpy y el número de horas
    return create_sequences(df[['open', 'high', 'low', 'close', 'value']].to_numpy(), numhoras)
```

Figura 3-21. Preparar_datos LSTM

Tras preparar los datos, implementé una función que permite evaluar el rendimiento de un modelo de redes neuronales, llamada evalRedLSTM, cuyos parámetros son una lista con los valores reales y otra con los valores predichos. La función calcula el error porcentual medio de las predicciones y ese es el rendimiento del modelo, por lo tanto, cuanto más cercano a 0 más preciso es el modelo.

```

def evalRedLSTM(ytest, y_pred):
    y_pred = y_pred.flatten() # Transforma una lista de listas en una lista de valores
    suma = 0
    n = len(y_pred) # Obtengo el tamaño de y
    for i in range(0,n): # Suma el error relativo de todas las predicciones
        suma = abs(y_pred[i] - ytest[i])/ytest[i] + suma
    error_medio = suma/n # Divido la suma entre el número de predicciones para calcular la media
    emp = error_medio*100 # Multiplico por 100 para obtener el error porcentual medio
    return emp

```

Figura 3-22. Eval LSTM

Una vez había preparado los datos, implementé dos funciones para entrenar el algoritmo, la primera se llama opti_redes_LSTM y recibe como argumentos: una lista de epoch, una lista de batch_size, el número de horas y las listas X e y de entrenamiento, validación y test. Su objetivo es, dada la lista de epoch y la lista de batch_size, entrenar quince redes neuronales LSTM por cada posible par de valores que se obtiene de combinar ambas listas y evaluar su rendimiento. Estos rendimientos se guardan en una lista que se devuelve transformada en un dataframe de pandas.

```

def opti_redes_LSTM(epoch_array, batch_array, X_trainLSTM, y_trainLSTM, X_valiLSTM, y_valiLSTM, X_testLSTM, y_testLSTM, numhoras):
    best_model = None
    best = float('inf') # Asigno a best un valor infinito de tipo float
    results = []
    for e in epoch_array: # Genero modelos con todos los epoch pasados como parámetro en el array
        for b in batch_array: # Genero modelos con todos los batch_size pasados como parámetro en el array
            best_value_of_the15 = float('inf') # Asigno a best_value_of_the15 un valor infinito de tipo float
            for l in range(15): # Genero 15 modelos con cada de par epoch, batch_size
                with tf.device('/CPU:0'): # Fuerzo que el código debajo de esta línea se ejecute usando la CPU
                    modelLSTM = Sequential() # Declaro el modelo como secuencial
                    # Creo la capa de entrada con 128 neuronas, definiendo la forma de la entrada y y cuya función de activación es Rectified Linear Unit
                    modelLSTM.add(LSTM(128, activation='relu', input_shape=(numhoras, 5)))
                    # Creo la capa de salida
                    modelLSTM.add(Dense(1))
                    # Compilo la red neuronal usando adam como optimizador y mape, Mean Absolute Percentage Error, como función que debe minimizar
                    modelLSTM.compile(optimizer='adam', loss='mape')
                    # Entreno el modelo
                    historyLSTM = modelLSTM.fit(X_trainLSTM, y_trainLSTM, epochs=e, batch_size=b, validation_data=(X_valiLSTM, y_valiLSTM), shuffle=False)
                    # Realizo una predicción usando los datos de test
                    y_pred = modelLSTM.predict(X_testLSTM)
                    valor = evalRedLSTM(y_testLSTM, y_pred) # Evalúo el rendimiento del modelo
                    print("epoch: "+str(e)+", batch_size: "+str(b)+" , value: "+str(valor)) # Imprimo la epoch, el batch_size y el rendimiento obtenido
                    if valor < best_value_of_the15: # Si el rendimiento que obtengo es el mejor de esta iteración hasta ahora lo sustituyo
                        best_value_of_the15 = valor
                        if valor < best: # Si el rendimiento que he obtenido es el mejor hasta ahora lo sustituyo y guardo el modelo en la variable best_model
                            best_model = modelLSTM
                            best = valor
                            if valor < 0.75: # Si el rendimiento es mejor que 0,75 guardo el modelo
                                # Creo el nombre del modelo y lo guardo en la variable cadena_guardado
                                cadena_guardado = "ModelosLSTMoptiMoreDataIMC/mi_modelo_LSTMopti_e"+str(e)+"_b"+str(b)+"_v"+str(round(valor, 3))
                                best_model.save(cadena_guardado+".keras") # Guardo el modelo
                    results.append([e, b, best_value_of_the15]) # Guardo la información de la epoch, el batch_size y el mejor rendimiento obtenido para estos
    df_results = pd.DataFrame(results, columns=["epoch", "batch_size", "value"]) # Transformo los datos guardados en un dataframe
    return df_results # Devuelvo el dataframe

```

Figura 3-23. Opti_redes LSTM

La segunda función se llama opti_rlSTM_h y recibe como argumentos una lista de horas, una lista de epoch y una lista de batch_size, el objetivo de esta función es entrenar las redes neuronales usando una cantidad de horas diferente cada vez, para ello crea las listas X e y correspondientes a la cantidad de horas de esa iteración y llama a

opti_redes_LSTM pasándoselos como argumento. La función guarda los dataframes recibidos opti_redes_LSTM y devuelve un mensaje diciendo que ha terminado.

```
def opti_rLSTM_h(h_array, epoch_array, batch_array):
    for i in h_array:
        Xtrain, ytrain = preparar_datosLSTM(df_train, i)      # Preparo los datos en el formato requerido para el entrenamiento
        Xvali, yvali = preparar_datosLSTM(df_vali, i)        # Preparo los datos en el formato requerido para la validación
        Xtest, ytest = preparar_datosLSTM(df_test, i)        # Preparo los datos en el formato requerido para el test
        # Guardo en df el dataframe que decuelve
        df = opti_redes_LSTM(epoch_array, batch_array, Xtrain, ytrain, Xvali, yvali, Xtest, ytest, i)
        csv_filename = "LstmH" + str(i) + ".csv"            # Guardo en csv_filename el nombre con el que voy a guardar el dataframe de df
        df.to_csv(csv_filename, index=False)     # Guardo el dataframe
    return "He terminado"
```

Figura 3-24. Opti_rLSTM_h LSTM

Para finalizar cree la llamada a la función opti_rLSTM_h, la lista de horas que elegí es [1, 3, 5, 7, 10, 12, 14, 18, 21], la lista de epoch es [4, 6, 10, 14, 20, 40] y la lista de batch_size es [4, 8, 12, 16, 32, 64, 128, 256].

```
# Llamo a la función de entrenamiento principal, la primera es la lista con las horas, al segunda es la lista con las epoch y la tercera es la lista de los batch_size
data = opti_rLSTM_h([10, 12, 14, 18, 21], [4, 6, 10, 14, 20, 40], BATCH_ARRAY)
```

Figura 3-25. Llamada LSTM

3.5 Regresión Simbólica

Las investigaciones en regresión simbólica comenzaron con trabajos pioneros como el de Gerwin en 1974 o el de Langley en 1981 cuyo objetivo era redescubrir leyes empíricas. [15]

El primero en proponer el uso de Programación genética fue John Koza en 1989, proponiendo codificar las expresiones matemáticas como arboles computacionales y modificar la población inicial iterativamente con operaciones basadas en procesos biológicos. [15]

La regresión simbólica ha ganado popularidad gracias al aumento de la capacidad computacional. Sus principales áreas de aplicación son la medicina, química y ciencia de materiales. La principal ventaja que posee la regresión simbólica ante otros algoritmos y una de las razones por las que es ampliamente utilizada en disciplinas científicas es su explicabilidad.

El funcionamiento de la regresión simbólica es el mismo que el de un algoritmo genético, sin embargo, los genes son ecuaciones. Durante cada iteración se evalúa el

rendimiento de cada gen y, si ninguno ha alcanzado el rendimiento objetivo, se procede a mutarlos y aplicar otras técnicas para intentar mejorar el rendimiento en la iteración siguiente. Este proceso se repite hasta que se llega al rendimiento objetivo o hasta que se llega un número máximo de iteraciones. [16]

3.5.1 Implementación

Realicé dos implementaciones distintas para la regresión simbólica, la primera no utiliza constantes, solo los datos de precios, la segunda si utiliza constantes. Discutiré primero la que no utiliza constantes.

3.5.1.1 Sin Constantes

La base sobre la que implementé el algoritmo es la definición que elegí para la estructura de los genes, estos son listas que empiezan y terminan con un número y que van alternando una operación con un número. Un ejemplo de esto sería: [1, '+, 7, '/', 2]. Los números representan una posición en otra lista. Esta otra lista contiene los valores de los precios de apertura, máximo, mínimo y cierre de las horas anteriores en este orden: Por ejemplo, si la lista fuera de las dos horas anteriores al precio que queremos predecir, la lista sería de esta manera: [apertura h-2, máximo h - 2, ..., cierre h - 1]. El gen puesto de ejemplo se traduciría como: [máximo h - 2 , '+, cierre h - 1, '/', mínimo h-2].

Para implementar el algoritmo cree una clase llamada RegresionSimbolica, esta clase cuenta con un constructor en el que se definen sus principales atributos, siendo estos: función de optimización, la cual evalúa el rendimiento de cada gen; la lista de operaciones, esta es una lista de listas en la que se encuentran las operaciones según su prioridad, es decir en la primera lista estarán las operaciones más prioritarias; el tamaño máximo que puede alcanzar un gen, el tamaño mínimo que puede tener un gen, una lista llamada genes que contiene todos los genes y un parámetro n que se corresponde con la cantidad de horas anteriores que pueden utilizarse para hacer predicciones.

```

def __init__(self, funcionOptimizacion: callable, operations: list[list], maxSize: int, minSize: int, n : int):
    # Asigno a funcionOptimizacion la función que se debe utilizar para calcular el rendimiento de los genes
    self.funcionOptimizacion = funcionOptimizacion
    self.operations = operations # Asigno a operations la lista de listas de operaciones
    self.maxSize = maxSize # Asigno a maxSize el tamaño máximo que puede tener un gen
    self.minSize = minSize # Asigno a minSize el tamaño mínimo que puede tener un gen
    self.genes: list[list] = [] # Asigno a genes una lista vacía
    self.n = n # Asigno a n la cantidad de horas anteriores que se tienen en cuenta
    pass

```

Figura 3-26. Init regresión simbólica

Tras esto implementé los métodos encargados de crear los genes de la población inicial, primero declaré un método `create` que recibe como parámetro el número de genes, si este es menor que 1 levanta una excepción, si no llama al método `__create_priv`.

```

def create(self, numGenes: int):
    if numGenes > 0:      # Compruebo que el número de genes sea al menos 1
        return self.__create_priv(4*self.n - 1, numGenes)
    else:
        # Levanto una excepción porque el número de genes es menor que 1
        raise Exception("El número de genes no puede ser menor que 1")

```

Figura 3-27. Create regresión simbólica

El método `__create_priv` recibe como parámetros `r`, el número de posiciones disponibles, y el número de genes. Para generar cada gen el método escoge un número aleatorio comprendido entre el tamaño mínimo y el tamaño máximo y va escogiendo una operación y un número aleatorios entre los posibles.

```

def __create_priv(self, r : int, numGenes: int):
    genes = []
    for j in range (numGenes):
        aux = []
        aux.append(random.randint(0, r)) # Añado al gen un número aleatorio entre 0 y la posición máxima
        # Genero un tamaño aleatorio para el gen comprendido entre el tamaño máximo y el tamaño mínimo
        for i in range(random.randint(self.minSize, self.maxSize - 1)):
            # Selecciono aleatoriamente una categoría de operación
            numLO = random.randint(0, len(self.operations) - 1)
            # Añado al gen una operación elegida aleatoriamente de entre las de la categoría
            aux.append(self.operations[numLO][random.randint(0, len(self.operations[numLO]) - 1)])
            # Añado al gen un número aleatorio entre 0 y la posición máxima
            aux.append(random.randint(0, r))
        genes.append(aux) # Añado el gen generado a la lista de genes
    return genes

```

Figura 3-28. Create_priv regresión simbólica

Una vez implementados los métodos para generar la primera población de genes, implementé los métodos encargados de evaluar su rendimiento. Para ello

implementé tres métodos, fitness2 cuyos parámetros son: X una lista de listas que contiene las listas de valores de precios de horas anteriores, y una lista que contiene los valores que se deben predecir usando las entradas de X, y el gen que queremos evaluar. Este método se encarga de llamar al método `_evaluate` con el candidato y cada entrada de y. Tras esto aplica la función de optimización utilizando los valores predichos con `evaluate` y los valores reales del parámetro y.

```
def fitness2(self, X, y, candidato: list):
    valores_generados = []
    for elem in X: # Recorro X obteniendo la predicción del modelo para cada entrada de X
        valores_generados.append(self._evaluate(elem, candidato))
    # Ejecuto al función de optimización en al predicción e y
    return self.funcionOptimizacion(valores_generados, y)
```

Figura 3-29. Fitness2 regresión simbólica

El método `_evaluate` recibe como parámetros la lista de valores y el gen a evaluar. Su función es sustituir el valor correspondiente en cada posición del gen y realizar las operaciones indicadas en el gen con el fin de obtener una predicción. Para realizar las operaciones recorre los elementos impares de la lista y si la operación está en la categoría de operaciones de esa iteración se realiza llamando al método `_aplica_operacion` con los valores adyacentes y se deposita el resultado en la posición del primer operando.

```
def __evaluate(self, valores: list[float], candidato: list):
    candidato_aux = []
    # Recorro el gen y sustituyo las posiciones por sus valores correspondientes creando una lista auxiliar
    for i in range(0, len(candidato) - 1, 2):
        candidato_aux.append(valores[candidato[i]]) # Añado el valor correspondiente a esa posición
        candidato_aux.append(candidato[i+1]) # Añado la operación
    candidato_aux.append(valores[candidato[len(candidato) - 1]]) # Añado el valor correspondiente a esa posición
    # Recorro las categorías de operaciones ejecutando las operaciones de cada categoría
    for categoria_operaciones in self.operaciones:
        j_offset = 0 # offset del índice
        # Recorro las operaciones del gen
        for j in range(1, len(candidato_aux), 2):
            indice = j + j_offset
            op = candidato_aux[indice] # Asigno la operación a op
            # Compruebo si la operación esta en el categoría de operaciones de esta iteración
            if(op in categoria_operaciones):
                # Realizo la operación
                op_result = self._aplica_operacion( op, candidato_aux[indice-1], candidato_aux[indice+1])
                candidato_aux[indice-1] = op_result # Asigno el resultado de la operación al lugar del operador izquierdo
                del candidato_aux[indice+1] # Elimino la operación del gen
                del candidato_aux[indice] # Elimino el operador derecho del gen
                j_offset = j_offset - 2 # Reduzco en 2 el offset
    return candidato_aux[0] # Devuelvo el valor obtenido
```

Figura 3-30. Evaluate regresión simbólica

El método `__aplica_operacion` recibe como parámetros la operación a aplicar como un string, y los valores que debe usar i, j. Utiliza `if_else` anidados para aplicar la operación correspondiente y devolver el valor resultante.

```
def __aplica_operacion(self, op, i, j):
    if op == '+': # Compruebo si la operación es la suma
        return i + j # Sumo i más j
    elif op == '-': # Compruebo si la operación es la resta
        return i - j # Resto i menos j
    elif op == '*': # Compruebo si la operación es la multiplicación
        return i * j # Multiplico i por j
    elif op == '/': # Compruebo si la operación es la división
        return i / j if j != 0 else 1e6 # Si j es distinto de 0 divido i entre j
    elif op == '^': # Compruebo si la operación es la potencia
        return i ** j # Elevo i a j
```

Figura 3-31. Aplica_operaciones regresión simbólica

También implementé el método `display`, encargado de devolver un gen como string.

```
def display(self, candidato: list):
    # Decuelvo el gen como string
    return ' '.join(map(str, [i for i in candidato]))
```

Figura 3-32. Display regresión simbólica

Tras esto implementé el método encargado de realizar mutaciones a los genes. Lo llame `mutate2` y recibe como parámetro el gen a mutar. El método añade o elimina un número y una operación del gen o lo deja tal cual estaba. Tiene un 2% de posibilidades de añadir, lo que se consigue generando un número aleatorio entre 0 y 49 y comprobando si es igual a 0. En el caso de que la mutación sea añadir, esta siempre se añadirá al final del gen, la de eliminar se realiza en cualquier parte del gen.

```

def mutate2(self, candidato: list):
    numberC = (1 + len(candidato)) / 2 # Tamaño del gen
    # Si el tamaño del gen no es mayor que el tamaño máximo y el número aleatorio es 0 el gen se muta añadiendo
    if numberC < self.maxSize and random.randint(0,49) == 0:
        op_cat = random.randint(0, len(self.operations) - 1) # Selecciono aleatoriamente una categoría de operación
        # Añado al gen una operación elegida aleatoriamente de entre las de la categoría
        candidato.append(self.operations[op_cat][random.randint(0, len(self.operations[op_cat]) - 1)])
        # Añado al gen un número aleatorio entre 0 y la posición máxima
        candidato.append(random.randrange(0, 4*self.n))
    # Si no se ha añadido, el tamaño del gen menos 2 no es menor que el tamaño mínimo y el número aleatorio es 0 el gen se muta quitando
    elif numberC - 2 > self.minSize and random.randint(0,49) == 0:
        # Selecciono aleatoriamente un índice del gen
        indice = random.randrange(0, len(candidato) - 1)
        #Elimino el elemento que está en al posición del indice
        del candidato[indice]
        #Elimino el elemento que está en al posición del indice
        del candidato[indice]
    return candidato # Devuelvo el gen resultante

```

Figura 3-33. Mutate2 primera versión regresión simbólica

La segunda versión del método mutate2 la implementé después de iniciar el entrenamiento y comprender que era ineficiente por la lentitud a la que mutaban los genes, esto es algo adecuado si además de mutarlos los recombinas, pero dado que no es el caso cambié la función para que se produjera una mutación en todas las iteraciones. Esto lo logre calculando un número aleatorio o bien 0 o 1, si es 0 la mutación sería añadir y si es 1 eliminar.

```

def mutate2(self, candidato: list):
    numberC = (1 + len(candidato)) / 2 # Tamaño del gen
    n = random.randint(0,1) # Genero un 0 o un 1
    # Si el tamaño del gen no es mayor que el tamaño máximo y n es 0 el gen se muta añadiendo
    if numberC < self.maxSize and n == 0:
        op_cat = random.randint(0, len(self.operations) - 1) # Selecciono aleatoriamente una categoría de operación
        # Añado al gen una operación elegida aleatoriamente de entre las de la categoría
        candidato.append(self.operations[op_cat][random.randint(0, len(self.operations[op_cat]) - 1)])
        # Añado al gen un número aleatorio entre 0 y la posición máxima
        candidato.append(random.randrange(0, 4*self.n))
    # Si no se ha añadido, el tamaño del gen menos 2 no es menor que el tamaño mínimo y n es 1 el gen se muta quitando
    if numberC > self.minSize and n == 1:
        # Selecciono aleatoriamente un índice del gen
        indice = random.randrange(0, len(candidato) - 1)
        # Elimino el elemento que está en la posición del índice
        del candidato[indice]
        # Elimino el elemento que está en la posición del índice
        del candidato[indice]
    return candidato # Devuelvo el gen resultante

```

Figura 3-34. Mutate2 versión final regresión simbólica

Finalmente, implemente la función principal a la que llamé run, esta función se encarga de manejar el flujo de ejecución del algoritmo, recibe como parámetros: el número de genes, las listas de datos X e y, y el rendimiento al que se quiere llegar al que llamo baremo. La primera versión que implementé fue esta:

```

def run(self, numGenes, X, y, baremo: float):
    best = 1000000
    post_best = 0
    self.genes = self.create(numGenes) # Creo la población inicial de genes
    for i in range (0, len(self.genes)):# Evalúo el rendimiento de cada gen y obtengo el mejor rendimiento
        value = self.fitness(X, y, self.genes[i]) # Evalúo el rendimiento del gen
        if(value < best): # Si es mejor que el mejor hasta ahora lo sustituye
            best = value
            post_best = i
    while(best > baremo): # Mientras que el mejor no sea mejor que el baremo se ejecuta el bucle
        for j in range (0, len(self.genes)):# Recorro la lista de genes
            self.genes[j] = self.mutate2(self.genes[j]) # Ejecuto mutate2 con el gen correspondiente
        for i in range (0, len(self.genes)):# Evalúo el rendimiento de cada gen y obtengo el mejor rendimiento
            value = self.fitness(X, y, self.genes[i]) # Evalúo el rendimiento del gen
            if(value < best): # Si es mejor que el mejor hasta ahora lo sustituye
                best = value
                post_best = i
    print("El mejor gen tiene un rendimiento de " + str(best)) # Imprimo el mejor rendimiento obtenido
    self.display(self.genes[post_best]) # Imprimo el gen que ha obtenido el mejor rendimiento
    return (post_best, best)

```

Figura 3-35. Run regresión simbólica

La siguiente versión, llamada run2, implementa la eliminación de la peor mitad de los genes en cada iteración, siendo sustituidos por nuevos genes creados aleatoriamente, el guardado de los resultados en un dataframe y el guardado de resultados en un archivo de texto. La implementación que cree para la eliminación no funcionaba correctamente, ya que utilizaba un diccionario para relacionar valores y posiciones, que después generaban una lista de genes ordenada. Pero eliminaba la mitad de la lista empezando desde la mitad, lo que generaba que hubiera acceso fuera de rango.

```

def run2(self, numGenes, X, y, baremo: float):
    best = 1000000
    candidato_best = []
    self.genes = self.create(numGenes) # Creo la población inicial de genes
    for i in range (0, len(self.genes)):
        # Evalúo el rendimiento de cada gen y obtengo el mejor rendimiento
        value = self.fitness(X, y, self.genes[i]) # Evalúo el rendimiento del gen
        if(value < best): # Si es mejor que el mejor hasta ahora lo sustituye
            best = value
            candidato_best = self.genes[i] # Guardo el gen como el mejor hasta el momento
    ronda = 0
    while(best > baremo): # Mientras que el mejor no sea mejor que el baremo se ejecuta el bucle
        ronda +=1 # Sumo 1 a la variable que guarda el número de iteraciones
        for j in range (0, len(self.genes)):
            self.genes[j] = self.mutate2(self.genes[j]) # Ejecuto mutate2 con el gen correspondiente
        dicc_aux = {}
        genes_aux = self.genes.copy()
        values_list = []
        for r in range (0, len(self.genes)): # Recorro la lista de genes
            value = self.fitness(X, y, self.genes[r]) # Evalúo el rendimiento del gen
            values_list.append(value) # Añado el rendimiento a la lista de rendimientos
            dicc_aux[value] = r # Añado al diccionario al posición en la lista de genes del gen con ese rendimiento
            if(value < best): # Si es mejor que el mejor hasta ahora lo sustituye
                best = value
                candidato_best = self.genes[r] # Guardo el gen como el mejor hasta el momento
        values_array = np.array(values_list) # Transformo la lista de valores en un array de numpy
        values_array.sort() # Ordeno el array de mayor rendimiento a menor rendimiento
        #Crear la lista que va a ir en el dataframe
        genesauxiliares = []
        for value in values_array: # Añado los genes a la lista auxiliar ordenados por rendimiento
            genesauxiliares.append(self.genes[dicc_aux[value]])
        df = pd.DataFrame([{"Guardo los daots en un dataframe": "Genes": genesauxiliares,
                            "Valor": values_array,
                            "NumHoras": NUMHORAS,
                            "Ronda": ronda}])
    nombre = "pandas/dataframe" + str(ronda) # Creo el nombre del dataframe
    df.to_csv(nombre + '.csv', index=False) # Guardo el dataframe como csv
    num_del = 0
    for z in range(int(values_array.size/ 2), values_array.size): # Elimino la mitad de los genes con peor rendimiento
        if genes_aux[dicc_aux[values_array[z]]] in self.genes:
            self.genes.remove(genes_aux[dicc_aux[values_array[z]]])
            num_del += 1 # Aumento en 1 el número de eliminados
    if num_del > 2: # Si elimino más de 2 genes creo el número de genes que he eliminado y los añado a la lista
        genescreados = self.create(num_del) # Creo los genes aleatoriamente
        for gen in genescreados: # Añado los genes
            self.genes.append(gen)
    # Imprimo el gen que ha obtenido el mejor rendimiento en esta iteración, el número de iteración y el mejor rendimiento hasta ahora
    print("El mejor gen," + str(display(candidato_best)) + ",de la ronda " + str(ronda) + " tiene un rendimiento de " + str(best))
    # Guardo en un archivo de texto el gen que ha obtenido el mejor rendimiento en esta iteración, el número de iteración y el mejor rendimiento hasta ahora
    with open('candidatos.txt', 'a') as archivo:
        archivo.write("El mejor gen," + str(display(candidato_best)) + ",de la ronda " + str(ronda) + " tiene un rendimiento de " + str(best) + "\n")
    print("El mejor gen tiene un rendimiento de " + str(best)) # Imprimo el mejor rendimiento obtenido
    self.display(candidate_best) # Imprimo el gen que ha obtenido el mejor rendimiento
    return (candidate_best, best)

```

Figura 3-36. Run segunda versión regresión simbólica

La versión final, llamada runcopy, recibe como parámetro adicional una variable de tipo booleano llamada cargar, esta nueva versión implementa correctamente la eliminación, eliminando solo dos genes cada 5 iteraciones y una vez se hayan superado las 200 primeras iteraciones. También implementa un guardado del estado de los genes cada 100 iteraciones y una nueva funcionalidad que permite cargar el estado guardado como población inicial en vez de generarla aleatoriamente. El guardado en archivo y dataframe también sufren modificaciones, guardando en este último solo cada 100 iteraciones.

