

Inteligencia Artificial aplicada a la predicción de precio de Insectos y Peces en A.C.N.H.



Jaime Alonso Fernández
Aprendizaje Automático y Big Data
Universidad Complutense de Madrid
Grado Ingeniería del Software plan 2019
Prof: Marta Caro

Índice

1. Descripción del Problema:	1
2. Análisis del dataset:	1
2.1. Filtración de Atributos	3
2.2. Atributos empleados	5
2.3. Transformación de Variables Nominales a Numéricas:	6
2.4. Transformación de Spawn Rates:	12
3. Empleo de Modelos de IA:	12
3.1. Modelos de Aprendizaje Automático Clásico	12
3.1.1. Modelo de Regresión Lineal	12
3.1.2. Modelo de Regresión Polinómica	13
3.1.3. Modelo de Regresión Logística	13
3.1.4. Conclusión:	13
3.2. Modelos de Aprendizaje Automático Moderno	14
3.2.1. Red Neuronal Feed Forward (FNN):	14
3.2.2. Red Neuronal Convolutiva (CNN):	14
3.2.3. Red Neuronal Long-Short Term Memory (LSTM):	15
3.2.4. Transformers:	15
3.2.5. Funciones de Activación:	15
3.2.6. Conclusión:	16
3.3. Implementación de Modelos de Aprendizaje Clásico:	17
3.3.1. Clase de Modelos de Regresión:	17
3.3.2. Clase Modelo de Regresión Lineal	17
3.3.1 Clase de Modelo de Regresión Polinomial	19
3.3.4. Análisis y Conclusión	21
3.4. Implementación de Modelos de Aprendizaje Moderno:	23
3.4.1. Implementación de FNN	23
3.4.2. Implementación de LSTM	25
3.4.3. Conclusiones	27
4. Empleo de Modelos de XAI:	28
4.1. Implementación de ALE	28
4.2. Implementación de SHAP	31
5. Pruebas de datos	34
6. Conclusiones	34
7. Referencias	36

1. Descripción del Problema:

Este proyecto tratará de resolver el algoritmo de establecimiento del precio de las diferentes criaturas (peces e insectos) a través de variables como la zona de aparición, las fechas de aparición o los horarios de aparición entre otras, en el videojuego de Nintendo: Animal Crossing New Horizons, para la consola Nintendo Switch.

Para ello he obtenido a través de Kaggle dos datasets de cada uno de los tipos de criaturas dentro del juego. Comenzaré transformando estos datasets de manera que pueda unirlos en un mismo dataset, para posteriormente realizar diferentes transformaciones en función del tipo y formato de las variables descritas.

Una vez establecido el dataset de uso final, se emplea con un Modelo de Regresión Lineal y otro de Redes Neuronales para proceder a comparar la precisión y eficiencia de cada uno. Finalmente se emplea un modelo de XAI (eXplicable Artificial Intelligence) que ayude a comprender la razón de los resultados obtenidos y estos serán explicados y detallados.

2. Análisis del dataset:

Durante este apartado, analizaremos los campos con los que cuentan los datasets empleados y trataremos las diferentes transformaciones que se darán sobre los mismos así como las columnas empleadas para el entreno del modelo explicando las causas que me han llevado a tomar las decisiones finales.

Para ello lo primero es ver que el dataset empleado se encuentra en Kaggle, siendo este perteneciente al siguiente conjunto. De aquí procederemos a extraer los datasets necesarios para nuestro proyecto: fish.csv y insects.csv.

Dataset	fish.csv	insects.csv
Atributos (columnas)	# Name Sell Where/How	# Name Sell Where/How

	Shadow	Weather
	Total Catches to Unlock	Total Catches to Unlock
	Spawn Rates	Spawn Rates
	Rain/Snow Catch Up	NH Jan
	NH Jan	NH Feb
	NH Feb	NH Mar
	NH Mar	NH Apr
	NH Apr	NH May
	NH May	NH Jun
	NH Jun	NH Jul
	NH Jul	NH Aug
	NH Aug	NH Sep
	NH Sep	NH Oct
	NH Oct	NH Nov
	NH Nov	NH Dec
	NH Dec	SH Jan
	SH Jan	SH Feb
	SH Feb	SH Mar
	SH Mar	SH Apr
	SH Apr	SH May
	SH May	SH Jun
	SH Jun	SH Jul
	SH Jul	SH Aug
	SH Aug	SH Sep
	SH Sep	SH Oct
	SH Oct	SH Nov
	SH Nov	SH Dec
	SH Dec	Color 1
	Color 1	Color 2
	Color 2	Icon Filename
	Size	Critterpedia Filename
	Lighting Type	Furniture Filename
	Icon Filename	Internal ID