```

def runcopy(self, numGenes, X, y, baremo: float, cargar):
    resultado = [] # Lista en la que guardo los resultados de cada iteración
    num_veces = 0 # Variable en la que guardo al cantidad de iteraciones
    best = float('inf') # Variable en la que guardo el mejor resultado
    candidato_best = [] # Variable en la que guardo el mejor gen
    if cargar: # Si cargar es true la población inicial se obtiene del archivo de texto estado
        with open('estado.txt', 'r') as archivo: # Abro el archivo en modo lectura
            primera_linea = archivo.readline() # Leo la primera linea
            num_veces = int(primeras_linea.strip()) # Transformo lo leido en un entero y se lo asigna a número de veces
            for linea in archivo: # Leo cada linea del archivo de una en una
                linea = linea.strip() # Elimino espacios al principio y al final
                self.genes.append(self.cargar(linea)) # Transformo la linea leida en un gen y lo añado a la lista de genes
            numGenes = len(self.genes) # Asigno al número de genes la longitud de la lista de genes
    else: # Si cargar es falso al población inicial se genera aleatoriamente
        self.genes = self.create(numGenes) # Creo la población inicial y la asigno a la lista de genes
    for i in range(0, len(self.genes)): # Evaluo el rendimiento de cada gen y obtengo el mejor rendimiento
        value = self.fitness2(X, y, self.genes[i]) # Evaluo el rendimiento del gen
        if value < best: # Si es mejor que el mejor hasta ahora lo sustituye
            best = value
            candidato_best = self.genes[i] # Guardo el gen como el mejor hasta el momento
    while(best > baremo and num_veces < 100000): # Mientras que el mejor no sea mejor que el baremo y el número de iteraciones sea menor de 100000 se ejecuta el bucle
        best_iteracion = float('inf') # Variable en la que guarda el mejor resultado de esta iteración
        candi_it = []
        for j in range(0, len(self.genes)): # Recorro la lista de genes
            self.genes[j] = self.mutate2(self.genes[j]) # Ejecuto mutate2 con el gen correspondiente
        values_list = []
        for r in range(0, len(self.genes)): # Recorro la lista de genes
            value = self.fitness2(X, y, self.genes[r]) # Evaluo el rendimiento del gen
            values_list.append(value) # Añado el rendimiento a la lista de rendimientos
            if(value < best_iteracion): # Si es mejor que el mejor de la iteración hasta ahora lo sustituye
                best_iteracion = value
                candi_it = self.genes[r] # Guardo el gen como el mejor de la iteración hasta el momento
            if value < best: # Si es mejor que el mejor hasta ahora lo sustituye
                best = value
                candidato_best = self.genes[r] # Guardo el gen como el mejor hasta el momento
        if num_veces > 200 and num_veces % 5 == 0: # Si el numero de iteraciones es mayor de 200 y multiple de 5 se ejecuta el proceso de eliminación de los peores genes
            indexed_fitness = list(enumerate(values_list)) # Asigno índices a cada valor de la lista de valores y lo guardo como lista de tuplas (índice, valor)
            indexed_fitness.sort(key=lambda x: x[1]) # Ordeno la lista por rendimiento
            mejor_indice = indexed_fitness[0][0] # Guardo el indice del mejor gen de la lista
            aux = self.genes[mejor_indice] # Guardo en aux el mejor gen de la lista
            peores_indices = [idx for idx, _ in indexed_fitness[-2:]] # Guardo el indice de los dos peores genes de la lista
            peores_indices.sort(reverse=True) # Ordeno los indices de mayor a menor
            for indice in peores_indices: # Recorro la lista de indices
                del self.genes[indice] # Elimino el gen que se encuentra en este indice
            for _ in range(2): # Añado a la lista de genes dos copias del mejor gen de la lista de genes
                self.genes.append(aux)
        num_veces += 1 # Sumo 1 al número de iteraciones
        # Imprimo el número de iteraciones, el mejor rendimiento de la iteración y el mejor rendimiento hasta ahora
        print(f"Vez num:{num_veces}, valor:{best_iteracion}, valor:{best}")
        genBest = self.display(candi_it) # Guardo el mejor gen de la iteración como string
        # Guardo el número de iteración, el mejor rendimiento de la iteración y el mejor gen de la iteración como string
        resultado.append({'iteracion': num_veces, 'valor': best_iteracion, 'gen': genBest})
        with open('generacionter.txt', 'a') as archivo: # Abro el archivo en modo append
            # Guardo en el archivo de texto el número de iteración, el mejor rendimiento de la iteración y el mejor gen de la iteración como string
            archivo.write(f"Vez num:{num_veces}, valor:{best_iteracion}, gen:{genBest}\n")
        if num_veces % 100 == 0: # Si el número de iteraciones es múltiplo de 100
            df_resultados = pd.DataFrame(resultado) # Transformo la lista en un dataframe
            # Creo el nombre de guardado del dataframe
            cadena = "Dataframes/resultados_regresionSimbolicaC_it" + str(num_veces) + ".csv"
            df_resultados.to_csv(cadena, index=False) # Guardo el dataframe como csv
            with open('estado.txt', 'w') as archivo_estado: # Abro el archivo en modo escritura
                cad = str(num_veces) + "\n" # Crea un string con el número de iteraciones y un salto de linea
                archivo_estado.write(cad) # Lo escribo en el archivo
            for gen in self.genes: # Recorro la lista de genes
                cadena = str(self.display(gen)) + "\n" # Crea un string con el gen y un salto de linea
                archivo_estado.write(cadena) # Lo escribo en el archivo
            print("El mejor gen tiene un rendimiento de " + str(best)) # Imprimo el mejor rendimiento obtenido
            self.display(candidato_best) # Guardo el mejor gen como string
            df_resultados = pd.DataFrame(resultado) # Guardo los resultados como dataframe
            cadena = "Dataframes/resultados_regresionSimbolica.csv" # Crea el nombre de guardado del dataframe
            df_resultados.to_csv(cadena, index=False) # Guardo el dataframe como csv
    return (candidato_best, best)

```

Figura 3-37. Run versión final regresión simbólica

3.5.1.2 Con Constantes

El cambio principal que debí implementar fue en la estructura de los genes, ya que ahora debía diferenciar entre los números que eran constantes y los que hacían referencia a posiciones. Para esto cree las clases posición y constante.

```

class position:
    def __init__(self, value):
        self.value = value

    def getValue(self, valores : list[float]):
        return valores[self.value] # Devuelve el valor asociado a esa posición

    def display(self):
        return ("p"+ str(self.value)) # Devuelve el objeto posición como string

```

Figura 3-38. Class Position regresión simbólica constantes

```

class constant:
    def __init__(self, value):
        self.value = value

    def getValue(self, valores : list[float]):
        return self.value # Devuelve el valor de la constante

    def display(self):
        return ("c"+ str(self.value)) # Devuelve el objeto constante como string

```

Figura 3-39. Class Constant regresión simbólica constantes

Además de esto debí de modificar el método cargar, para que reconociera los objetos de clase posición y constante y los interpretara correctamente.

```

def cargar(self, linea):
    gen = []
    # Uso una expresión regular para encontrar:
    # constantes que empiezan con 'c' seguidas de dígitos y opcionalmente decimales (c\d+(:\.\d+)?)?
    # posiciones que empiezan con 'p' seguidas de dígitos (p\d+)
    # o operadores (+, -, *, /)
    tokens = re.findall(r'c\d+(:\.\d+)?|p\d+|[+\-*\/()]+', linea)
    for tok in tokens:
        if tok.startswith('c'):
            # Si el token empieza con 'c', creo una constante convirtiendo el valor a float
            gen.append(_constant(float(tok[1:])))
        elif tok.startswith('p'):
            # Si el token empieza con 'p', creo una posición convirtiendo el valor a int
            gen.append(_position(int(tok[1:])))
        else:
            # Si es un operador, lo agrego como cadena
            gen.append(tok)
    return gen # Devuelvo el gen

```

Figura 3-40. Cargar regresión simbólica constantes

También tuve que modificar significativamente los métodos relacionados con la creación de la primera población. Creando los métodos `create_gen`, que crea un gen recibiendo como parámetro el número máximo de posiciones.

```

def create_gen(self, r : int):
    aux = []
    aux.append(aniadirNum(r)) # Añado al gen una posición o constante
    # Genero un tamaño aleatorio para el gen comprendido entre el tamaño máximo y el tamaño mínimo
    for i in range(random.randint(self.minSize, self.maxSize - 1)):
        # Selecciono aleatoriamente una categoría de operación
        numLO = random.randint(0, len(self.operations) - 1)
        # Añado al gen una operación elegida aleatoriamente de entre las de la categoría
        aux.append(self.operations[numLO][random.randint(0, len(self.operations[numLO]) - 1)])
        aux.append(aniadirNum(r)) # Añado al gen una posición o constante
    return aux # Devuelvo el gen generado

```

Figura 3-41. Create_gen regresión simbólica constantes

El segundo método llamado aniadirNum, recibe también como parámetro el número máximo de posiciones. Su función es devolver una posición o una constante.

```

def aniadirNum(n:int):
    if random.randint(0,1) == 0: # Si es número aleatorio es 0 añado una constante
        return constant(random.uniform(0.0, 5.0)) # Añado una constante con valor entre 0 y 5
    else: # Si es número aleatorio es 1 añado una posición
        return position(random.randrange(0, n)) # Añado una posición con valor entre 0 y n-1

```

Figura 3-42. AniadirNum regresión simbólica constantes

Otro método que modificó fue display, para adaptarlo a la impresión de constantes y posiciones.

```

def display(self, candidato: list):
    cadena = ""
    for i in range(0, len(candidato)): # Recorro el gen
        if i % 2 == 1: # Si i es impar estoy añadiendo una operación a la cadena
            cadena = cadena + candidato[i] + ' ' # Añado la operación con un espacio detrás
        else: # Si i es impar estoy añadiendo una posición o constante a la cadena
            # Añado la posición o constante, transformada en un string, con un espacio detrás
            cadena = cadena + candidato[i].display() + ' '
    return cadena # Devuelvo la cadena como string

```

Figura 3-43. Display regresión simbólica constantes

El resto de los métodos sufrieron cambios menores o no sufrieron cambios para poder adaptarlos al uso de constantes.

Capítulo 4 - Evaluación de Resultados

Tras entrenar los algoritmos procedí a la evaluación de los resultados. Realicé la evaluación de rendimiento en dos fases, primero evalué el rendimiento de cada algoritmo individualmente, para así poder verificar o refutar la hipótesis que sostiene que el rendimiento es superior al utilizar los datos que contienen el índice miedo-codicia, y posteriormente comparé los resultados de los algoritmos entre sí para dilucidar cual obtuvo un mejor rendimiento. Todos los rendimientos que expresaré en el documento son tantos por ciento.

4.1 Random Forest

4.1.1 Sin Índice de Miedo-Codicia

Para visualizar los datos de rendimiento del algoritmo represento en la gráfica el rendimiento en el eje y, la profundidad máxima del árbol en el eje X y el número de horas a través del color y forma de cada línea de la gráfica.

La primera gráfica se corresponde con las profundidades máximas que van de 1 a 400. En esta gráfica no pueden apreciarse correctamente las variaciones de rendimiento. Lo que sí se puede apreciar es una mejora de rendimiento muy grande cuando aumenta la profundidad desde uno hasta siete, a partir de nueve parece mantenerse constante.

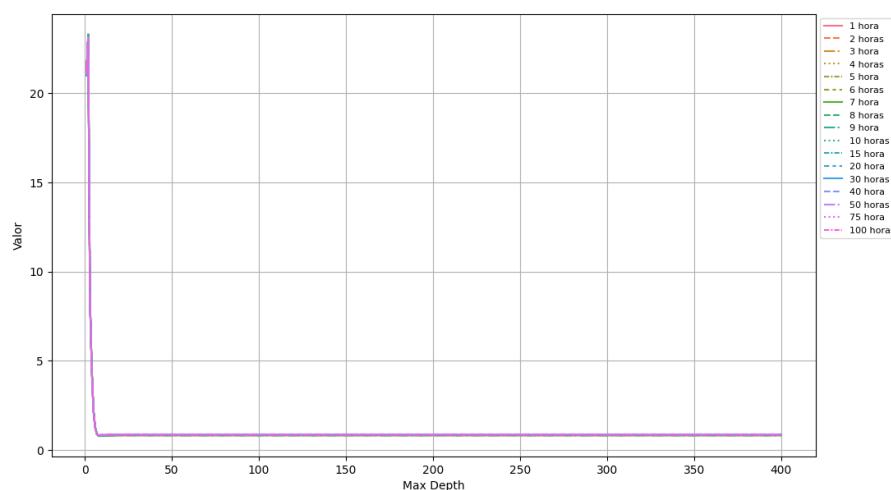


Tabla 4-1. Profundidad máxima 1 a 400 RandomForest

La siguiente gráfica se corresponde con las profundidades máximas que van de 7 a 150. En esta gráfica puede observarse como hay una mejora hasta nueve, seguida de un empeoramiento de nueve hasta 40 y a partir de ahí el rendimiento se vuelve constante.

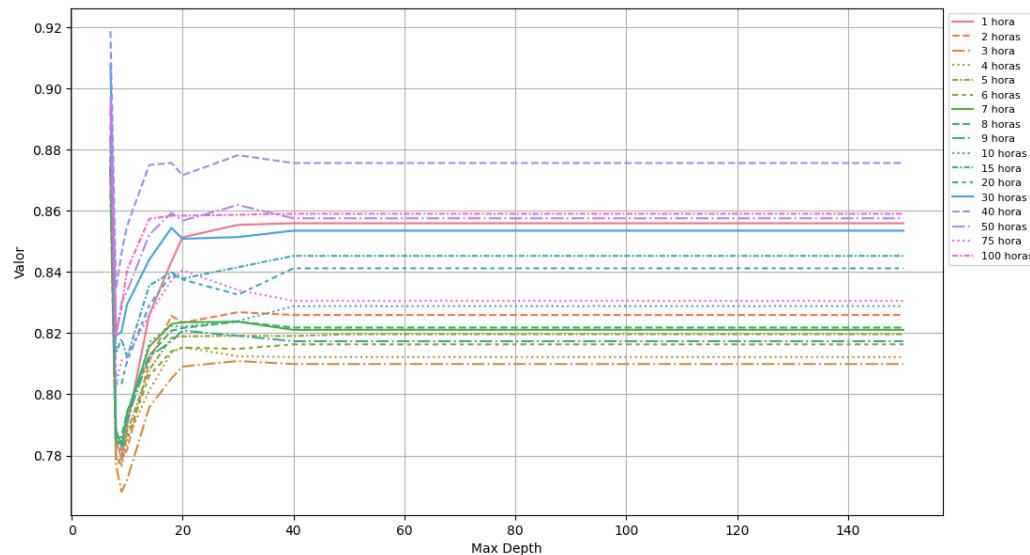


Tabla 4-2. Profundidad máxima 7 a 150 RandomForest

La siguiente gráfica nos muestra de 1 a 8 para permitir observar el ritmo al que mejora el rendimiento en esos primeros incrementos de profundidad. Podemos observar que la mayor mejora se produce entre dos y tres.

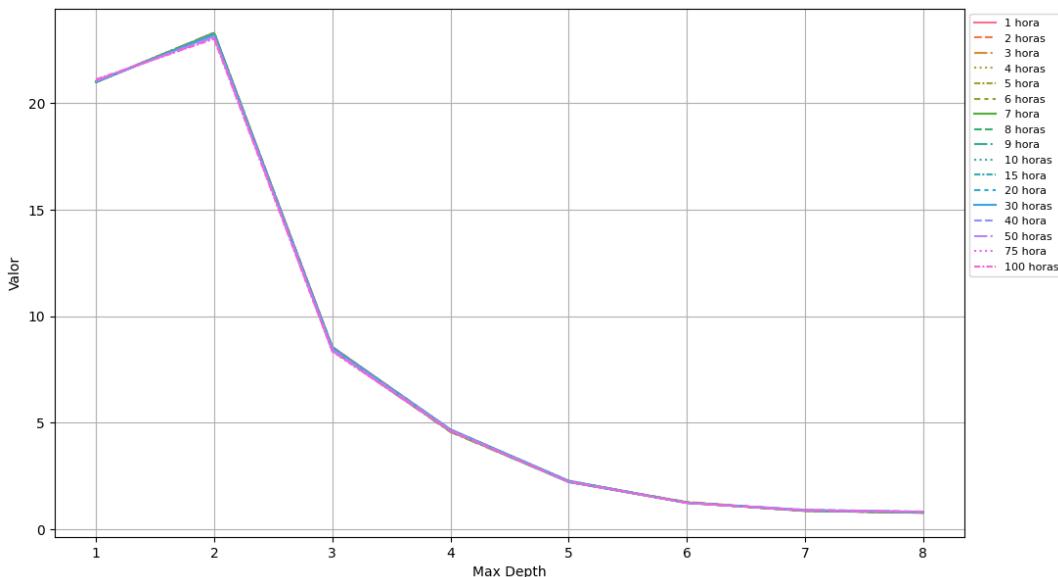


Tabla 4-3. Profundidad máxima 1 a 8 RandomForest

La siguiente gráfica nos muestra de 7 a 10 para permitir observar la evolución del rendimiento en ese intervalo. Podemos observar que el mejor rendimiento lo alcanza usando tres horas y una profundidad máxima de nueve.

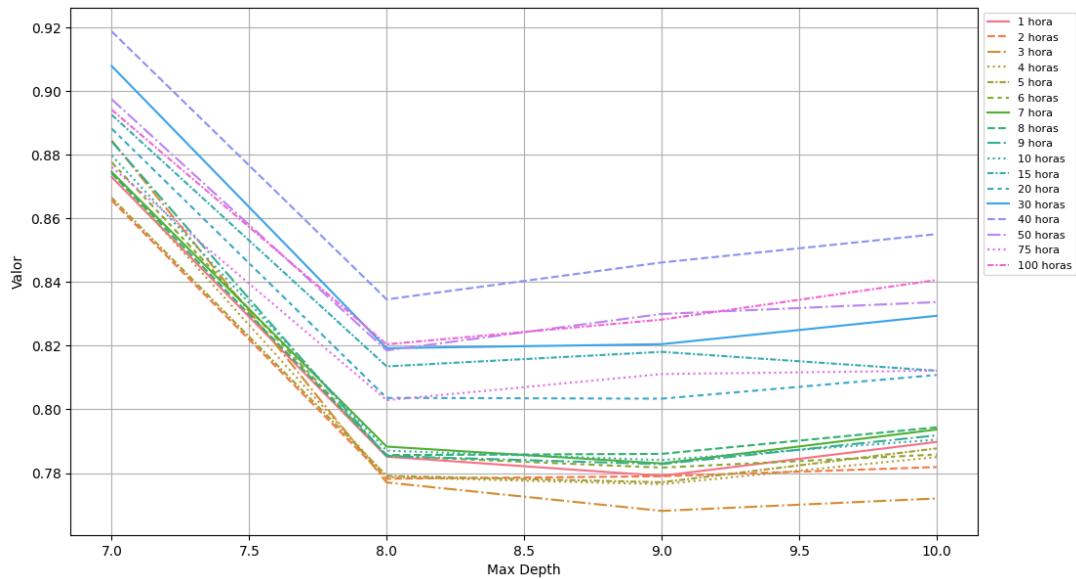


Tabla 4-4. Profundidad máxima 7 a 10 RandomForest

Por último, esta gráfica nos muestra los resultados de 7 a 400, permitiéndonos observar que el rendimiento se mantiene constante a partir de cuarenta de profundidad máxima.

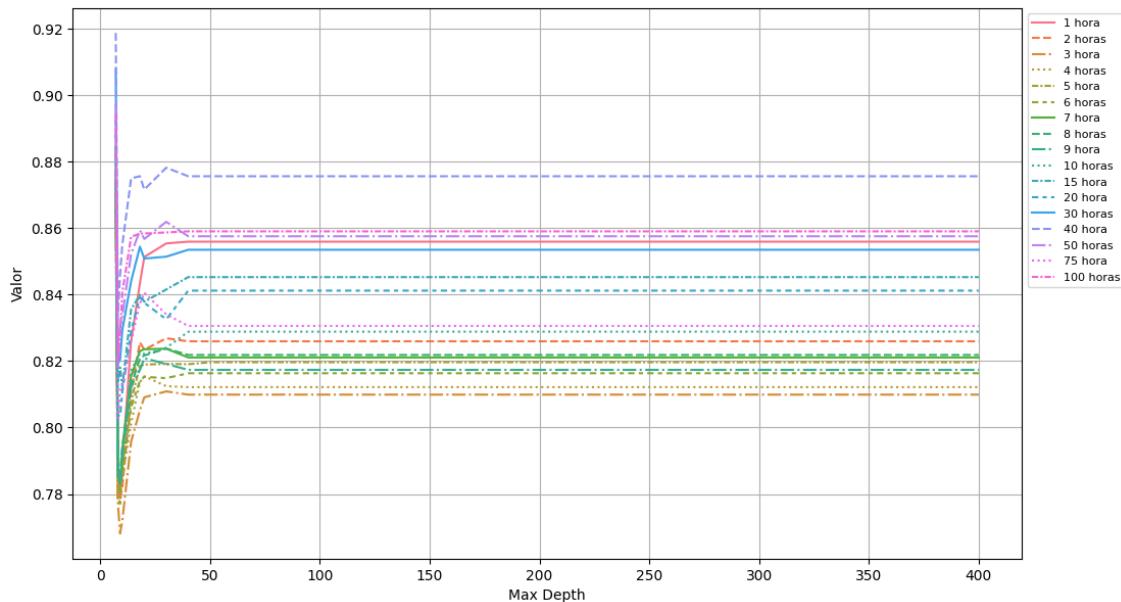


Tabla 4-5. Profundidad máxima 7 a 400 RandomForest

El mejor rendimiento obtenido es de 0,768 obtenido usando tres horas para hacer la predicción y una profundidad máxima de 9.

4.1.2 Con Índice de Miedo-Codicia

Mostraré las mismas gráficas en el caso con IMC, comenzaré de nuevo con la gráfica de 1 a 400. De nuevo se observa una gran mejora desde profundidad uno hasta ocho, seguido de lo que parece ser un rendimiento constante, sin embargo, como en el caso anterior esta gráfica no permite una visualización precisa.

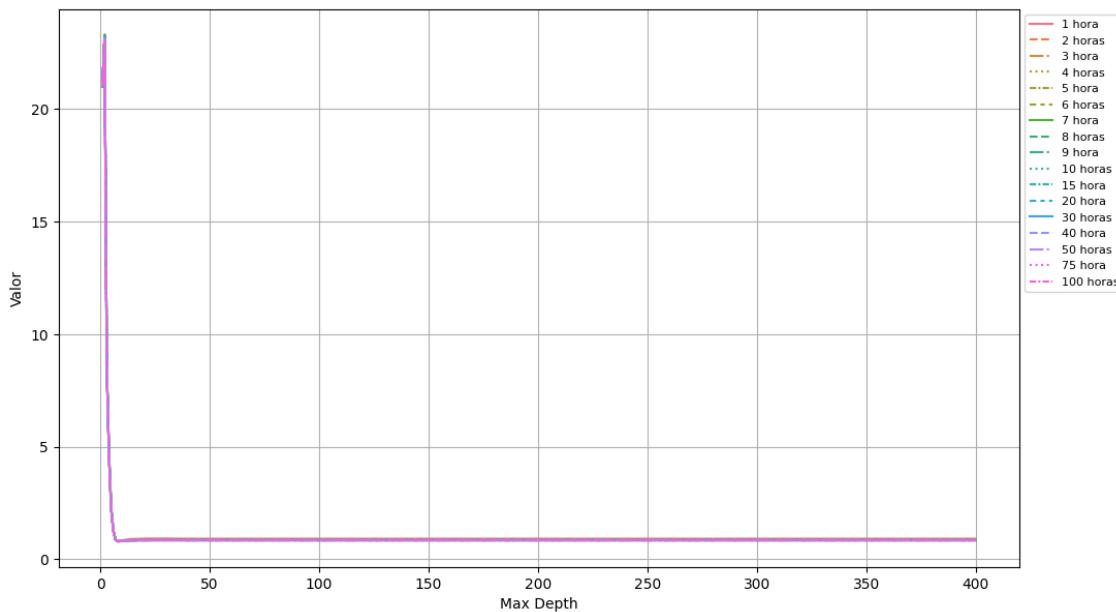


Tabla 4-6. Profundidad máxima 1 a 400 RandomForest IMC

La siguiente gráfica se corresponde con las profundidades máximas que van de 7 a 150. En esta gráfica puede observarse como hay una mejora hasta nueve, seguida de un empeoramiento de nueve hasta 40 y a partir de ahí el rendimiento se vuelve constante.

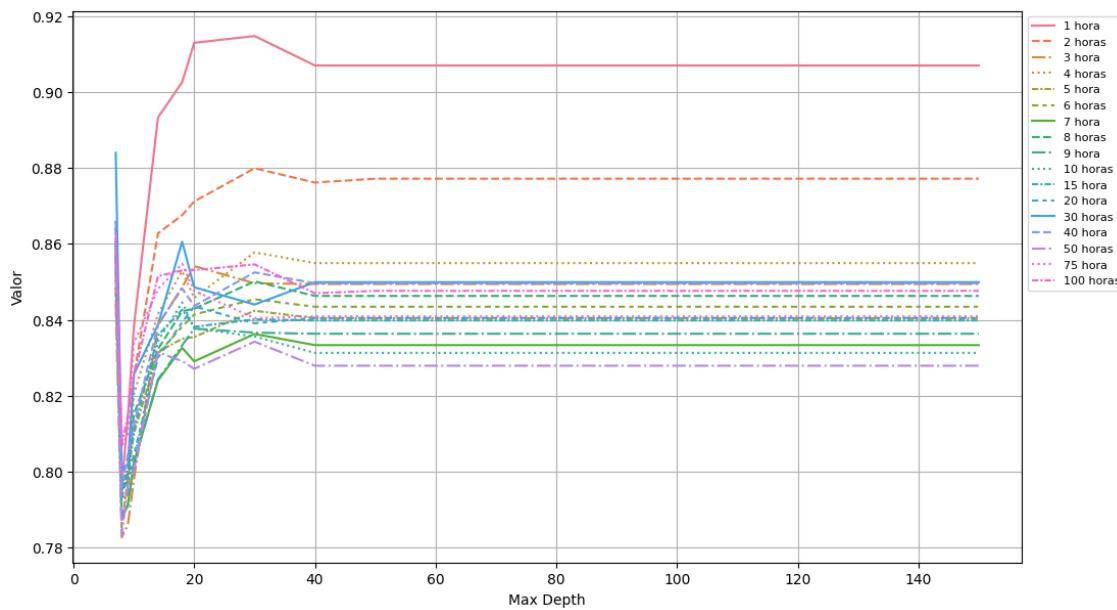


Tabla 4-7. Profundidad máxima 7 a 150 RandomForest IMC

La siguiente gráfica nos muestra de 1 a 8 para permitir observar el ritmo al que mejora el rendimiento en esos primeros incrementos de profundidad. Podemos observar que la mayor mejora se produce entre dos y tres.

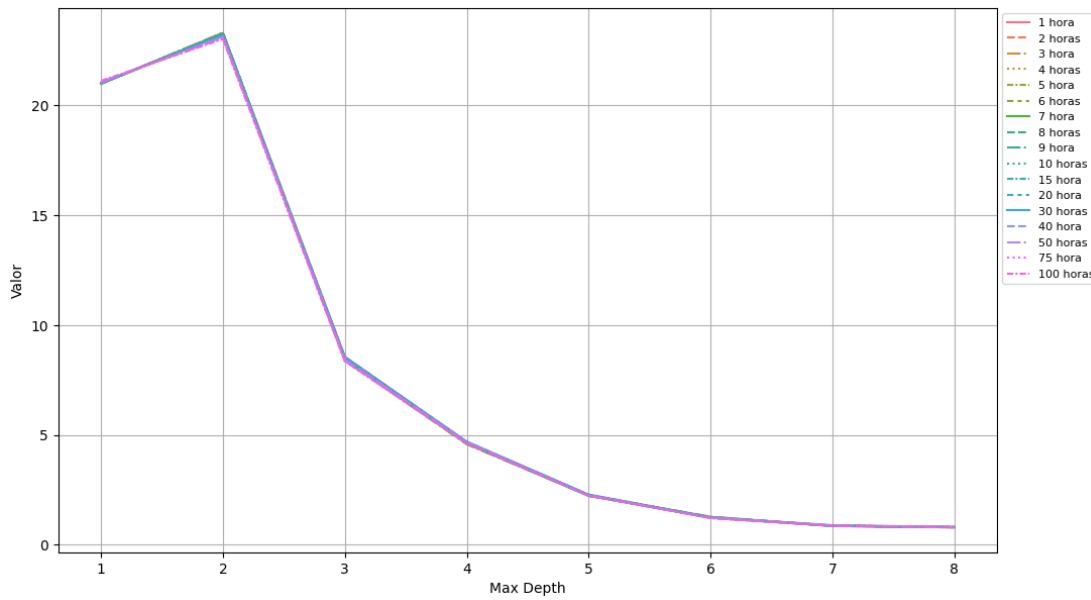


Tabla 4-8. Profundidad máxima 1 a 8 RandomForest IMC

La siguiente gráfica nos muestra de 7 a 10 para permitir observar la evolución del rendimiento en ese intervalo. Podemos observar que el mejor rendimiento lo alcanza usando tres horas y una profundidad máxima de 8.

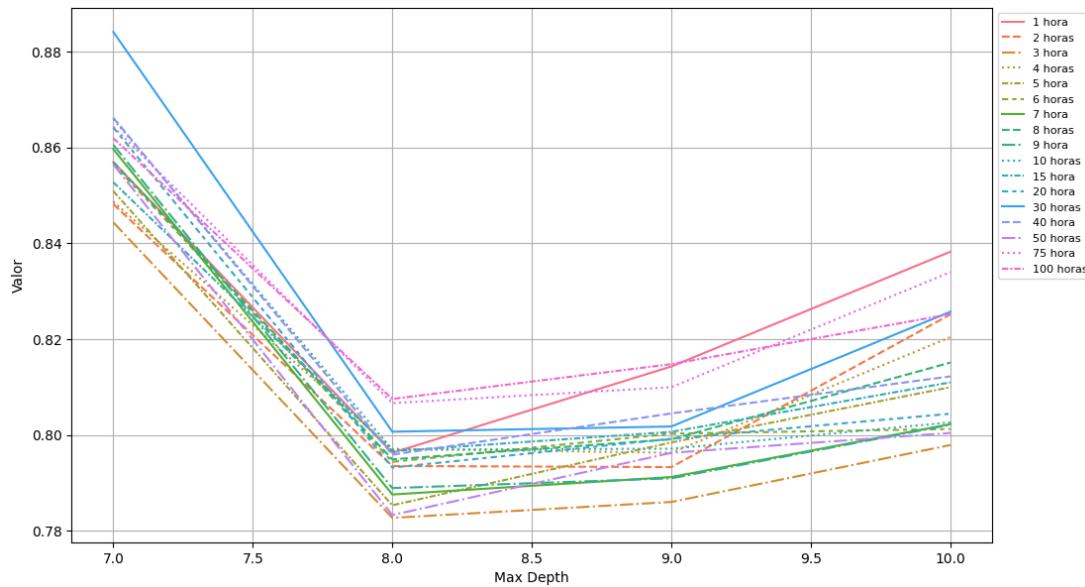


Tabla 4-9. Profundidad máxima 7 a 10 RandomForest IMC

Por último, esta gráfica nos muestra los resultados de 7 a 400, permitiéndonos observar que el rendimiento se mantiene constante a partir de cuarenta de profundidad máxima.

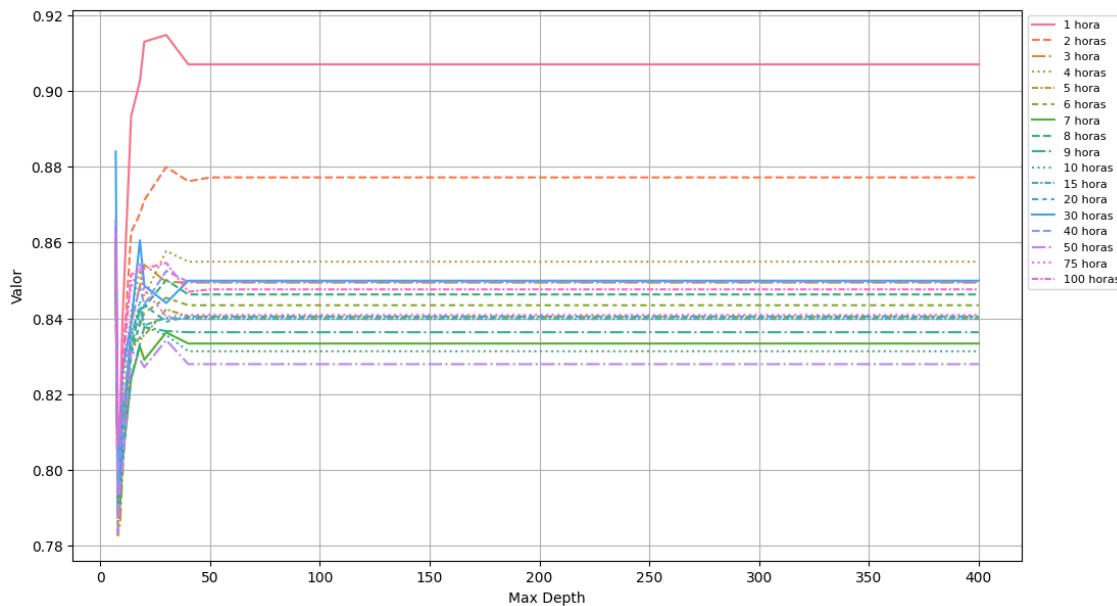


Tabla 4-10. Profundidad máxima 7 a 400 RandomForest IMC

El mejor rendimiento obtenido es de 0,782, obtenido usando tres horas y una profundidad máxima de 8.

4.1.3 Conclusión

Tras analizar los datos obtenidos puedo concluir que la hipótesis relativa al índice de miedo-codicia no se puede confirmar para el algoritmo RandomForest ya que las predicciones con los dataframe que lo usan tienen un menor rendimiento al usar cantidades de horas pequeñas pero un mayor rendimiento al usar cantidades de horas grandes para hacer el entrenamiento y la predicción.

También es destacable que la diferencia de rendimiento entre la mejor cantidad de horas y la peor cantidad de horas es significativamente mayor al usar el índice de miedo-codicia, esto parece indicar que para cantidades pequeñas de horas el índice miedo-codicia introduce ruido en vez de una señal.