	Critterpedia Filename	Unique Entry ID
	Furniture Filename	
	Internal ID	
	Unique Entry ID	

Tabla 2.1: Atributos vanilla por Set

2.1. Filtración de Atributos

Analizando en detalle los atributos de cada uno de los sets (Tabla 2.1), podemos encontrar similitudes en ciertos atributos, atributos idénticos y atributos únicos. Por ello lo primero que voy a hacer es remover información redundante o irrelevante del dataset. Haciendo esto nos encontramos con lo siguiente:

- Los meses de aparición en el hemisferio Sur (SH) es irrelevante en este contexto al contar con el tiempo en los meses de aparición en el hemisferio Norte (NH).
- Toda la información ligada a los IDs y nombres propietarios (Unique Entry ID, Internal ID, Critterpedia File Name, Furniture Filename, Icon FileName, Name) nos será irrelevante ya que en un principio no contrastaremos esta información a la hora de trabajar con el modelo.

Una vez eliminada la información irrelevante, es hora de tratar los elementos que no tienen en común los datasets, para eliminarlos o adaptarlos con el fin de tener un único modelo versátil. Los atributos diferentes que podemos ver son los siguientes:

- fish.csv: Shadow - Define el tamaño de la sombra que se ve antes de pescar el pez.
- fish.csv: Rain Snow Catch Up - Define en valor booleano (Yes, No) si se necesita que nieve o llueva para que aparezca el pez.
- fish.csv: Size - Define el tamaño final del pez una vez pescado.

- insects.csv: Weather - Define el tiempo que debe de hacer para que aparezca el insecto.
- fish.csv: Lighting Type - Define la luminosidad del pez una vez pescado

La manera con la procederemos en base a estas diferencias será la siguiente. Eliminaremos los campos de “**Size**”, “**Shadow**” y “**Lighting type**” del dataset de “*fish.csv*” ya que no existe forma de encontrar un igual en el otro dataset. Por otra parte, adaptamos el formato de “**Rain/Snow catch up**”, de manera que se asemeje a aquel de weather en “*insects.csv*”.

Dataset	fish.csv	insects.csv
Atributos (columnas)	Sell Where/How Weather (Rain/Snow adap) Total Catches to Unlock Spawn Rates NH Jan NH Feb NH Mar NH Apr NH May NH Jun NH Jul NH Aug NH Sep NH Oct NH Nov NH Dec Color 1 Color 2	Sell Where/How Weather Total Catches to Unlock Spawn Rates NH Jan NH Feb NH Mar NH Apr NH May NH Jun NH Jul NH Aug NH Sep NH Oct NH Nov NH Dec Color 1 Color 2

Tabla 2.2: Atributos filtrados por Set

2.2. Atributos empleados

Los resultados de esta filtración de datos se pueden encontrar en la Tabla 2.2. Una vez filtrados los datos, pasaremos a analizarlos uno por uno argumentando que son y las razones para mantenerlos, así como una clasificación por tipo de variable (Tabla 3).

- Sell: Esta es la variable a predecir, por lo que es importante tenerla en cuenta en los modelos de entreno.
- Where/How: El lugar de aparición puede ser definitorio, sitios más específicos puede que incrementen el precio.
- Weather: Como de disponible sea el animal puede determinar el precio.
- Total Catches to unlock: Si un animal no aparece a no ser que ya hayas cazado otros, puede ser razón o no de un incremento en el precio.
- Spawn Rates: La tasa de aparición define la rareza del animal.
- NH *: Ciertos meses pueden ser más beneficiosos para un cazador/pescador y considero que un análisis sobre esto puede influenciar el cómo se juega al juego.
- Color *Es posible que ciertos colores o combinaciones denotan una rareza.

En cuanto al análisis de los tipos de variables (Tabla 2.3) podemos ver que predominan las variables Categóricas Nominales. Esto nos puede causar un problema ya que los modelos trabajan mejor con variables numéricas. En el siguiente apartado trataremos de transformar el máximo número de variables nominales a numéricas.