Tabla 4-11. RandomForest IMC vs Normal

4.2 XGBoost

4.2.1 Sin Índice de Miedo-Codicia

Para visualizar los datos de rendimiento del algoritmo XGBoost he decidido fijar el parámetro eta y representar en la gráfica el rendimiento en el eje y, la profundidad máxima del árbol en el eje X y el número de horas a través del color y forma de cada línea de la gráfica. La primera gráfica se corresponde con eta 0.03 que es una tasa de aprendizaje muy rápida y que genera aprendizajes agresivos que corren el riesgo de producir overfitting.

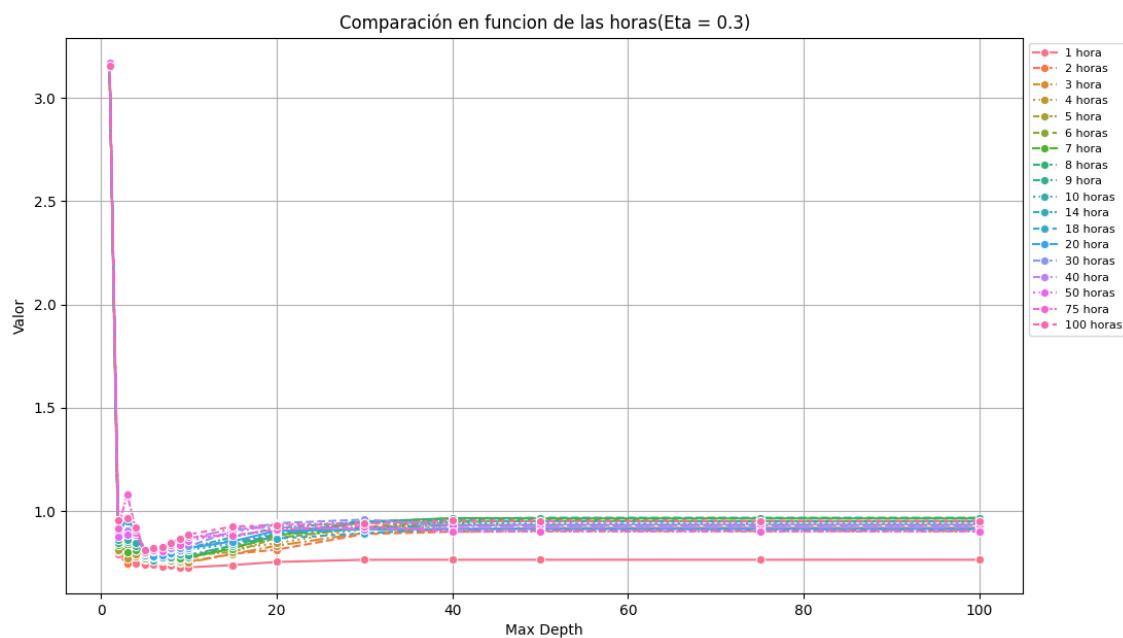


Tabla 4-12. Eta = 0,3 completa XGBoost

Para favorecer una correcta visualización, la siguiente gráfica contiene solo las profundidades máximas de 1 a 20.

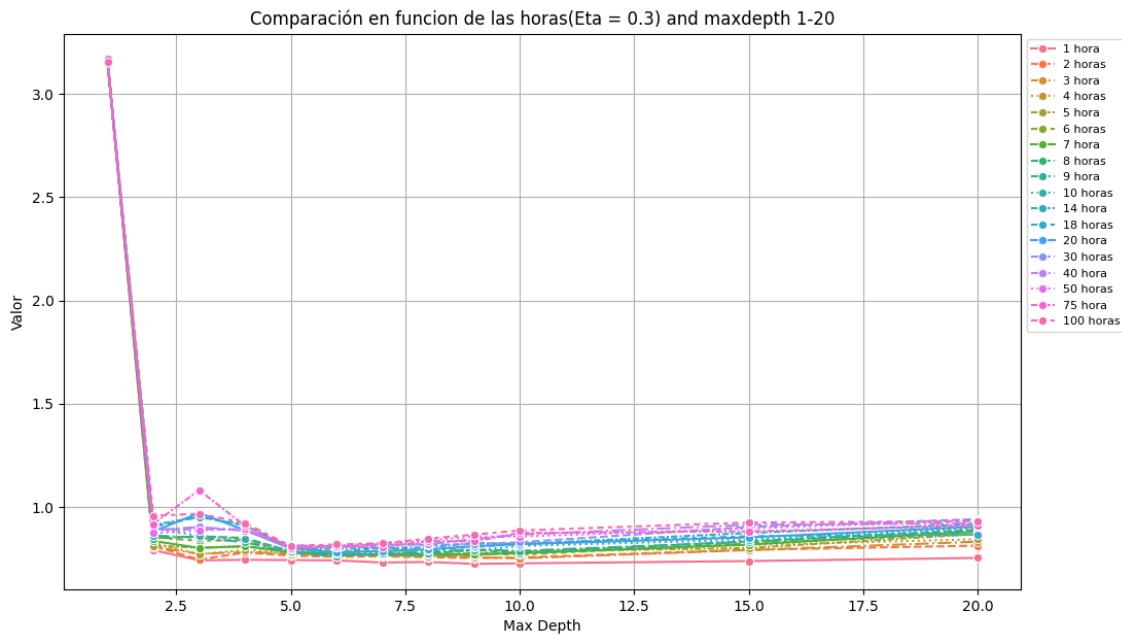


Tabla 4-13. $\eta = 0.3$, max depth 20 XGBoost

En la gráfica se puede observar cómo el rendimiento de todas las horas experimenta una gran mejoría cuando la profundidad máxima aumenta de uno a dos, esa tendencia de mejora persiste hasta la profundidad máxima cinco, a partir de la cual empiezan a empeorar los resultados. También podemos observar que el número de horas que ha obtenido el mejor rendimiento es el de una hora, observándose que la tendencia es que a más horas peor rendimiento cuando la profundidad máxima es baja, superando sin embargo al resto de horas, excepto a las predicciones usando solo una hora, cuando la profundidad máxima aumenta.

La siguiente gráfica se corresponde con eta 0,1 el cual es un equilibrio entre velocidad de aprendizaje y generalización, ay que aprende más lento que con eta 0,3 pero presenta mucho menos riesgo de overfitting.

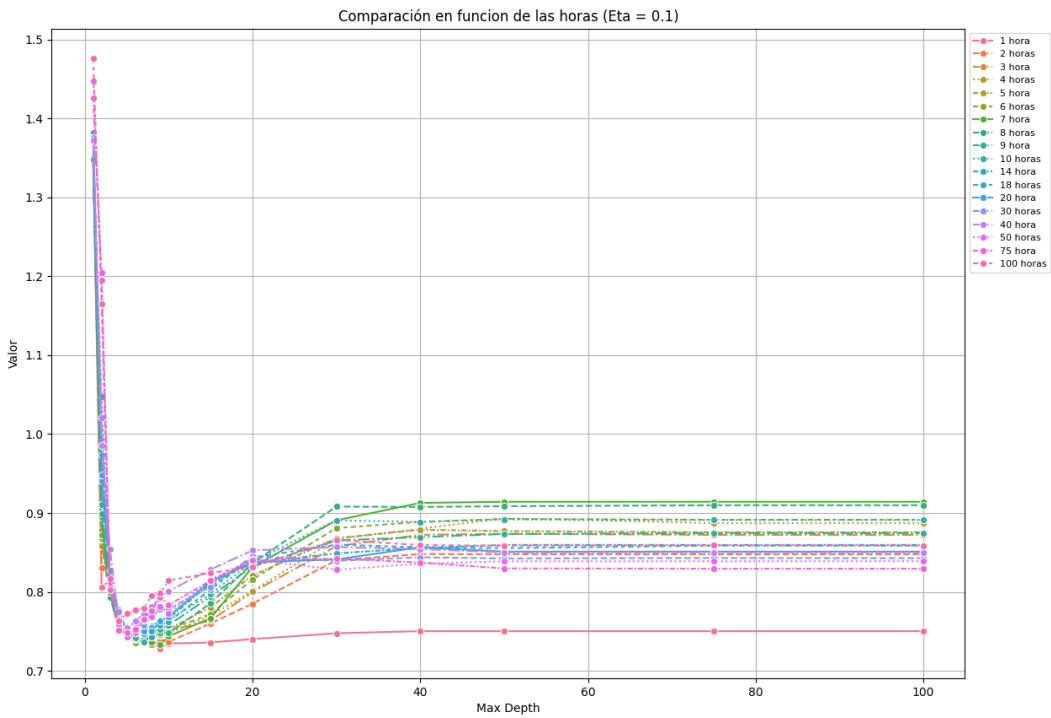


Tabla 4-14. Eta = 0,1 completa XGBoost

Para favorecer una correcta visualización, la siguiente gráfica contiene solo las profundidades máximas de 1 a 20.

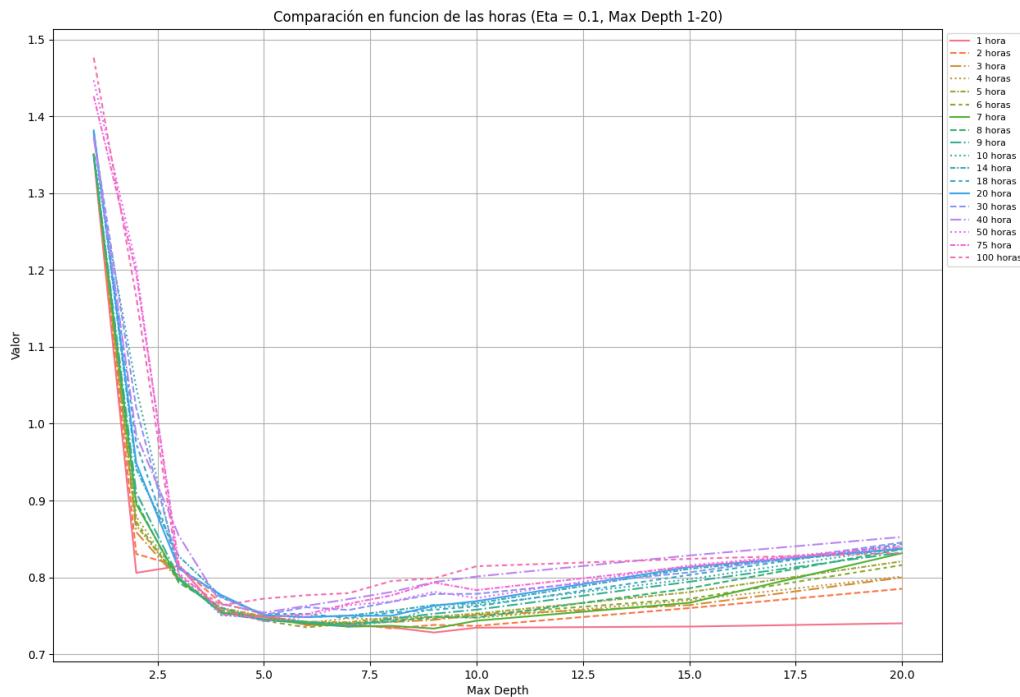


Tabla 4-15. Eta = 0,1, max depth 20 XGBoost

Como sucede con eta 0,3, al aumentar la profundidad máxima aumenta el rendimiento al usar más horas para la predicción respecto a usar menos horas. Aún así el mejor rendimiento se obtiene al usar una hora con una profundidad máxima de 9. En este caso sin embargo la mejora es más progresiva y el rendimiento aumenta significativamente de profundidad máxima uno a profundidad máxima nueve, a partir de la cual comienza a empeorar el rendimiento.

La gráfica siguiente muestra el eta 0,01, este eta produce un aprendizaje muy lento con gran estabilidad y muy bajo riesgo de overfitting, sin embargo, hacen falta un gran número de árboles.

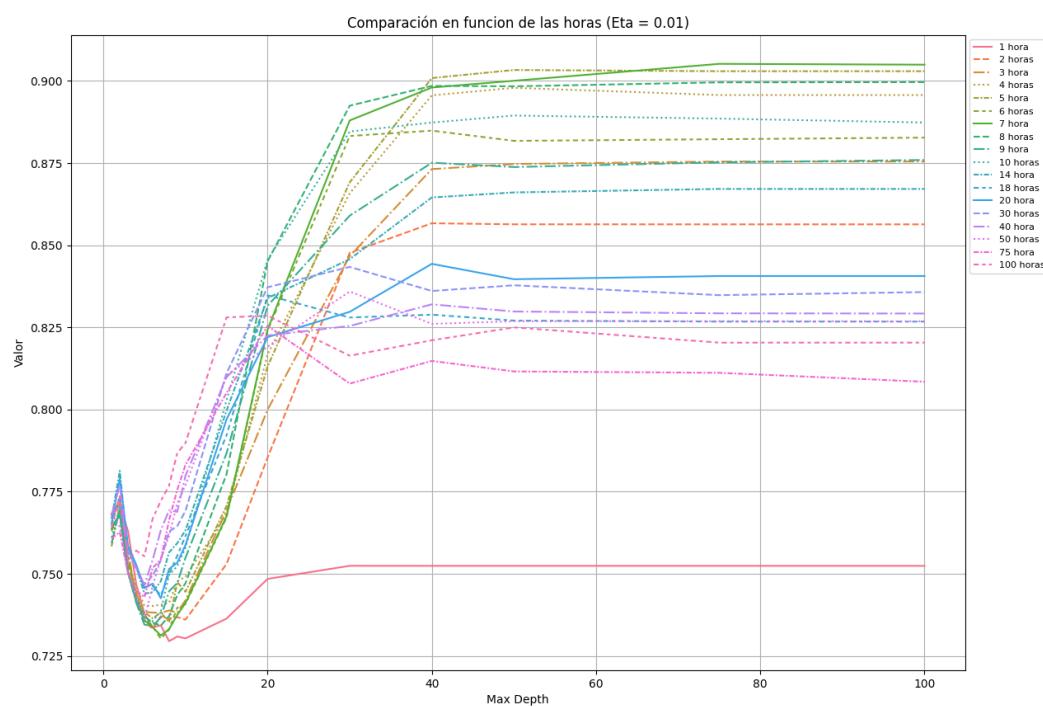


Tabla 4-16. Eta = 0,01 completa XGBoost

Para favorecer una correcta visualización, la siguiente gráfica contiene solo las profundidades máximas de 1 a 20.

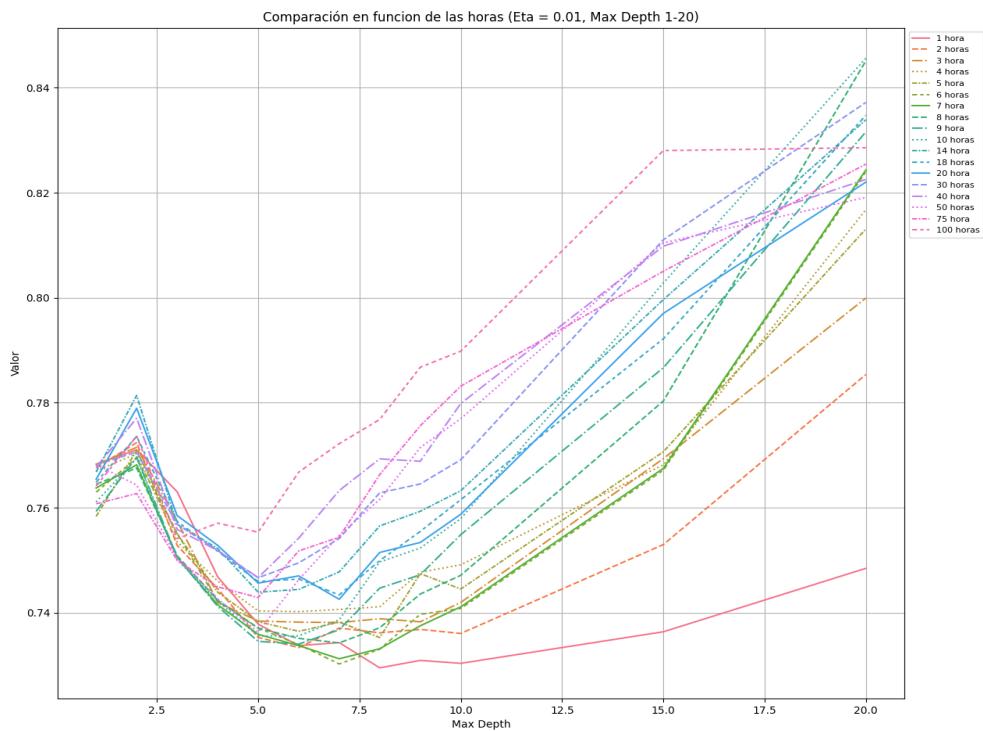


Tabla 4-17. Eta = 0,01 , max depth 20 XGBoost

Como se puede observar en la gráfica se rompe la tendencia de las dos eta anteriores y no se produce mejora de uno de profundidad máxima a dos, cabe destacar que al contrario que las dos eta anteriores con esta eta el rendimiento obtenido con uno de profundidad máxima se encuentra cerca del mejor rendimiento. Producéndose una pequeña mejora de dos a diez de profundidad máxima, a partir de diez el rendimiento empeora drásticamente, empeorando más para los modelos que utilizan pocas horas para hacer las predicciones.

El mejor rendimiento obtenido es de 0,723 obtenido usando una hora para hacer la predicción, siendo eta igual a 0,3 y una profundidad máxima de 9. Sin embargo, el mejor rendimiento obtenido usando los otros dos eta es muy similar a este.

4.2.2 Con Índice de Miedo-Codicia

Seguiré los mismos pasos que para el caso sin IMC. La primera gráfica se corresponde con eta 0,03.

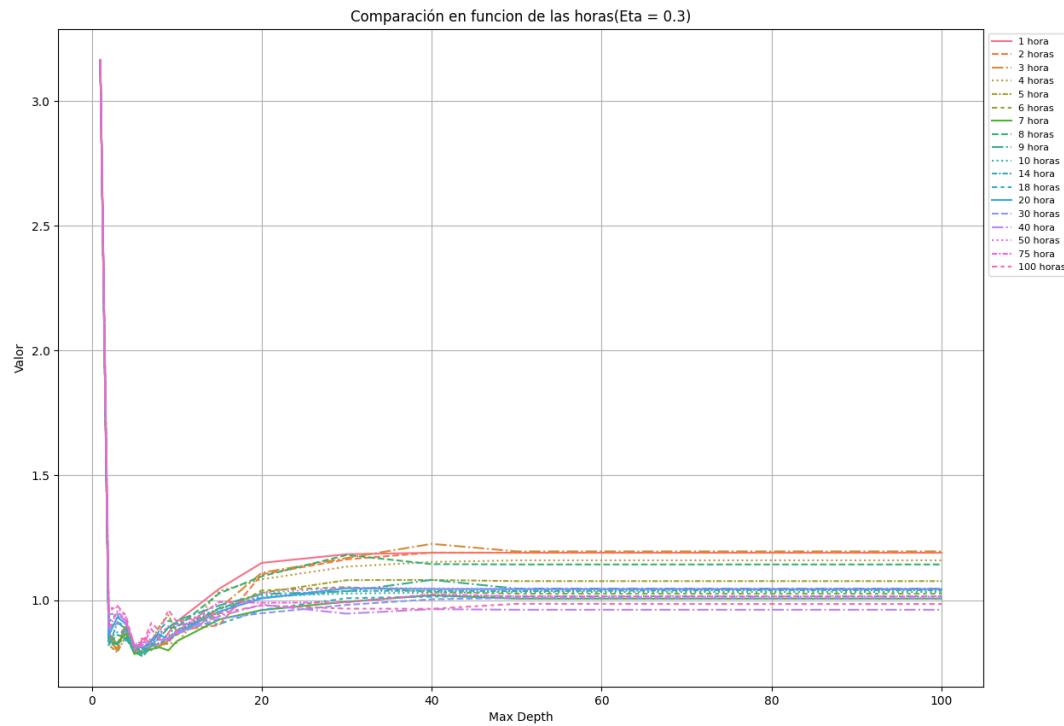


Tabla 4-18. Eta = 0,3 completa XGBoost IMC

Para favorecer una correcta visualización, la siguiente gráfica contiene solo las profundidades máximas de 1 a 20.

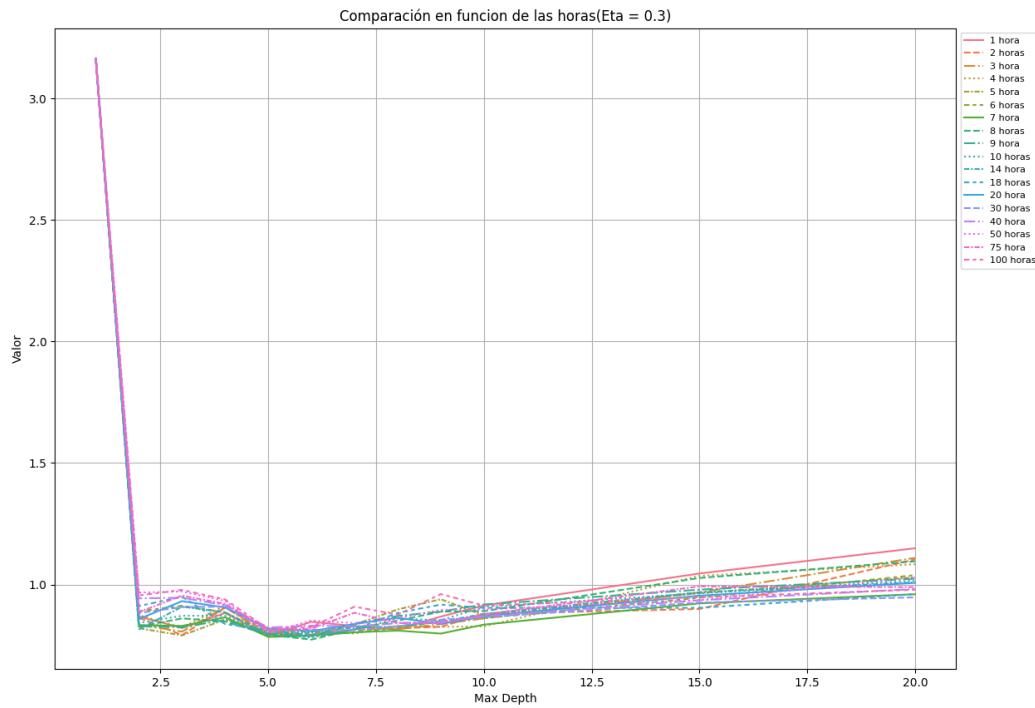


Tabla 4-19. Eta = 0,3, max depth 20 XGBoost IMC

Tal como se puede observar en la gráfica se produce una gran mejora cuando la profundidad máxima pasa de uno a dos, quedándose estancado hasta 7 y empeorando considerablemente desde esa profundidad.

La siguiente gráfica se corresponde con eta 0,1.

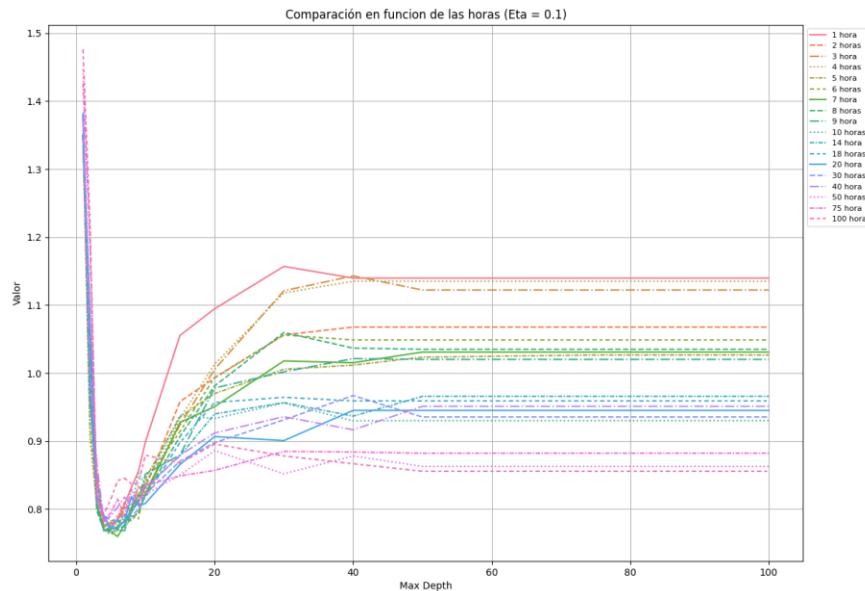


Tabla 4-20. Eta = 0,1 completa XGBoost IMC

Para favorecer una correcta visualización, la siguiente gráfica contiene solo las profundidades máximas de 1 a 20.

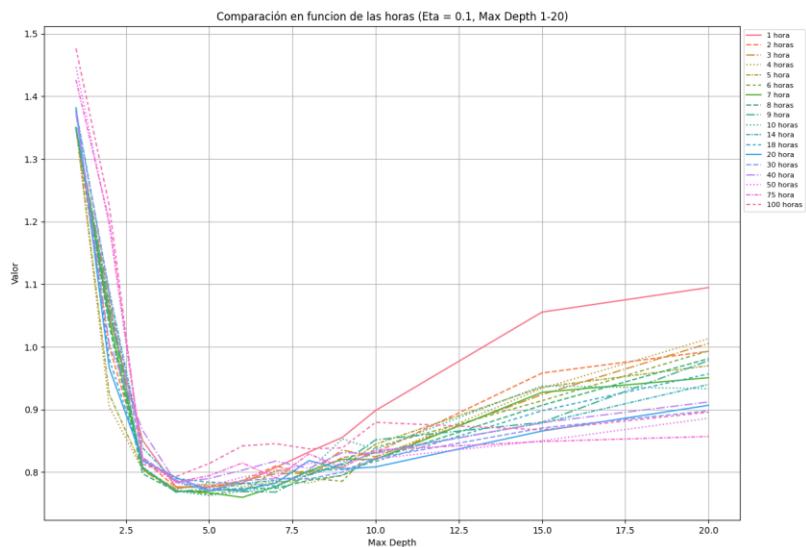


Tabla 4-21. Eta = 0,1, max depth 20 XGBoost IMC

En este caso la mejora se produce de manera algo más progresiva desde que la profundidad máxima es uno hasta que llega a seis, a partir de seis el rendimiento comienza a empeorar hasta que en cuarenta deja de variar.

La siguiente gráfica se corresponde con eta 0,01.

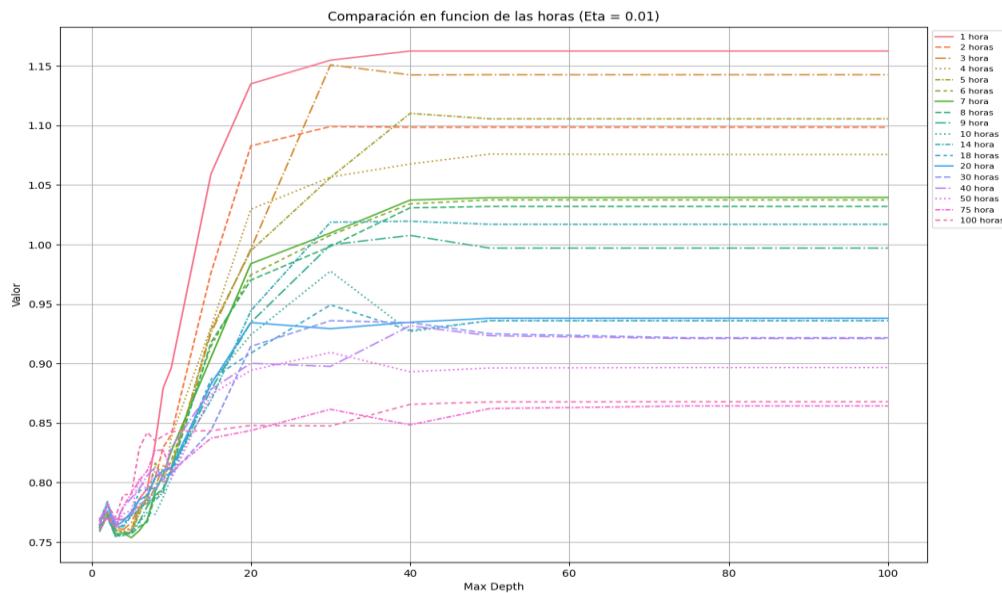


Tabla 4-22. Eta = 0,01 completa XGBoost IMC

Para favorecer una correcta visualización, la siguiente gráfica contiene solo las profundidades máximas de 1 a 20.

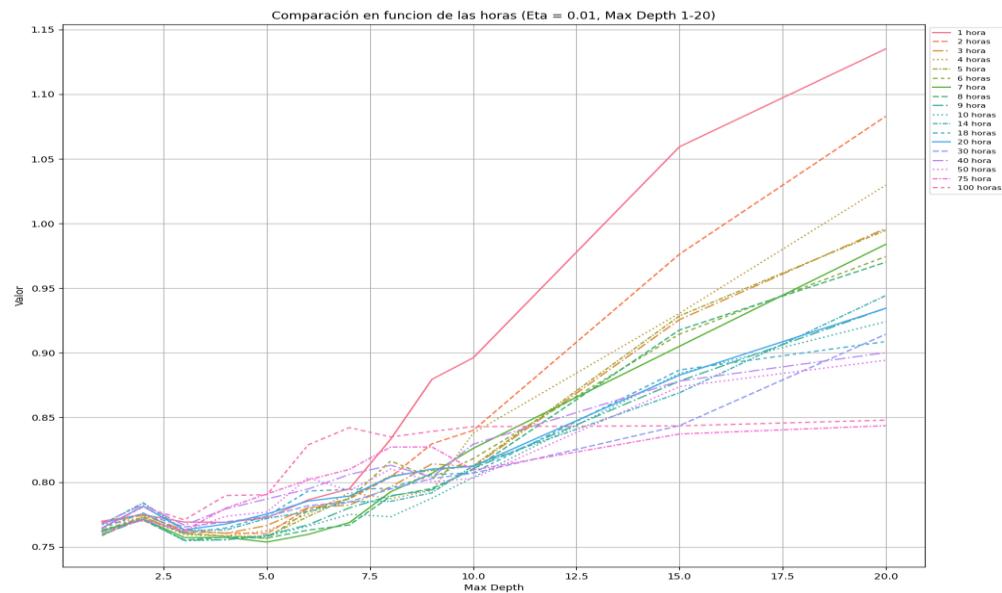


Tabla 4-23. Eta = 0,01, max depth 20 XGBoost IMC

En este caso podemos observar como hay un pequeño empeoramiento cuando la profundidad máxima pasa de uno a dos para inmediatamente volver a mejorar hasta llegar a cinco. A partir de cinco se produce un gran empeoramiento, especialmente en las predicciones que usan pocas horas, este empeoramiento continúa hasta que la profundidad máxima llega a cuarenta, profundidad máxima a partir de la cual el rendimiento se estanca.

El mejor rendimiento obtenido es de 0,753, obtenido usando siete horas para el entrenamiento del modelo, siendo eta igual a 0,01 y una profundidad máxima de 5. Sin embargo, el mejor rendimiento obtenido usando los otros dos eta es similar a este.

4.2.3 Conclusión

Tras analizar los datos obtenidos puedo concluir que la hipótesis relativa al índice de miedo-codicia no se cumple para el algoritmo XGBoost ya que las predicciones con los dataframe que lo usan tienen un menor rendimiento, tanto el mejor rendimiento como el rendimiento general.

A parte de la diferencia de rendimiento, la principal diferencia que he encontrado ha sido la diferencia en cuanto al rendimiento obtenido por los modelos entrenados usando solo una hora, en el caso de no usar el índice de miedo-codicia los modelos entrenados con una sola hora presentaban el mejor rendimiento para prácticamente todas las profundidades. Sin embargo, pasaba todo lo contrario para los modelos que usan el índice de miedo-codicia siendo los modelos entrenados usando una hora los que peor rendimiento presenta para una amplia mayoría de profundidades máximas.

Todo ello me lleva a concluir que, en el caso del algoritmo XGBoost, el uso del índice de miedo-codicia solo introduce ruido en el modelo empeorando la calidad de las predicciones.

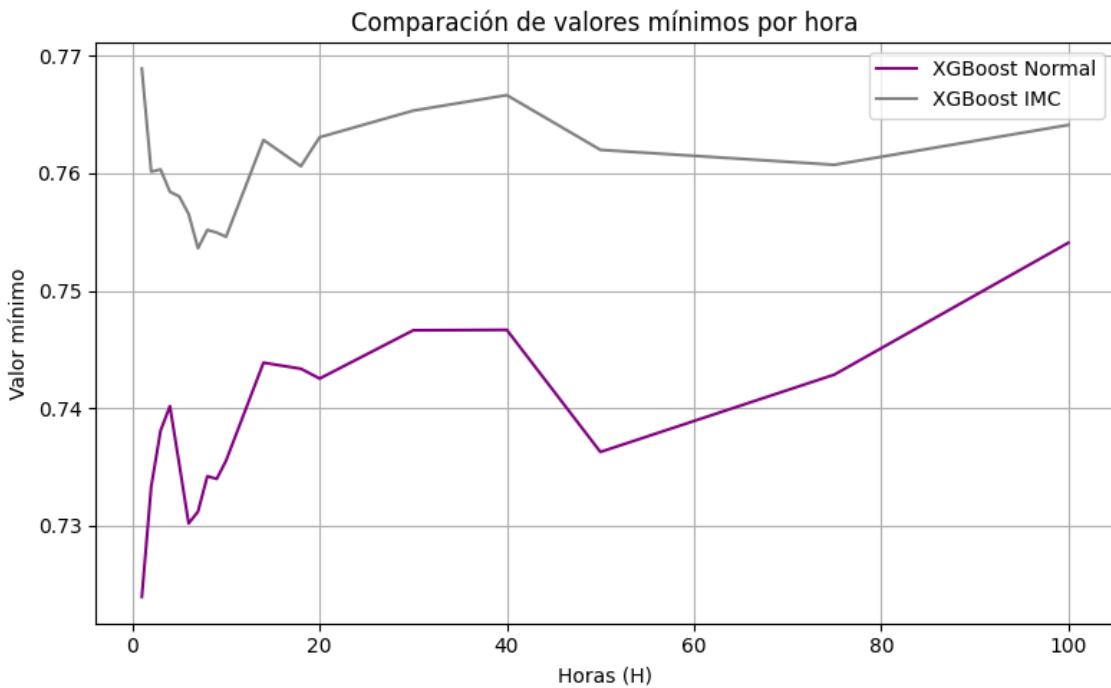


Tabla 4-24. XGBoost IMC vs Normal

4.3 Redes Neuronales

4.3.1 Sin Índice de Miedo-Codicia

Para visualizar los datos de rendimiento de las redes neuronales he decidido fijar el parámetro epoch y representar en la gráfica el rendimiento en el eje y, el batch_size en el eje X y el número de horas a través del color y forma de cada línea de la gráfica. La primera gráfica se corresponde con epoch 4, un epoch es una pasada completa por todo el conjunto de datos de entrenamiento.

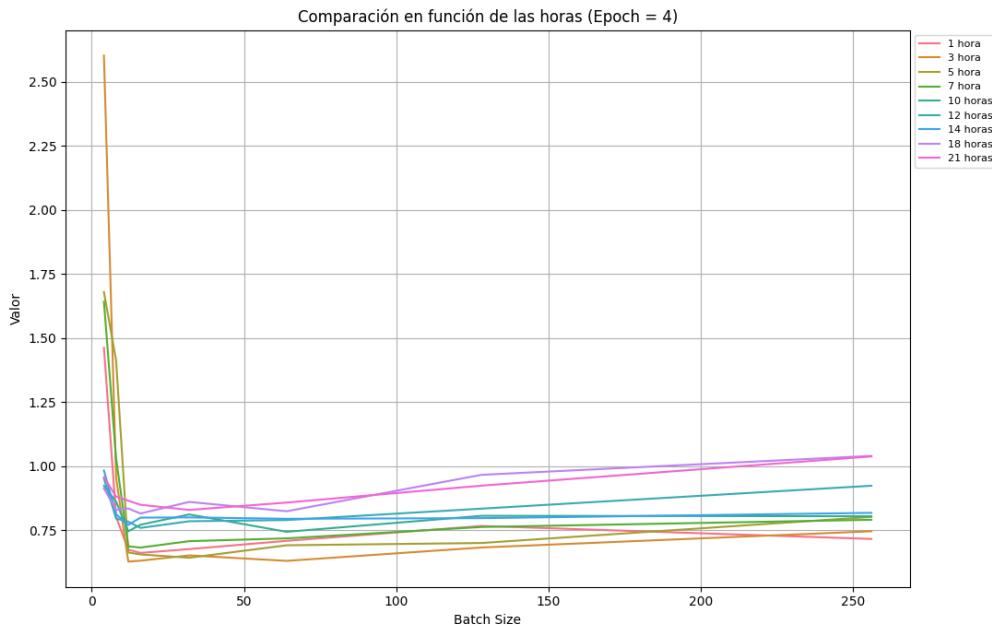


Tabla 4-25. Epoch 4 redes neuronales

Como se puede observar en la gráfica, para epoch igual a cuatro el mejor resultado se obtiene con batch_size igual a dieciséis, empeorando ligeramente con batch_size superiores. También observamos una gran mejora de batch_size cuatro a batch_size dieciséis.

La siguiente gráfica se corresponde con epoch 6.

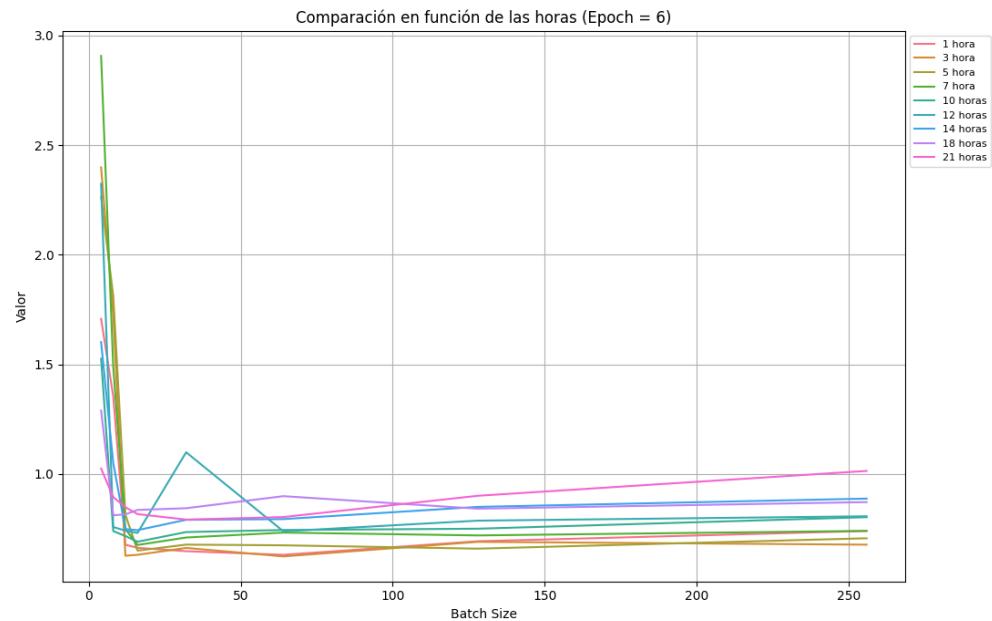


Tabla 4-26. Epoch 6 redes neuronales

Como se puede observar en la gráfica, para epoch igual a seis los mejores resultados se obtienen con batch_size igual a doce, aunque el mejor resultado se obtiene con cinco horas y batch_size sesenta y cuatro. El rendimiento se mantiene prácticamente constantes a partir de batch_size catorce. De nuevo, observamos una gran mejora de batch_size cuatro a batch_size doce.

La siguiente gráfica se corresponde con epoch 10.

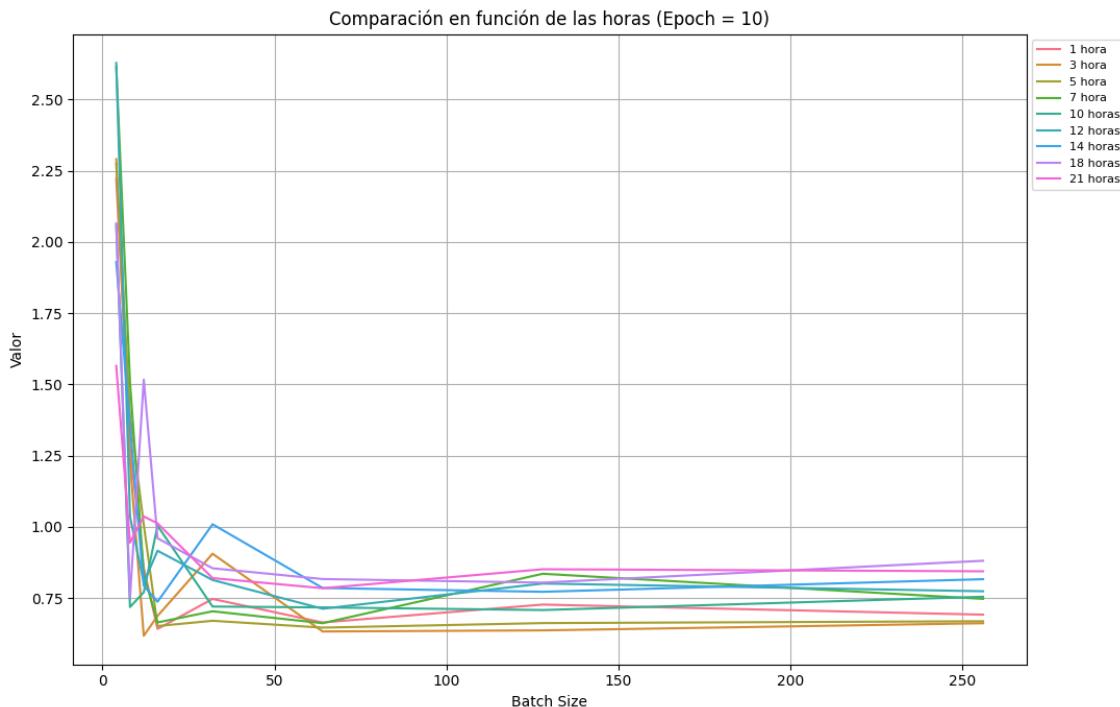


Tabla 4-27. Epoch 10 redes neuronales

En esta gráfica se puede observar que, para epoch igual a diez la única tendencia que se mantiene es la mejora de rendimiento de batch_size cuatro a doce, los resultados son más irregulares que para las epoch anteriores y no hay ninguna relación clara entre batch_size y rendimiento.

La siguiente gráfica se corresponde con epoch catorce.

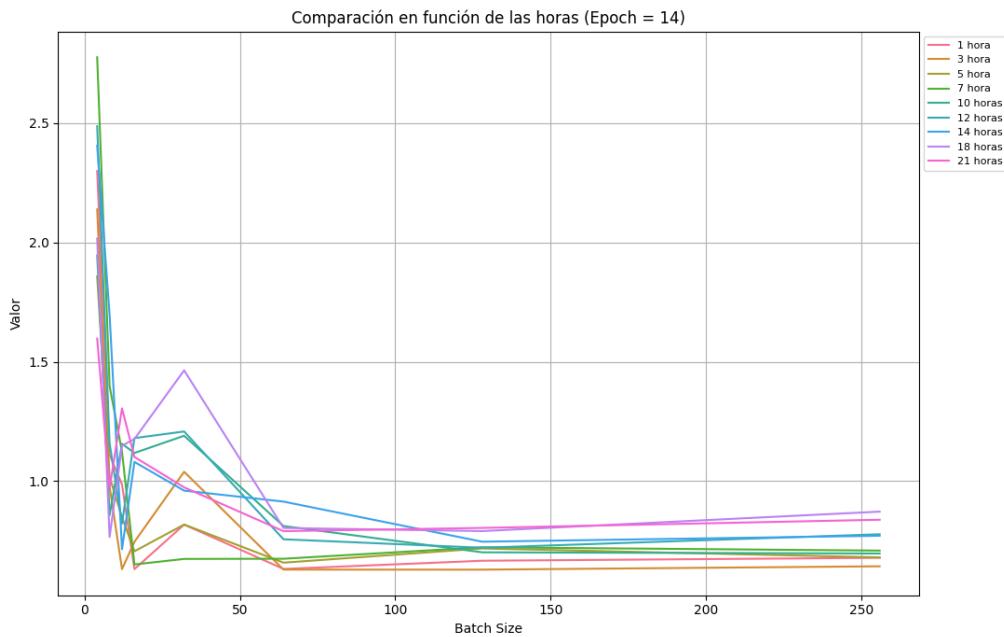


Tabla 4-28. Epoch 14 redes neuronales

La gráfica que se corresponde con epoch catorce es más irregular que la de epoch diez, en esta la mejora de batch_size cuatro a doce no se produce en todas las horas, el mejor rendimiento se obtiene en batch_size sesenta y cuatro a partir del cual el rendimiento se mantiene constante.

La siguiente gráfica se corresponde con epoch veinte.

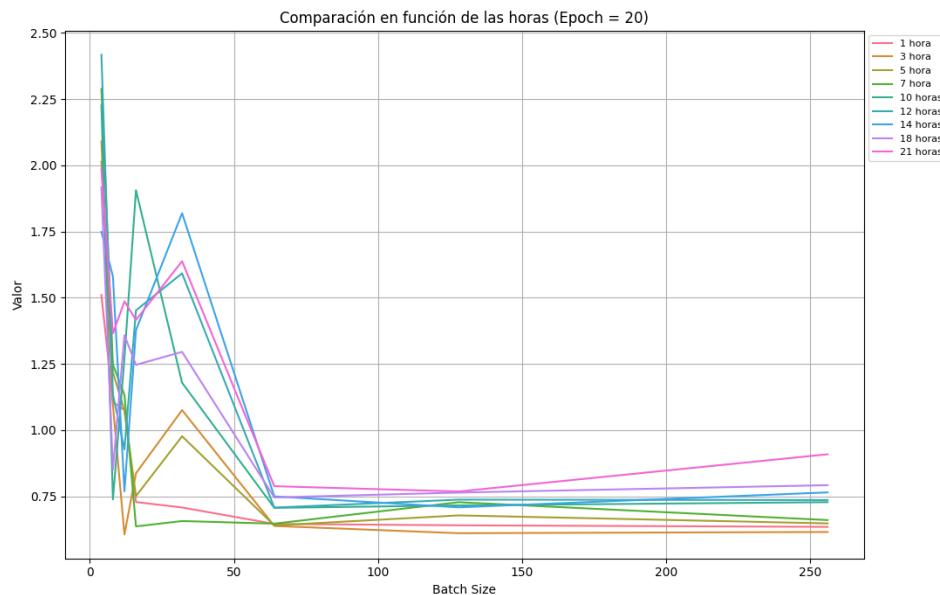


Tabla 4-29. Epoch 20 redes neuronales

Como podemos observar en la gráfica la irregularidad hasta el batch_size treinta y dos es muy alta, del treinta y dos al sesenta y cuatro se produce una gran mejora y a partir del sesenta y cuatro se vuelve constante.

La siguiente gráfica se corresponde con epoch cuarenta.

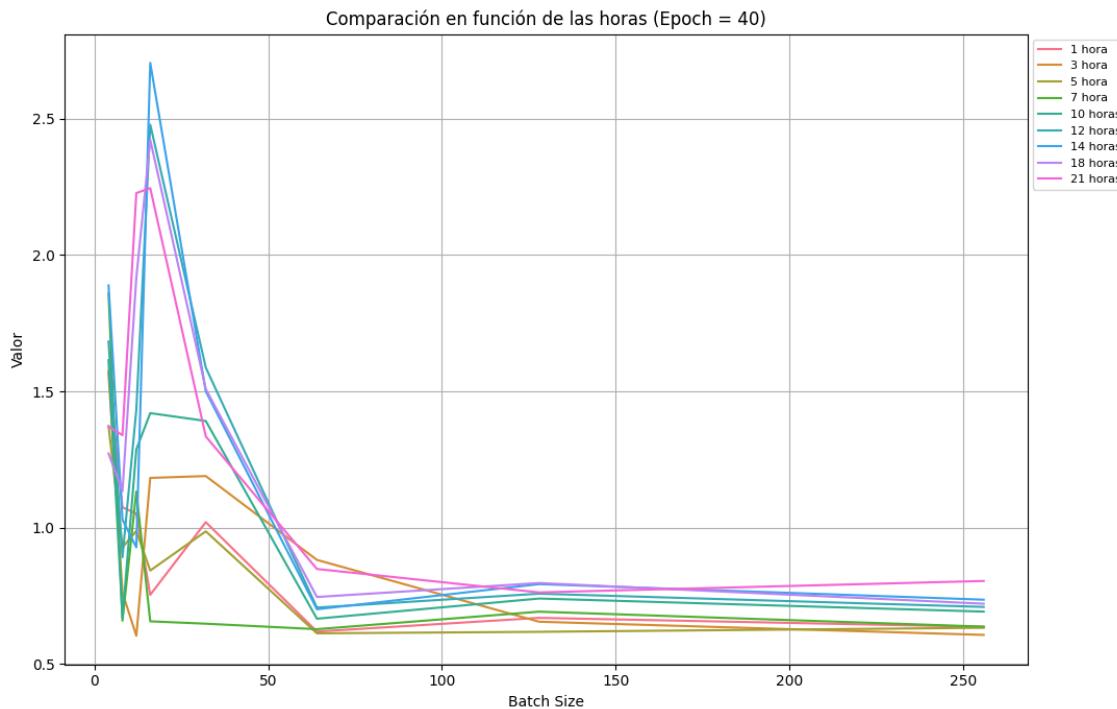


Tabla 4-30. Epoch 40 redes neuronales

De nuevo la gráfica presenta una gran irregularidad hasta el batch_size treinta y dos, del treinta y dos al sesenta y cuatro se produce una gran mejora y a partir del sesenta y cuatro se vuelve constante.

El mejor rendimiento obtenido es de 0,603 obtenido usando tres horas para hacer la predicción, siendo el epoch igual a cuarenta y el batch_size de doce.

4.3.2 Con Índice de Miedo-Codicia

Seguiré el mismo sistema que para el caso que no usa el índice de miedo-codicia, esta gráfica se corresponde con epoch cuatro.

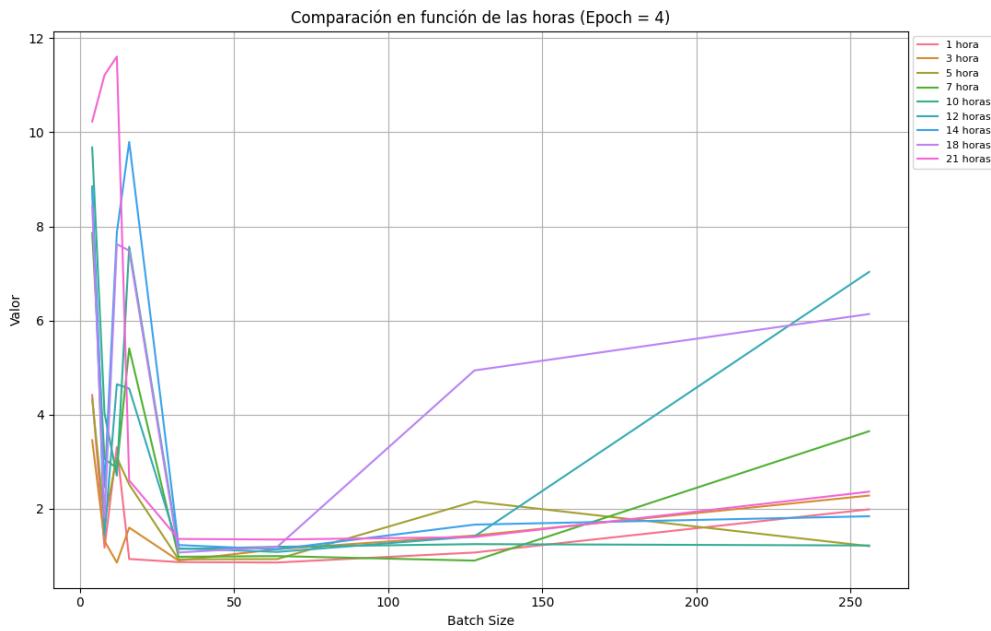


Tabla 4-31. Epoch 4 redes neuronales IMC

Como se puede observar en la gráfica el rendimiento es muy irregular, obteniéndose el mejor rendimiento con batch_size treinta y dos y sesenta y cuatro para la mayoría de las horas.

La siguiente gráfica se corresponde con epoch seis.

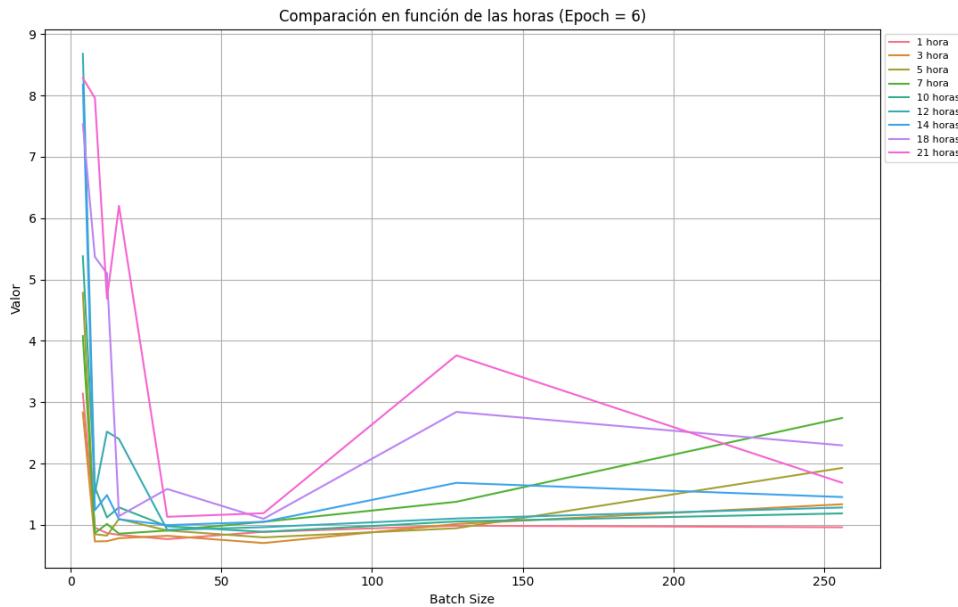


Tabla 4-32. Epoch 6 redes neuronales IMC

Se observa algo menos de irregularidad para este epoch que para epoch cuatro, sin embargo, los mejores valores también se concentran para batch_size treinta y dos y sesenta y cuatro.

La siguiente gráfica se corresponde con epoch diez.

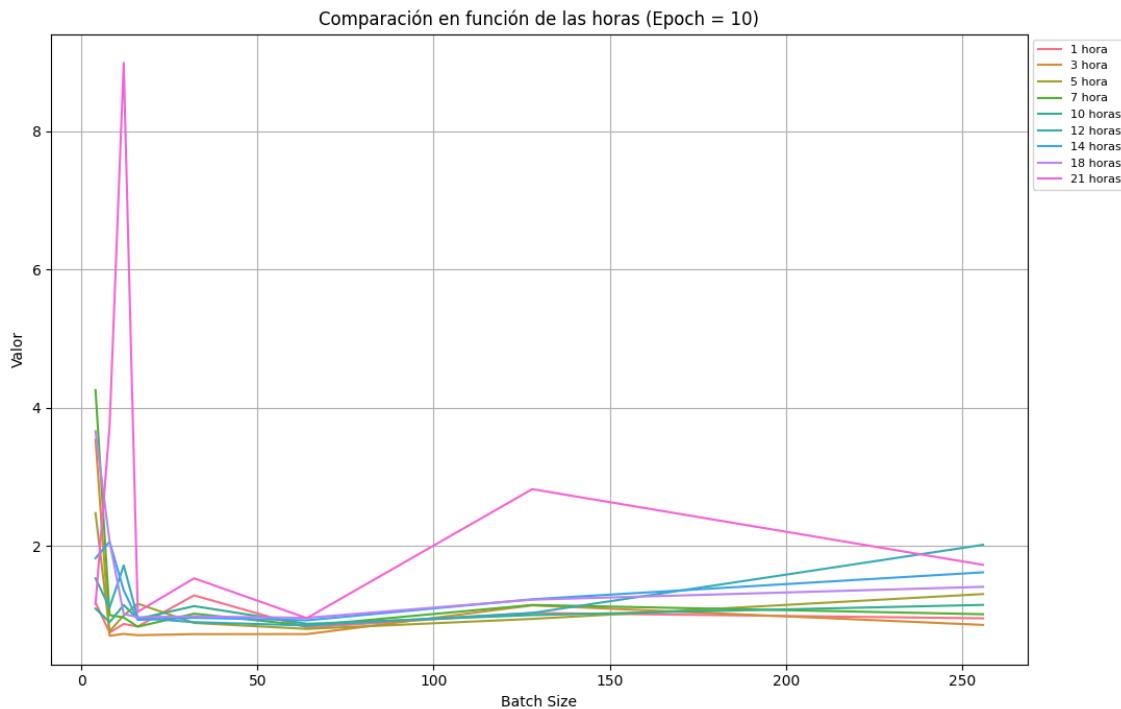


Tabla 4-33. Epoch 10 redes neuronales IMC

En esta gráfica los resultados se vuelven mucho más regulares para la mayoría de las cantidades de horas, excepto para veintiuno, a partir de sesenta y cuatro de epoch se observa cierto empeoramiento.

La siguiente gráfica se corresponde con epoch catorce.

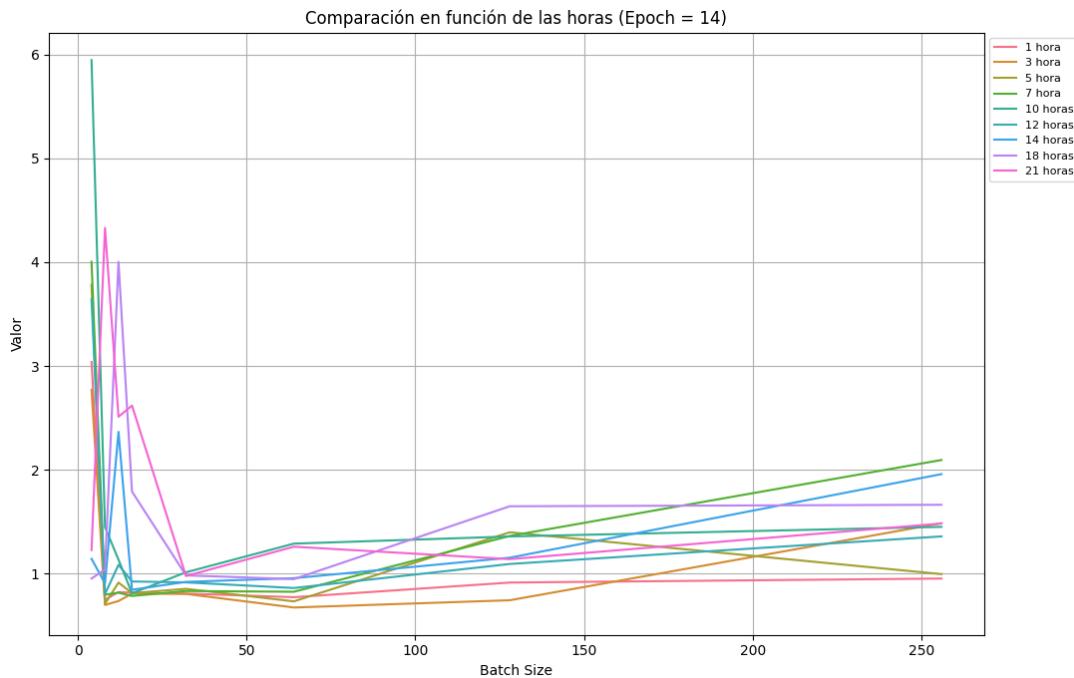


Tabla 4-34. Epoch 14 redes neuronales IMC

Se observa mucha irregularidad con valores pequeños de batch_size y una ligera pérdida de rendimiento a partir de batch_size sesenta y cuatro.

La gráfica se corresponde con epoch veinte.

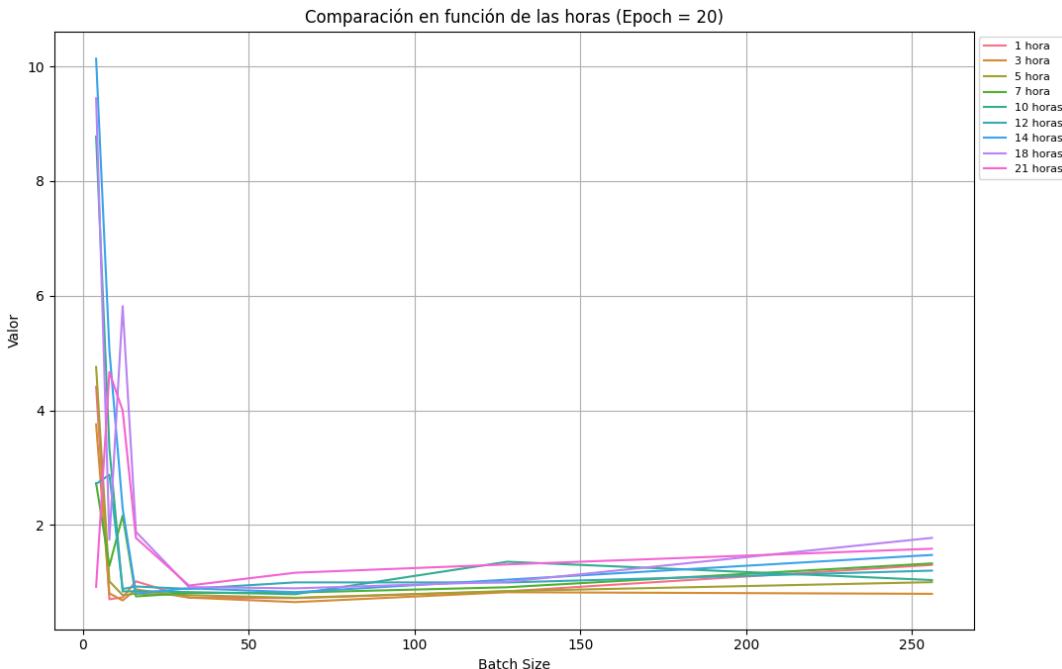


Tabla 4-35. Epoch 20 redes neuronales IMC