Variable	Tipo de Variable
Sell	Numérica Discreta

Where/How	Categórica Nominal
Weather	Categórica Nominal
Total Catches to unlock	Numérica Discreta
Spawn Rates	Numérica Discreta
NH *	Categórica Nominal Horario
Color *	Categórica Nominal

Tabla 2.3: Análisis de las variables elegidas

2.3. Transformación de Variables Nominales a Numéricas:

Tras revisar los apuntes de teoría y de prácticas anteriores, he tomado diferentes decisiones de cara a cada uno de los atributos nominales presentes en la tabla. Esto lo hacemos porque aunque vayamos a trabajar con una red neuronal capaz de procesar el texto, no vamos a trabajar exclusivamente con ella y queremos que ambas muestras trabajen con la misma información para poder hacer una comparativa válida.

De cara a trabajar con los horarios (NH *), se han barajado varias posibilidades, desde un sistema de transformación de horarios a array de manera que se vean en un formato como el siguiente: [0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,0]; hasta tratar de generar todas las características mediante One Hot Encoder, lo cual generaría 24 horas * 12 meses = 288 columnas. Finalmente me he decantado por usar un sistema bastante más generalista, el cual si bien puede fallar en precisión, considero que pueda cumplir su función. Este sistema consiste en la separación en estaciones (Primavera, Verano, Otoño, Invierno) y momentos del día (Amanecer, Mañana, Mediodía, Tarde, Anochecer, Medianoche) donde los límites de las franjas horarias se pueden observar en la Tabla 2.4. Esto funciona ya que tras analizar el dataset detalladamente, me he percatado de lo siguiente:

Si una criatura aparece en varios meses, aparecerá siempre en la misma franja horaria a través de todos sus meses de aparición.

Finalmente para la categorización estacional, usaré las estaciones naturales del año (Tabla 2.5). Para que una criatura figure dentro de esa estación, ha de aparecer en al menos 2 de los 3 meses que abarca cada una. Este último criterio será empleado también sobre las horas del día: Para que una criatura figure en un momento del día, ha de estar presente en al menos 2 de las 4 horas que éste abarca.

Horas que abarca (24h)	Momento del día
04-07	Amanecer
08-11	Mañana
12-15	Mediodía
16-19	Tarde
20-23	Anocheecer
00-03	Medianoche

Tabla 2.4: Relación horas y momento del día

Meses que abarca	Estación
Enero - Marzo	Invierno
Abril - Junio	Primavera
Julio - Septiembre	Verano
Octubre - Diciembre	Otoño

Tabla 2.5: Relación meses y estaciones

Tras un análisis de opciones para el campo **weather** dentro del dataset, podemos observar que solo existen 3 opciones: “Todas menos lluvia”, “Todas” y “Solo con lluvia”. Debido a la

naturaleza corta de esta característica, me he decantado por trabajar con un sistema como el de One Hot Encoder para este atributo.

Este mismo formato es el que podemos utilizar para los colores, no sin antes aplicar un filtro de tonalidad por cada uno de ellos ya que añadir 11 campos (elementos únicos en el dataset) prolongará mucho el número de columnas. Este filtro transforma los datos en tono cálido, neutro y frío. Una visión más detallada se puede observar en la tabla 2.6, donde vemos que se transforman los datos de manera que se mantenga un número similar en cada categoría (4 o 5 por cada una).

Tonalidad	Valores únicos de la clase Color
Frío	Blue Light Blue Purple Green
Neutro	Black Beige Gray White
Cálido	Brown Orange Red Pink Yellow

Tabla 2.6: Opciones únicas del atributo **color**

Finalmente nos queda observar y analizar que vamos a hacer con el atributo ***Where/How***, el cual en su estado raw cuenta con los elementos vistos en la tabla 2.7, alcanzando un total de 32 elementos. Estos comparten características las cuales nos permitirá realizar una reducción de las mismas con el objetivo de minimizar la complejidad que pueda generar un número de

existencias tan grande (bien sea por sparsity al abusar de 0s si usamos one hot encoder o prioridades en caso de usar label encoder).