De nuevo, observamos mucha irregularidad con valores pequeños de batch_size y un rendimiento prácticamente constante a partir de batch_size treinta y dos.

La gráfica se corresponde con epoch cuarenta.

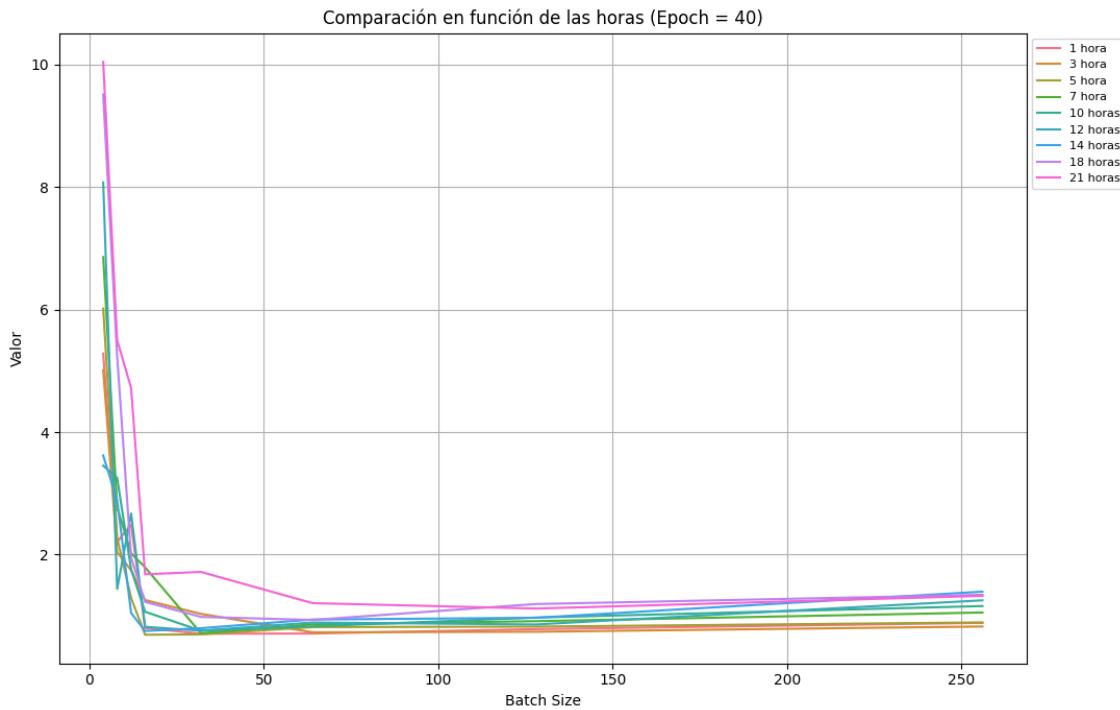


Tabla 4-36. Epoch 40 redes neuronales IMC

Podemos observar una gran mejora de rendimiento desde batch_size cuatro hasta batch_size dieciséis. A partir de este el rendimiento se mantiene constante para la mayoría de las cantidades de horas.

El mejor rendimiento obtenido es de 0,654 obtenido usando tres horas para hacer la predicción, siendo el epoch igual a veinte y el batch_size de sesenta y cuatro.

4.3.3 Conclusión

Tras analizar los datos obtenidos puedo concluir que la hipótesis relativa al índice de miedo-codicia no se cumple en las redes neuronales ya que las predicciones con los dataframes que lo usan tienen un menor rendimiento, tanto el mejor rendimiento como el rendimiento general.

Otra diferencia notable es la irregularidad que se puede observar al usar los datos que contienen el índice de miedo-codicia, además de esto los peores valores obtenidos usando los datos que contienen el índice de miedo-codicia son significativamente peores.

Estos factores me llevan a concluir que, en el caso las redes neuronales, el uso del índice de miedo-codicia solo introduce ruido en el modelo, empeorando la calidad de las predicciones.

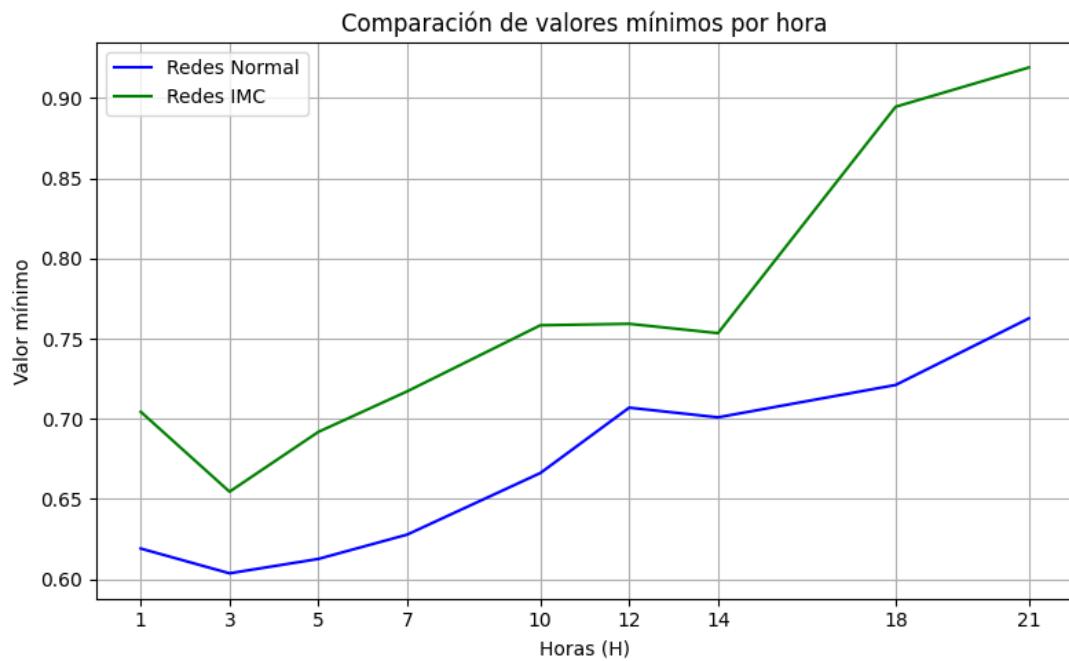


Tabla 4-37. Redes neuronales IMC vs Normal

4.4 Redes Neuronales LSTM

4.4.1 Sin Índice de Miedo-Codicia

Para visualizar los datos de rendimiento de las redes neuronales LSTM he decidido fijar el parámetro epoch y representar en la gráfica el rendimiento en el eje y, el batch_size en el eje X y el número de horas a través del color y forma de cada línea de la gráfica. La primera gráfica se corresponde con epoch cuatro.

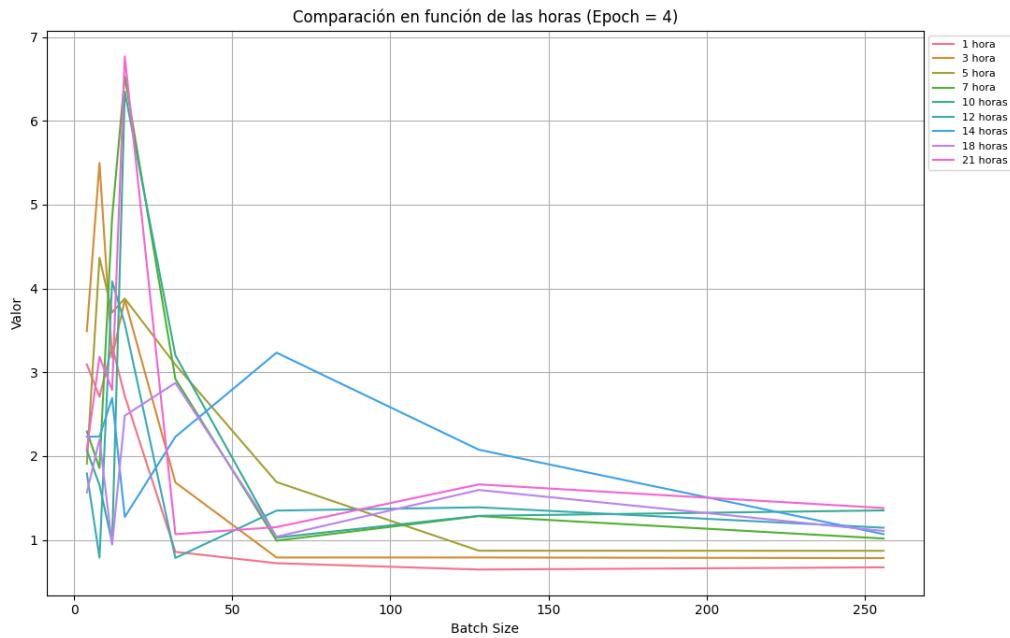


Tabla 4-38. Epoch 4 redes neuronales LSTM

En la gráfica se puede observar resultados muy irregulares con los batch_size más pequeños. Según van creciendo los batch_size el rendimiento se estabiliza y se vuelve más similar entre todas las cantidades de horas.

La siguiente gráfica se corresponde con epoch seis.

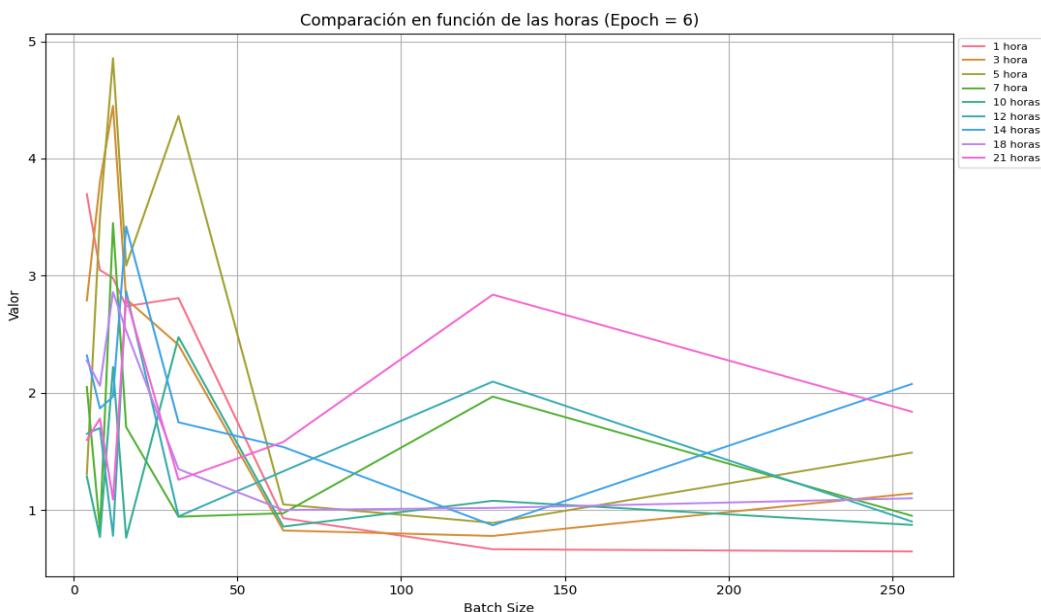


Tabla 4-39. Epoch 6 redes neuronales LSTM

En la gráfica se puede observar resultados muy irregulares con todos los batch_size. Los mejores rendimientos se obtienen con batch_size sesenta y cuatro para la mayoría de las cantidades de horas.

La siguiente gráfica se corresponde con epoch diez.

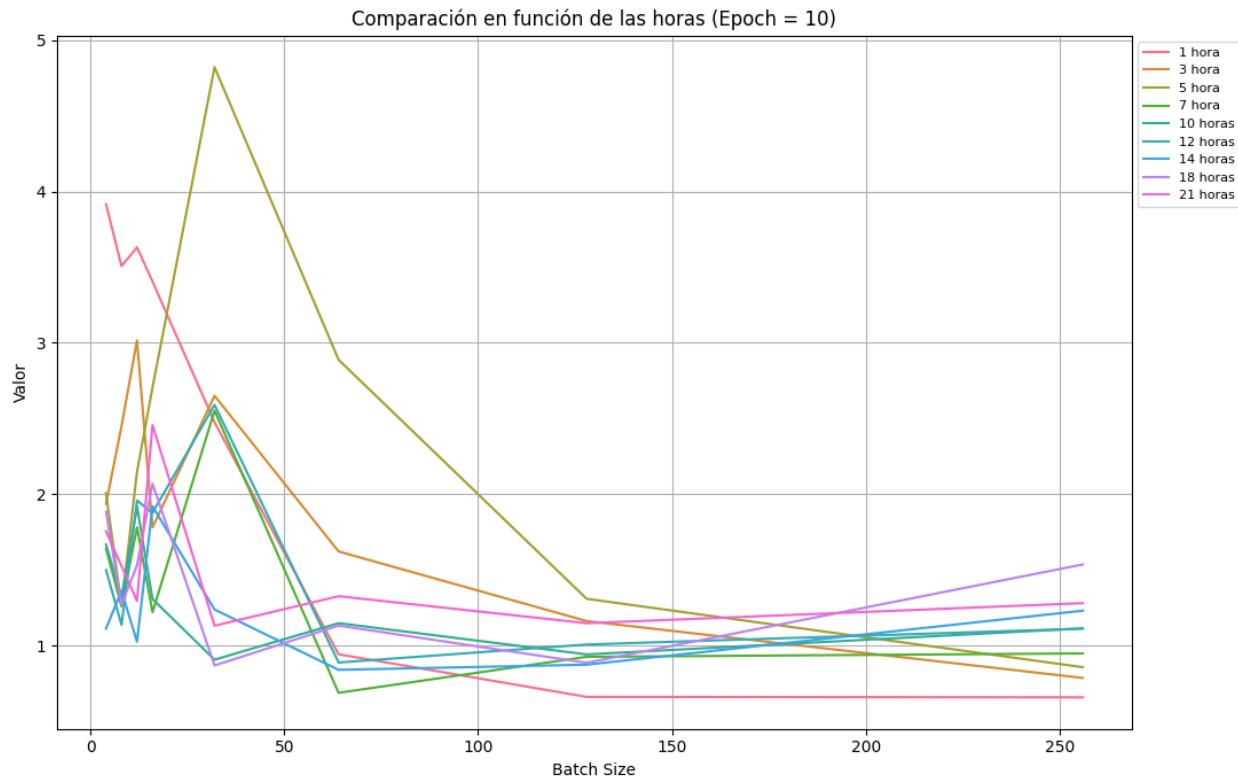


Tabla 4-40. Epoch 10 redes neuronales LSTM

En la gráfica se puede observar resultados muy irregulares con los batch_size más pequeños, los resultados se estabilizan a partir de batch_size ciento veintiocho.

La siguiente gráfica se corresponde con epoch catorce.

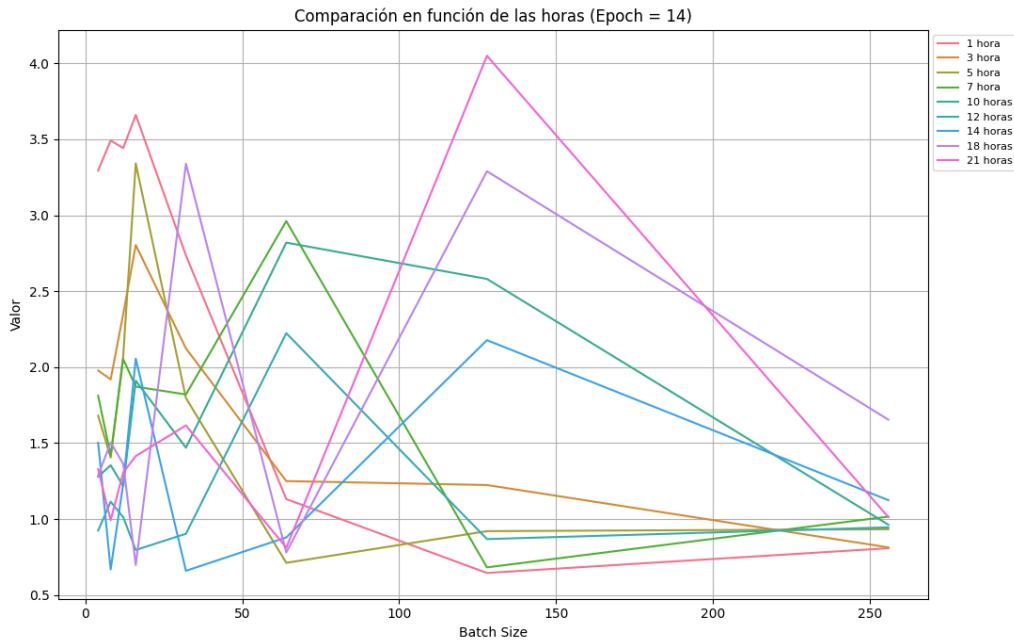


Tabla 4-41. Epoch 14 redes neuronales LSTM

En la gráfica se puede observar resultados muy irregulares con todos los batch_size. Esta gráfica no sigue ningún tipo de patrón y cada cantidad de horas obtiene su mejor rendimiento para un batch_size distinto.

La siguiente gráfica se corresponde con epoch veinte.

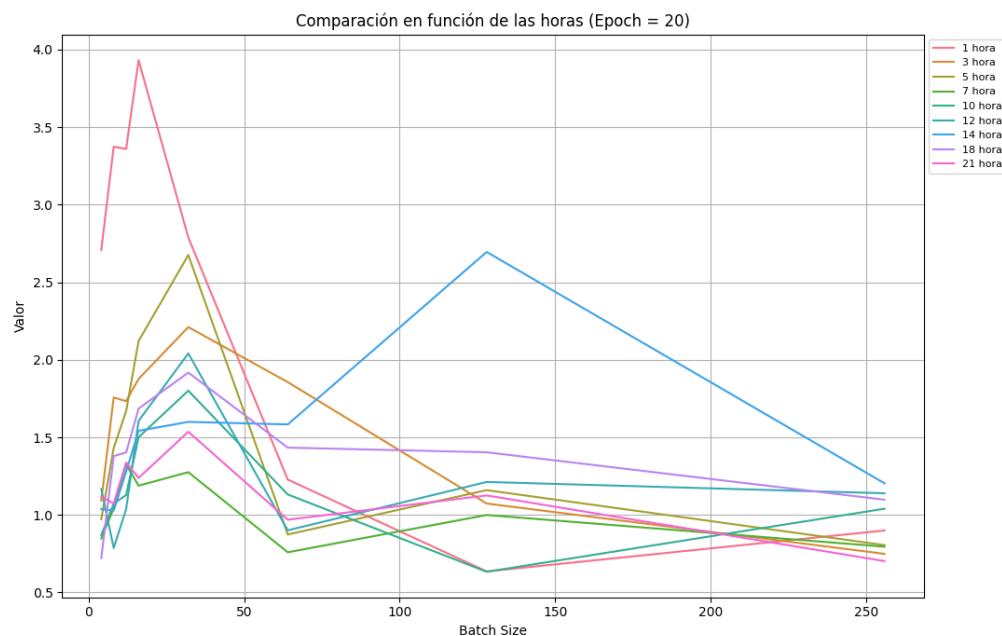


Tabla 4-42. Epoch 20 redes neuronales LSTM

En la gráfica se puede observar que los resultados sufren cierto empeoramiento hasta batch_size treinta y dos para posteriormente mejorar y estabilizarse.

La siguiente gráfica se corresponde con epoch veinte.

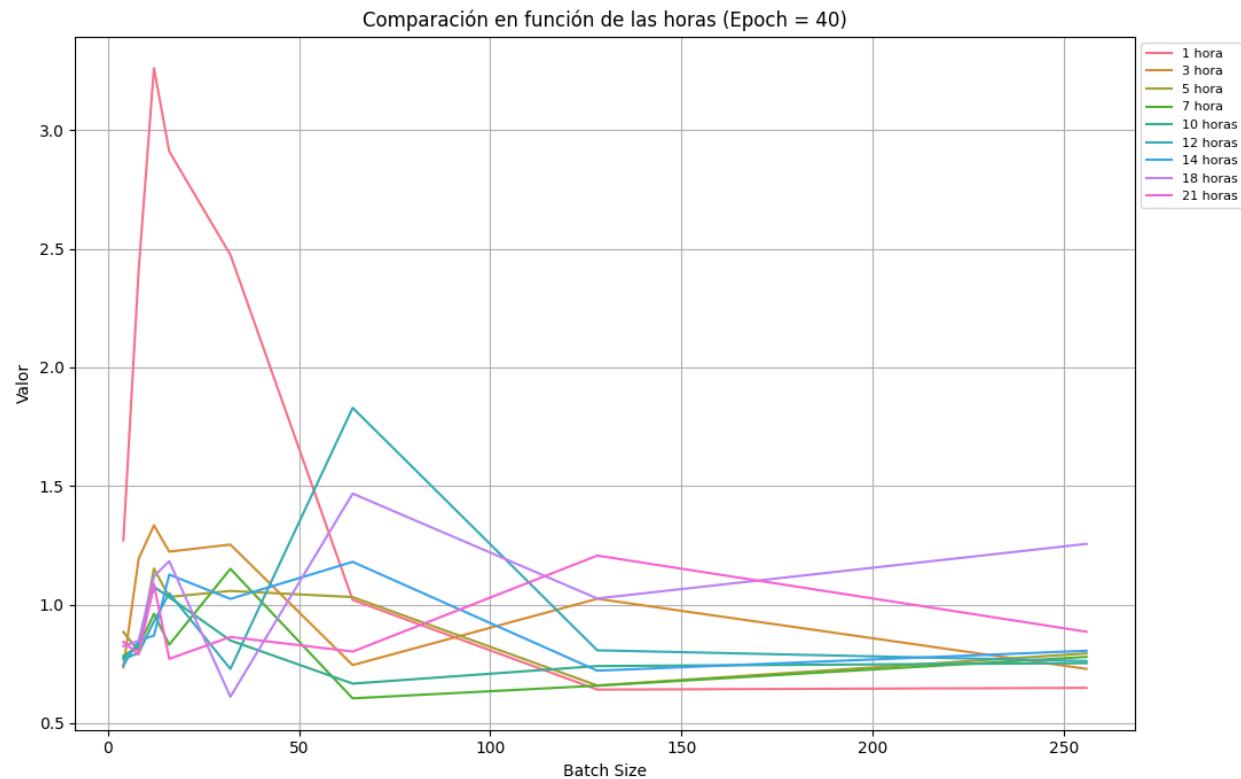


Tabla 4-43. Epoch 40 redes neuronales LSTM

En la gráfica se puede observar que los resultados se mantienen bastante constantes para los diferentes batch_size, también podemos observar mucha menos irregularidad que en el resto de epoch.

El mejor rendimiento obtenido es de 0,604 obtenido usando siete horas para hacer la predicción, siendo el epoch igual a cuarenta y el batch_size de sesenta y cuatro.

4.4.2 Con Índice de Miedo-Codicia

Seguiré el mismo sistema que para el caso que no usa el índice de miedo-codicia, esta gráfica se corresponde con epoch cuatro.

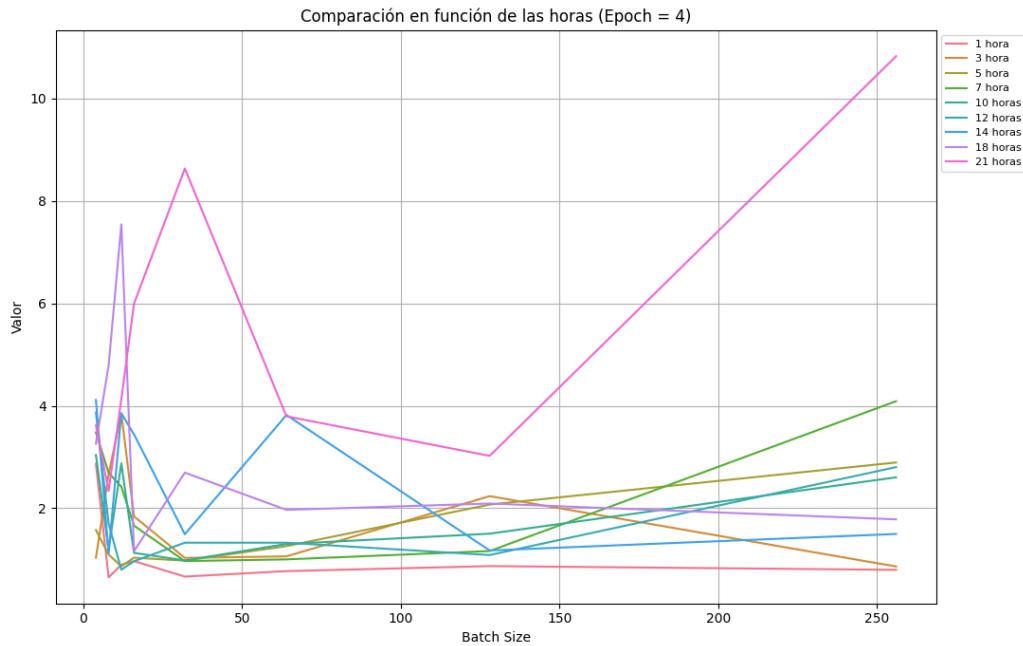


Tabla 4-44. Epoch 4 redes neuronales LSTM IMC

En la gráfica se puede observar resultados muy irregulares para las cantidades más grandes de horas. Para cantidades de horas más pequeñas los resultados se vuelven bastante constantes a partir de batch_size treinta y dos.

La siguiente gráfica se corresponde con epoch seis.

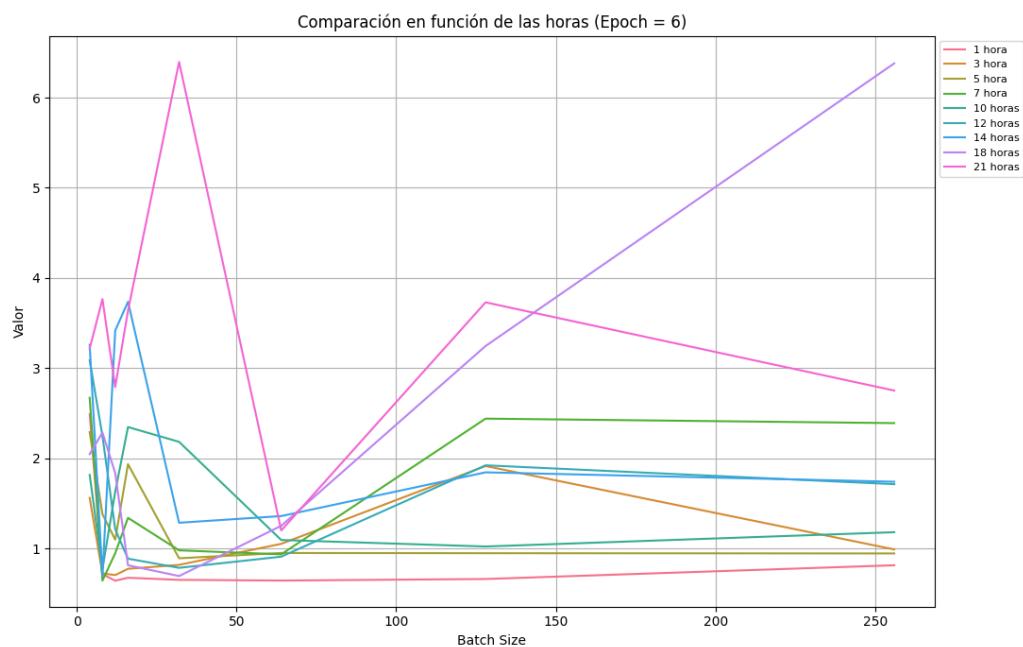


Tabla 4-45. Epoch 6 redes neuronales LSTM IMC

En la gráfica se puede observar resultados muy irregulares para las cantidades grandes de horas. Para cantidades de horas pequeñas los resultados se vuelven bastante constantes a partir de batch_size treinta y dos.

La siguiente gráfica se corresponde con epoch diez.

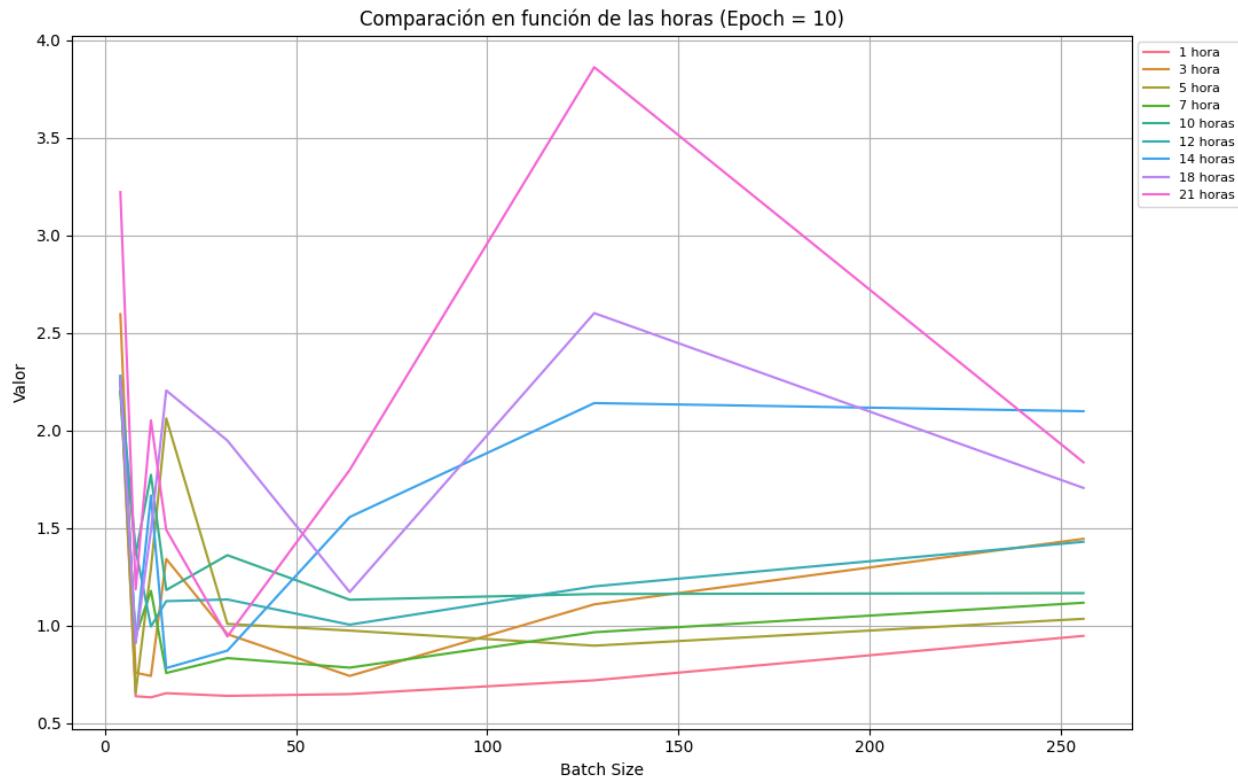


Tabla 4-46. Epoch 10 redes neuronales LSTM IMC

En la gráfica se puede observar resultados muy irregulares para las cantidades grandes de horas. Para cantidades de horas pequeñas los resultados se puede observar cierto empeoramiento a partir de batch_size sesenta y cuatro.

La siguiente gráfica se corresponde con epoch catorce.

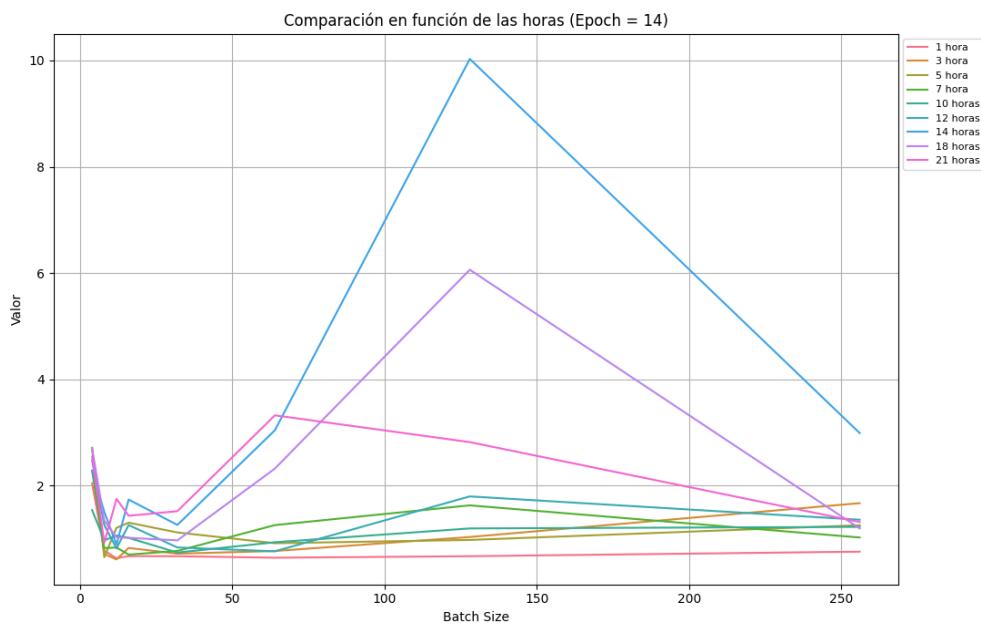


Tabla 4-47. Epoch 14 redes neuronales LSTM IMC

En la gráfica se puede observar resultados muy irregulares para las cantidades grandes de horas. Para cantidades de horas pequeñas los resultados se mantienen constantes a partir de batch_size dieciséis.

La siguiente gráfica se corresponde con epoch veinte.

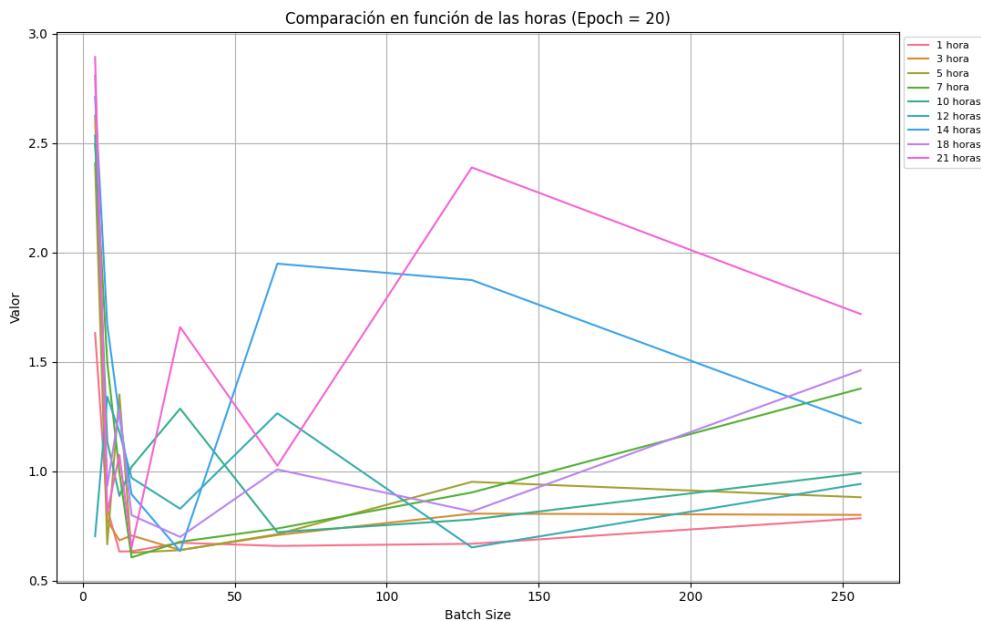


Tabla 4-48. Epoch 20 redes neuronales LSTM IMC

En la gráfica se puede observar resultados muy irregulares para las cantidades grandes de horas. Para cantidades de horas pequeñas los resultados empeoran significativamente a partir de batch_size sesenta y cuatro.