DataSet origina al que pertenece	Valores únicos para <i>Where/How</i>
<i>Where/How</i> en peces	Sea River Pier Pond River (clifftop) Sea (rainy days) River (mouth)
<i>Where/How</i> en insectos	Flying near flowers On rotten turnips On trees (any kind) Shaking trees (hardwood or cedar only) Flying near water On the ground On palm trees On hardwood/cedar trees From hitting rocks On tree stumps Flying On rivers/ponds Pushing snowballs On villagers Flying near trash (boots, tires, cans) or rotten turnips Disguised on shoreline On flowers Underground (dig where noise is loudest) Flying near light sources On white flowers

	Flying near blue/purple/black flowers On rocks/bushes Disguised under trees Shaking trees On beach rocks
--	--

Tabla 2.7: Posibilidades de **Where/How** según tipo de criatura

En cuanto a la reducción, esta se hará en las categorías vistas en la tabla 2.8. Es en esta misma tabla que se especifica que valores recoge cada categoría reducida, quedándonos un total de 10 categorías. La manera que tendremos de proceder será emplear un Label Encoder sobre estos elementos. La razón por la que escogemos este sistema y no One Hot Encoder es porque este último propone una alternativa que puede estar cargada de sparsity al contar con 9 0s asegurados contra un único 1.

Por otra parte, la pega que podía tener el uso de Label Encoder (prioridades), puede emplearse a nuestro favor si ordenamos las categorías de manera que aquellas categorías con menos elementos (y por ello más raras) reciban un orden de prioridad superior contra aquellas que recogen más casos

Categoría	Elementos en categoría	Código Label Encoder
On villagers	On villagers	1
River (clifftop)	River (clifftop)	2
On beach rocks	On beach rocks	3
River (mouth)	River (mouth)	4
Aguas Interiores (Inland Water)	River Pond On rivers/ponds	5
Aguas Exteriores (Outland Water)	Sea Sea (rainy days)	6

	Pier	
Flora	On rotten turnips On flowers On white flowers On rocks/bushes	7
Tierra (Ground)	On the ground From hitting rocks Disguised on shoreline Underground (dig where noise is loudest) Pushing snowballs	8
Volando (Flying)	Flying Flying near water Flying near flowers Flying near light sources Flying near blue/purple/black flowers Flying near trash (boots, tires, cans) or rotten turnips	9
Árboles (trees)	On trees (any kind) On palm trees On hardwood/cedar trees On tree stumps Shaking trees (hardwood or cedar only) Disguised under trees Shaking trees	10

Tabla 2.8: Agrupamiento y numeración del atributo **Where/How** para label encoder

2.4. Transformación de Spawn Rates:

Finalmente, la variable Spawn Rates va a tener que ser transformada al contar con elementos variables, por ejemplo 7-9. En estos casos donde se proporciona un rango, cambiaremos el valor por la media de los valores proporcionados, para el ejemplo anterior, esa fila tomará valor 8.

3. Empleo de Modelos de IA:

Durante esta práctica vamos a estar trabajando con un modelo de Aprendizaje Automático Clásico y otro modelo de Aprendizaje Automático Moderno. Para saber que representante de estas tecnologías vamos a emplear finalmente, haremos un análisis de un conjunto de cada grupo y concluimos especificando los modelos usados, la razón para ello y como se emplearán en nuestro caso específico.

3.1. Modelos de Aprendizaje Automático Clásico

Los modelos de aprendizaje automático clásico recogen modelos cuya interpretabilidad es sencilla y su entrenamiento es rápido al ser algoritmos “simples”. Esta simpleza conlleva que su efectividad vaya decayendo conforme incrementa el número de parámetros y datos que se emplean para su entreno, por eso se suele emplear en modelos pequeños o medianos. Otra de las cosas a tener en cuenta sobre estos modelos es que solo pueden operar sobre tablas, estadísticas y otras formas estructuradas de datos **numéricos**.

Durante este proyecto me centraré en comparar los 3 modelos de regresión principales vistos en clase: Lineal, Polinómica y Logística.

3.1.1. Modelo de Regresión Lineal

Los modelos de regresión lineal destacan por su uso en predicción de variables continuas, como pueden ser temperaturas, edades o precios, especialmente en casos donde no existen atributos con dependencias profundas entre ellos y donde la simpleza prevalezca. Este modelo no se adapta muy bien a casos donde haya que hacer correcciones pequeñas y por ello

sus predicciones tienden a ser aproximaciones generales, no coincidencias exactas con los datos de entrenamiento.

En cuanto a su uso para nuestro problema, podría ser apreciado al estar buscando una categorización continua del precio de la criatura en base a sus cualidades. Pese a esto, no tenemos ninguna información de la existencia de una relación entre los diferentes atributos de la tabla, lo que podría condicionar su uso.

3.1.2. Modelo de Regresión Polinómica

Los modelos de regresión polinomial operan de una manera similar a los lineales, y aunque la precisión en modelos de mucha fluctuación es mayor a la encontrada en el modelo descrito anteriormente, este modelo es más susceptible a Overfitting y Underfitting, especialmente en grados de potencia altos y bajos respectivamente.