La siguiente gráfica se corresponde con epoch cuarenta.

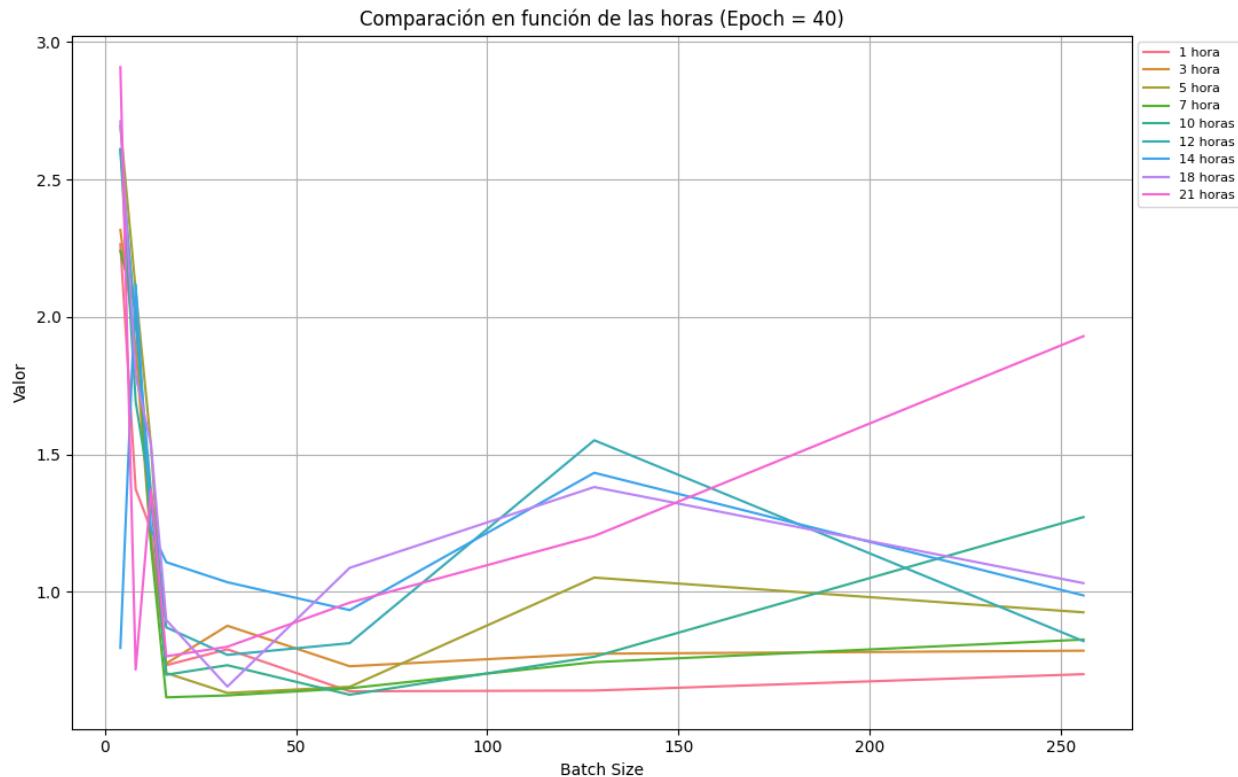


Tabla 4-49. Epoch 40 redes neuronales LSTM IMC

En la gráfica se puede observar resultados muy irregulares para las cantidades grandes de horas. Para cantidades de horas pequeñas los resultados se mantienen constantes a partir de batch_size sesenta y cuatro.

El mejor rendimiento obtenido es de 0,605 obtenido usando siete horas para hacer la predicción, siendo el epoch igual a veinte y el batch_size dieciséis.

4.4.3 Conclusión

Tras analizar los datos obtenidos puedo concluir que la hipótesis relativa al índice de miedo-codicia se cumple en las redes neuronales LSTM ya que las predicciones con los dataframe que lo usan tienen un mejor rendimiento.

Una diferencia notable es la irregularidad que se puede observar para algunas cantidades de horas al usar los datos que contienen el índice de miedo-codicia.



Tabla 4-50. LSTM IMC vs Normal

4.5 Regresión Simbólica

Para visualizar los datos de rendimiento de la regresión simbólica he decidido representar en la gráfica el rendimiento en el eje y, y el número de iteraciones en el eje X . Cada gráfica se corresponde con unas condiciones.

4.5.1 Sin Índice de Miedo-Codicia ni Constantes

En la gráfica se puede observar una gran disminución del rendimiento al aumentar las iteraciones. Esta disminución del rendimiento me llevo a en la iteración número 5600 detener el algoritmo y continuar ejecutándolo con nuevos genes generados aleatoriamente. El mejor rendimiento obtenido fue de 1,016, sin embargo, el gen que lo obtuvo no es válido ya que su predicción es siempre el precio de cierre de la hora anterior, el mejor rendimiento obtenido de un gen valido es de 1,106.

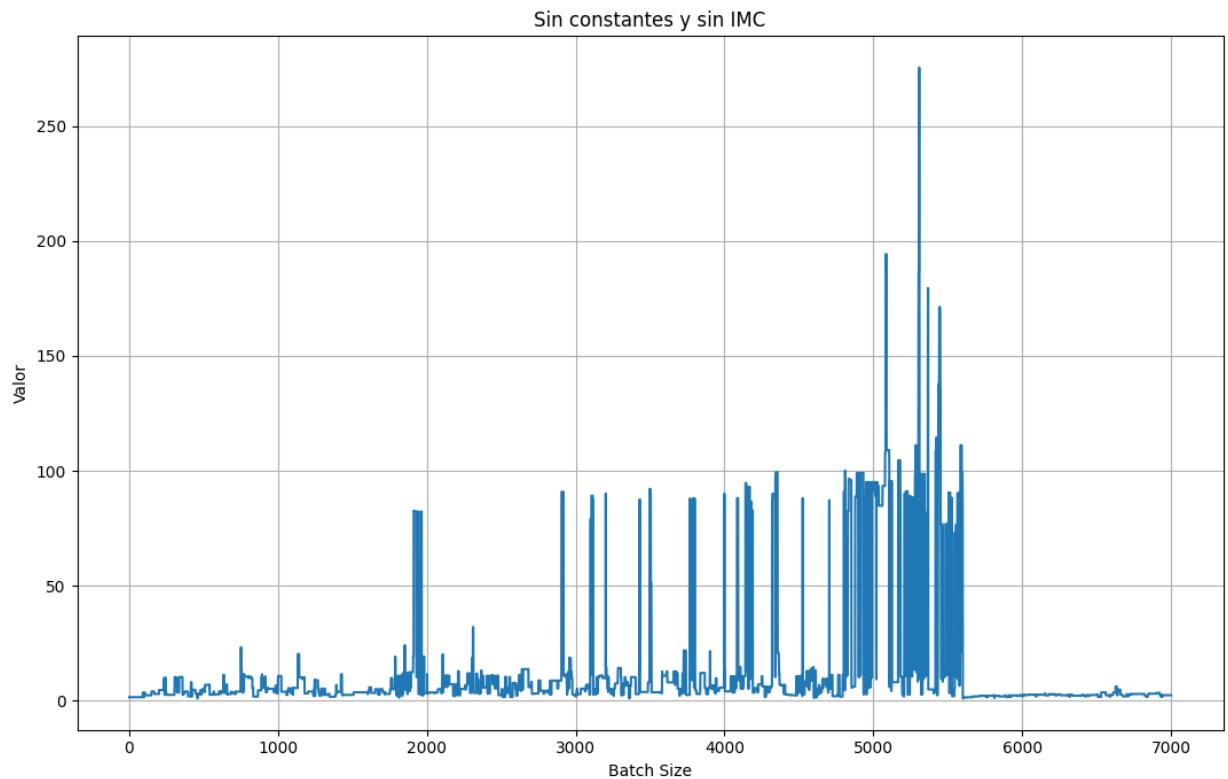


Tabla 4-51. Regresión Simbólica sin constantes y sin IMC

4.5.2 Sin Índice de Miedo-Codicia con Constantes

En la gráfica se puede observar una gran disminución del rendimiento al aumentar las iteraciones. Esta disminución del rendimiento me llevo a en la iteración número 3000 detener el algoritmo y continuar ejecutándolo con nuevos genes generados aleatoriamente, sin embargo, tras esta nueva población inicial no se vuelve a observar el empeoramiento del rendimiento, esto se debe a que al observar rendimientos superiores al 100 de manera sostenida generaba una nueva población inicial. El mejor rendimiento obtenido fue de 1,019 que fue obtenido por un gen válido.

También se puede observar que el algoritmo llega a obtener algunos rendimientos disparatados como el de 14000.

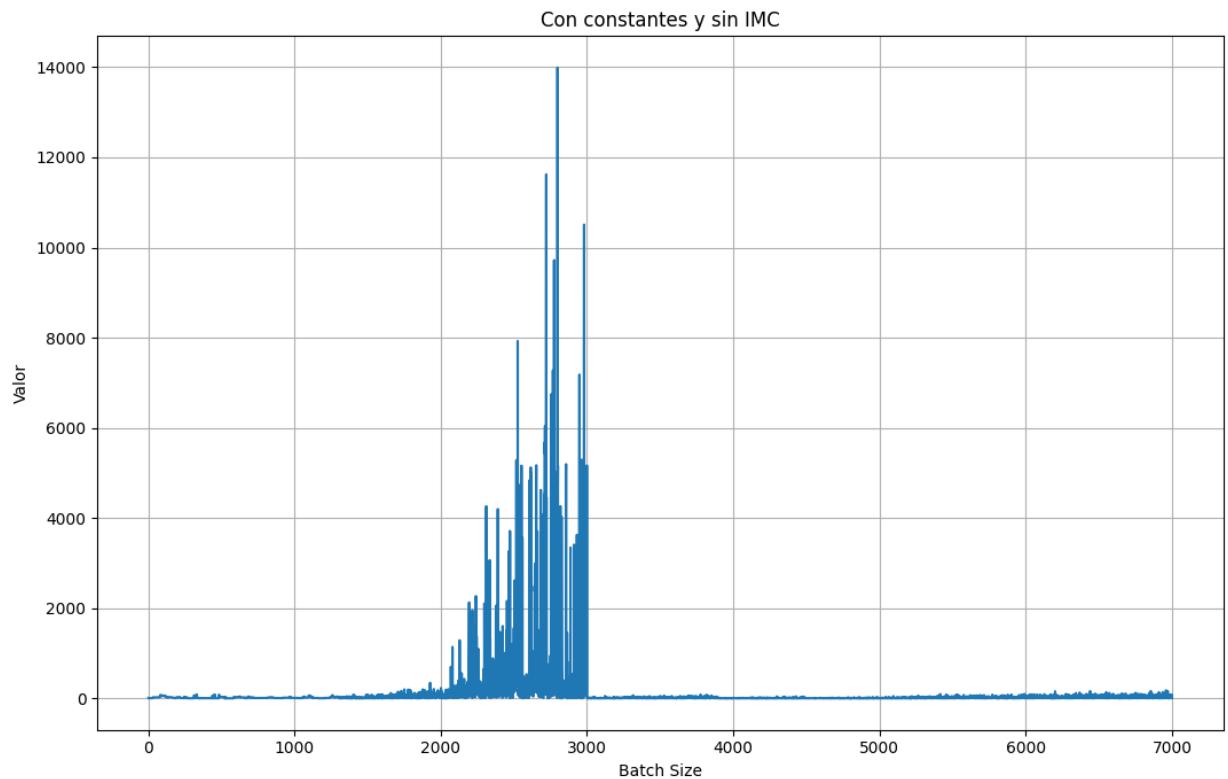


Tabla 4-52. Regresión Simbólica con constantes y sin IMC

4.5.3 Con Índice de Miedo-Codicia sin Constantes

En la primera gráfica se puede observar un rendimiento disparatado del 14.000.000. Este rendimiento no permite observar con claridad lo que sucede en el resto de las iteraciones por lo que he generado la segunda gráfica que contiene solo los registros por debajo de 1000. En la segunda gráfica solo quedaron 5756 de las 7000 iteraciones originales y podemos observar una gran irregularidad en los resultados con cambios de rendimiento muy bruscos y unos resultados generales bastante alejados del resto de algoritmos.

El mejor rendimiento obtenido fue de 1,043 que fue obtenido por un gen válido.

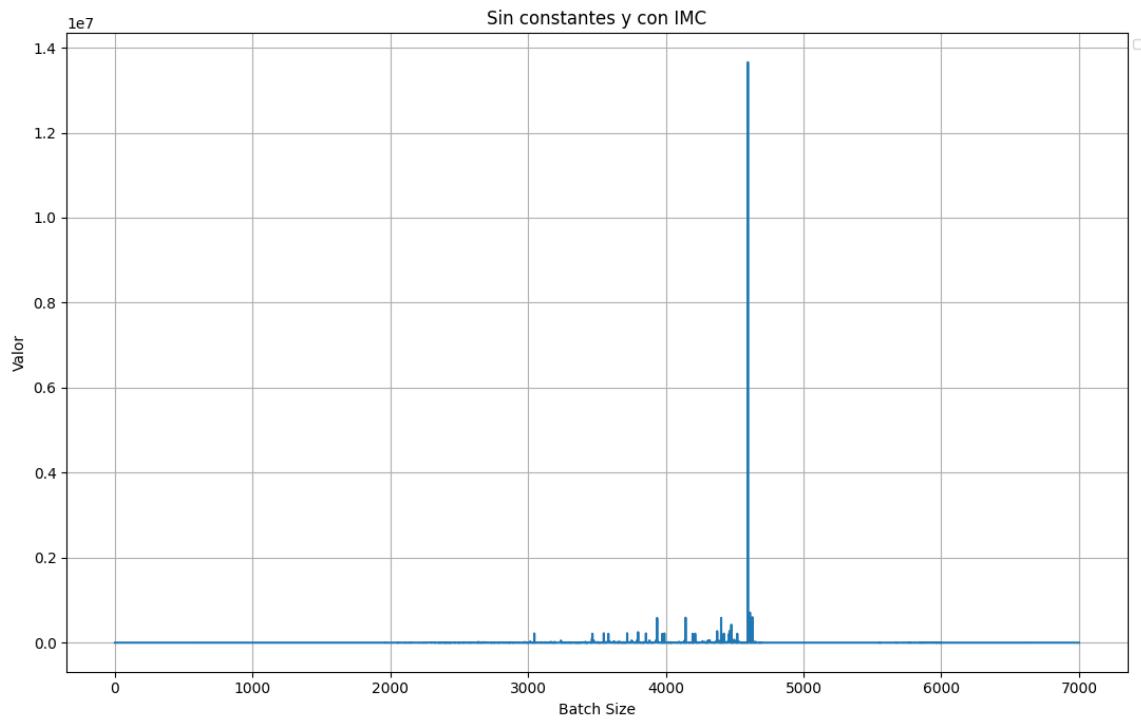


Tabla 4-53.. Regresión Simbólica sin constantes y con IMC completa

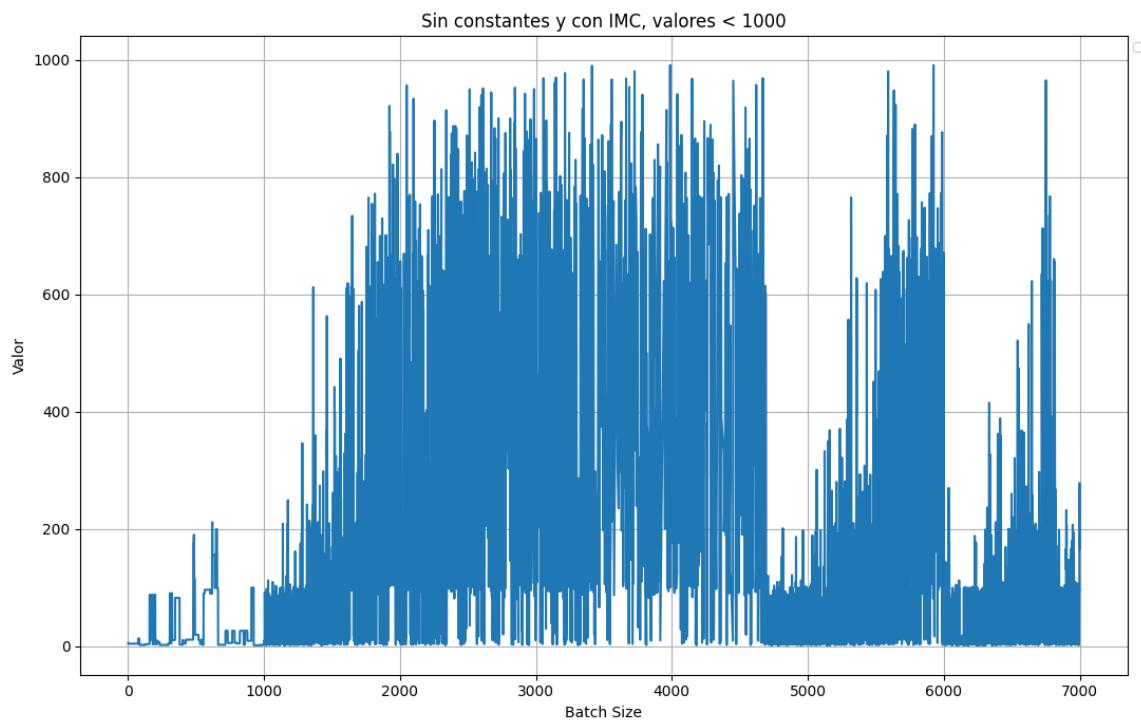


Tabla 4-54. Regresión Simbólica sin constantes, con IMC y valor < 1000

4.5.4 Con Índice de Miedo-Codicia con Constantes

En la gráfica se puede observar una gran disminución del rendimiento al aumentar las iteraciones. Sin embargo, podemos observar que el rendimiento se mantiene bastante estable hasta llegar a las 5000 iteraciones a partir de las cuales empieza a presentar peores resultados y más irregularidad. El mejor rendimiento obtenido fue de 1,016, sin embargo, el gen que lo obtuvo no es válido ya que su predicción es siempre el precio de cierre de la hora anterior, el mejor rendimiento obtenido de un gen valido es de 1,017.

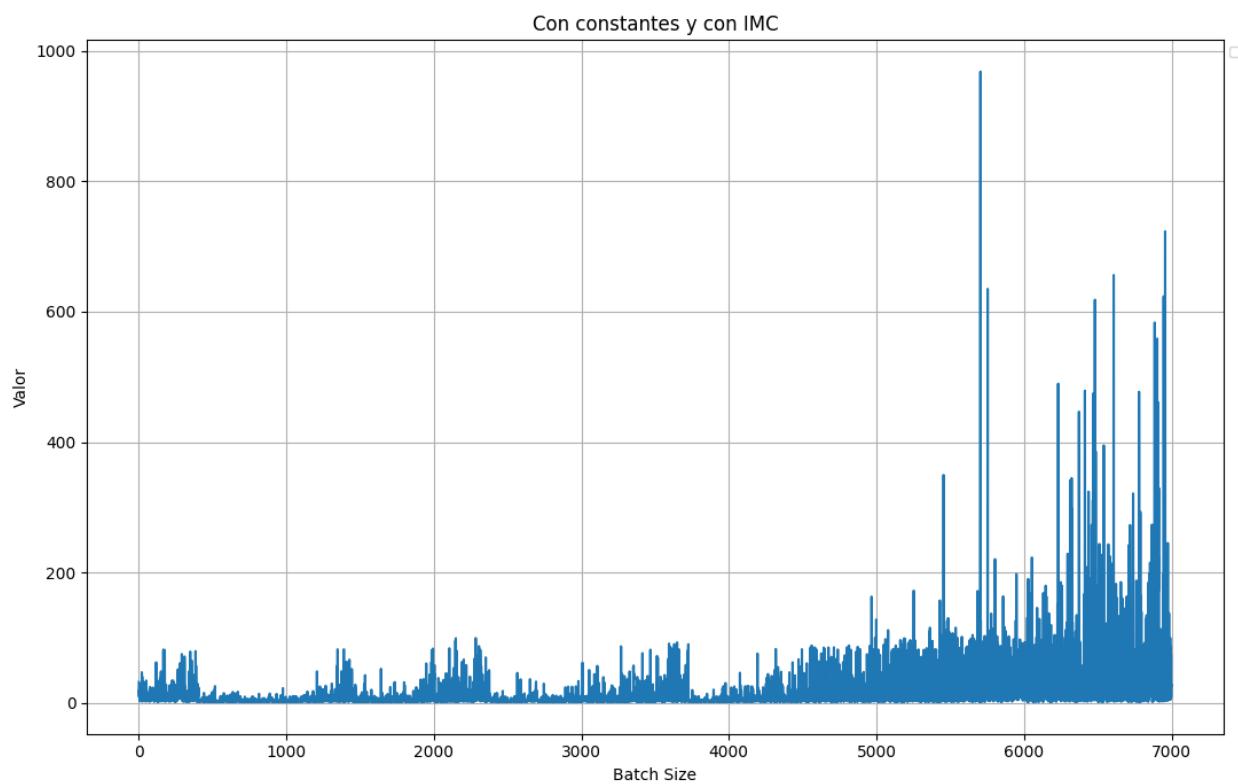


Tabla 4-55. Regresión Simbólica con constantes y con IMC

4.5.5 Conclusión

Como se puede observar en la figura, el rendimiento utilizando constantes es muy superior al obtenido sin utilizarlas. La hipótesis del índice miedo-codicia se cumple para el algoritmo de regresión simbólica, sin embargo, los modelos que lo utilizan presentan una gran irregularidad en sus resultados de rendimiento, llegando a obtener resultados

ridículos como el de 1,4e7. Además de esto la diferencia de rendimiento entre utilizarlo y no utilizarlo es mínima, un 0,017.

Observando los resultados puede inferirse también la importancia del valor de cierre de la hora anterior ya que es el único presente en todos los genes que presentan un gran rendimiento. En el caso de no utilizar IMC sería la posición 27 y en el caso de utilizarlo la 33.

El mejor rendimiento obtenido es de 1,017 y fue obtenido por el modelo que usaba el índice de miedo-codicia y las constantes. Esto se debe a que los rendimientos en las posiciones 1, 2 y 6 provienen de genes no válidos.

Top 10 mejores valores únicos:		
	iteracion	valor
16807	2808	1.016758
457	458	1.016758
16013	2014	1.017357
13196	6197	1.019015
17296	3297	1.020025
18614	4615	1.024363
15655	1656	1.024885
17968	3969	1.025888
15710	1711	1.026408
10109	3110	1.034252

dataframe_nombre	
16807	Con IMC y con constantes
457	Sin IMC y sin constantes
16013	Con IMC y con constantes
13196	Sin IMC y con constantes
17296	Con IMC y con constantes
18614	Con IMC y con constantes
15655	Con IMC y con constantes
17968	Con IMC y con constantes
15710	Con IMC y con constantes
10109	Sin IMC y con constantes

Tabla 4-56. Regresión Simbólica top 10 resultados

Capítulo 5 - Conclusiones y trabajo futuro

5.1 Conclusiones

Como conclusión evaluaré el rendimiento de los algoritmos conjuntamente.

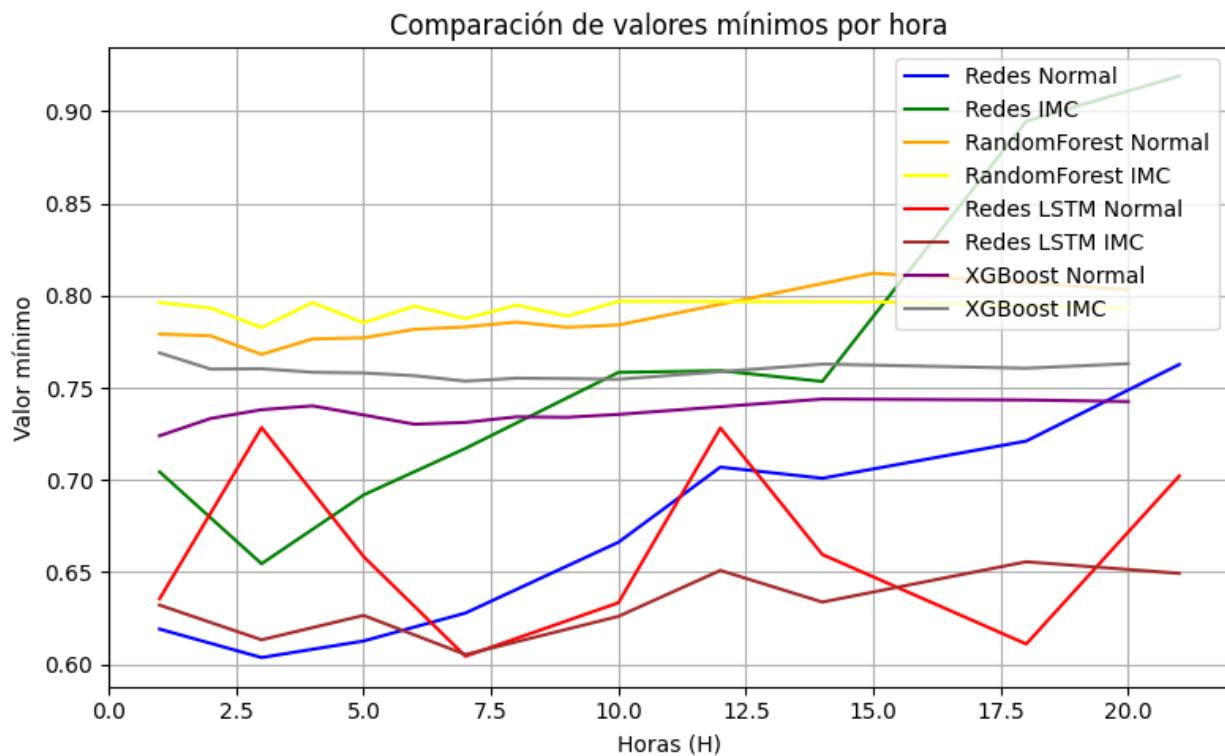


Tabla 5-1. Comparación de algoritmos

En la gráfica se puede observar que el rendimiento de las redes neuronales y las redes neuronales LSTM es significativamente mejor que el del resto de algoritmos. En el caso de la regresión simbólica que no aparece en la gráfica su rendimiento es significativamente peor ya que no consiguió bajar del 1%.

Basándonos en los datos obtenidos podemos concluir que las redes neuronales y las redes neuronales LSTM son los algoritmos más adecuados para la predicción de series temporales. El mejor resultado obtenido fue por las redes neuronales y fue de 0,603, quedándose muy cerca también el segundo mejor resultado que fue obtenido por las redes neuronales LSTM y fue de 0,604.

En el caso de la hipótesis relativa al índice de miedo-codicia podemos observar que solo se cumple para algunos algoritmos como las redes neuronales LSTM y la Regresión Simbólica y parcialmente para Random Forest, sin embargo, para el resto de los algoritmos, XGBoost y redes neuronales podemos observar que no se cumple. Por lo tanto, no podemos concluir que mejore las capacidades predictivas de los algoritmos de manera general, ya que solo proporciona una mejora para algunos algoritmos y no es una mejora muy significativa.

5.2 Trabajo Futuro

Algunos aspectos que me gustaría implementar en el futuro sería una media entre los resultados de los algoritmos, entre todos y también por parejas o tríos, para comprobar si la media entre las predicciones de los modelos obtiene mejores resultados que estos individualmente.

También me gustaría incluir mejoras en el algoritmo de regresión simbólica implementando un mayor número de operaciones, lo que permitiría una mayor complejidad de las ecuaciones y probablemente mayor rendimiento. Además, me gustaría implementar mutaciones por recombinación, es decir tomar dos genes y recombinarlos en uno nuevo.

Otro aspecto que me gustaría desarrollar es la predicción de precio más horas vista, hasta ahora solo predigo el precio de cierre de la hora siguiente, una buena ampliación sería poder predecir las 8, 16 y 24 horas siguientes.

5.3 Conclusions

In conclusion, I will evaluate the performance of the algorithms together.

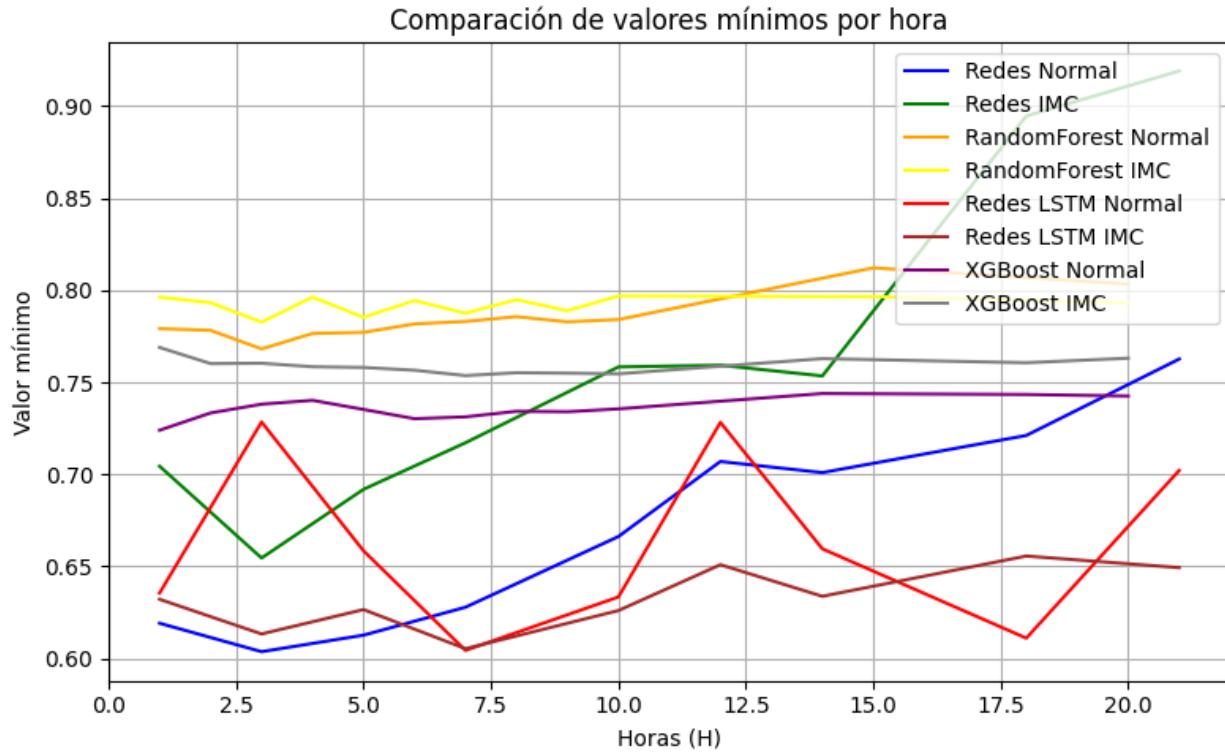


Tabla 5-2. Comparison of algorithms

The graph shows that the performance of neural networks and LSTM neural networks is significantly better than that of the other algorithms. In the case of symbolic regression, which is not shown in the graph, its performance is significantly worse, as it failed to fall below 1%.

Based on the data obtained, we can conclude that neural networks and LSTM neural networks are the most suitable algorithms for time series prediction. The best result obtained by neural networks was 0.603, also very close behind was the second-best result, which was obtained by LSTM neural networks, which was 0.604.

Regarding the hypothesis regarding the fear-greed index, we can see that it is only true for some algorithms, such as LSTM neural networks and Symbolic Regression, and partially true for Random Forest. However, for the rest of the algorithms, XGBoost, and neural networks, we can see that it is not true. Therefore, we cannot conclude that it improves the predictive capabilities of algorithms in general, since it only provides an improvement for some algorithms and it is not a very significant improvement.

5.4 Future Work

Some aspects I would like to implement in the future would be an average of the algorithms' predictions, both across all models and in pairs or triplets, to see if the average of the models' predictions yields better results than the model's individual predictions.

I would also like to include improvements to the symbolic regression algorithm by implementing a greater number of operations, which would allow for greater complexity of the equations and better performance. Additionally, I would like to implement recombination mutations, that is, taking two genes and recombining them into a new one.

Another aspect I would like to develop is price prediction over time. So far, I have only predicted the closing price for the next hour. A good extension would be to be able to predict the next 8, 16, and 24 hours.

BIBLIOGRAFÍA

- [1] T. Mucci. <https://www.ibm.com/think/topics/history-of-artificial-intelligence>.
- [2] https://en.wikipedia.org/wiki/History_of_artificial_intelligence.
- [3] https://en.wikipedia.org/wiki/Timeline_of_artificial_intelligence.
- [4] https://es.wikipedia.org/wiki/Inteligencia_artificial.
- [5] <https://blog.hubspot.es/marketing/tipos-inteligencia-artificial>.
- [6] https://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico.
- [7] https://es.wikipedia.org/wiki/Random_forest.
- [8] https://en.wikipedia.org/wiki/Random_forest.
- [9] <https://es.wikipedia.org/wiki/XGBoost>.
- [10] [https://en.wikipedia.org/wiki/Neural_network_\(machine_learning\)](https://en.wikipedia.org/wiki/Neural_network_(machine_learning)).
- [11] <https://es.wikipedia.org/wiki/Retropropagaci%C3%B3n>.
- [12] <https://codificandobits.com/blog/funcion-de-activacion/>.
- [13] https://en.wikipedia.org/wiki/History_of_artificial_neural_networks.
- [14] https://es.wikipedia.org/wiki/Memoria_larga_a_corto_plazo.
- [15] C. Springer, «Makke, N., Sadeghi, M.A., Chawla, S. (2022). Symbolic Regression for Interpretable Scientific Discovery. In: Sachdeva, S., Watanobe, Y., Bhalla, S. (eds) Big-Data-Analytics in Astronomy, Science, and Engineering. BDA 2021. Lecture Notes in Computer Scienc».

[16] https://en.wikipedia.org/wiki/Symbolic_regression.