De cara al uso en nuestro problema, llama la atención de nuevo la predicción de variable continua que propone este modelo, y en el supuesto caso de contar con relación entre atributos, podría ser más interesante que su símil lineal. De todas formas el alto número de columnas con las que contamos (20) condiciona la máxima exponenciación a grado 2 o 3, limitando la precisión en casos específicos.

3.1.3. Modelo de Regresión Logística

Los modelos de regresión logística destacan en categorización de atributos en valores binarios (positivo o negativo) en base a un umbral sobre una función sigmoide, contrastando con lo propuesto por los dos modelos anteriores.

En cuanto a su uso en base a nuestro problema, esta cualidad binaria del mismo no es lo que estaríamos buscando, puesto que buscamos predecir el precio específico y no la respuesta a una pregunta como “¿Es cara o barata esta criatura?”.

3.1.4. Conclusión:

A falta de mayor información sobre el dataset, podemos ver que tanto la opción de Modelo de Regresión Lineal como la de Modelo de Regresión Polinomial, podrían encajar para la solución de nuestro problema.

La solución para resolver esta disyuntiva sobre el modelo a emplear deberá resolverse comparando los resultados empíricos de ambos modelos tras su implementación y comparativa.

3.2. Modelos de Aprendizaje Automático Moderno

Los modelos de aprendizaje automático modernos recogen modelos de baja interpretabilidad al estar conformados por algoritmos de caja negra, basándose en correcciones e iteraciones. Su entrenamiento es lento pero esta lentitud conlleva una corrección precisa para cada situación específica. De todas formas, hay que ser cuidadosos con la búsqueda de la precisión absoluta ya que estos modelos son muy susceptibles al overfitting si se da la situación descrita anteriormente. Una de las ventajas de estos modelos es su versatilidad a la hora de operar con datos que no sean estructurados, como imágenes, texto, audio y video.

3.2.1. Red Neuronal Feed Forward (FNN):

La red neuronal *Feedforward* es un tipo de Perceptrón Multicapa que emplea técnicas de retroalimentación en ambos sentidos, *backpropagation*, lo habitual en perceptrones multicapa y *forward propagation*, lo nuevo que aporta.

Su característica doble corrección junto con su simplicidad puede garantizar unos buenos resultados en nuestro problema con un coste computacional mínimo entendiendo que se encuentra en el campo de las redes neuronales. De todas formas y debido al número de características, podría darse el caso de que el límite superior de precisión no sea óptimo ya que su sencillez junto con un número alto de atributos es más propenso a provocar overfitting.

3.2.2. Red Neuronal Convolutiva (CNN):

Las redes neuronales convolucionales son otro tipo de Perceptrones Multicapa, esta vez con más de una hidden layers (categorizada entonces como modelo de *Deep Learning*) que se especializa en el procesamiento de imágenes.

Es esta última característica la que hace que esta técnica esté lejos de ser ideal para nuestro problema.

3.2.3. Red Neuronal Long-Short Term Memory (LSTM):

Las redes neuronales LSTM, son una especificación del modelo RNN (Recurrent Neural Network), que a su vez podrían ser considerados una evolución de los perceptrones multicapa al contar esta vez con la propiedad de “recordar” estados anteriores, lo que se conoce como memoria, así como evitar problemas de su predecesor como puede ser el desvanecimiento de gradiente. Esto aunque mejora la precisión, garantiza un coste computacional mayor.

Al no contar con restricciones más allá de las ligadas al hardware de entrenamiento, esta pega final no sería algo que afectaría en gran medida, sobre todo de poder contar con una precisión mayor y una sensibilidad al overfitting menor al formar parte del grupo de los modelos de *Deep Learning*.

3.2.4. Transformers:

Los transformadores o *transformers* son, sin duda, el modelo más complejo de los estudiados aquí. Este está conformado por varios elementos, entre los cuales podemos mencionar una capa de *FNN* y otra de *Self-Attention*. Esto provoca que su precisión sea muy buena, con un entrenamiento adecuado, pero empleando un gran coste computacional.

De cara a nuestro problema, el empleo de este modelo podría ser excesivo, aunque podría ser interesante ver sus resultados, ya que su capacidad para manejar relaciones a largo plazo podría ofrecer ventajas dependiendo de la naturaleza de los datos.

3.2.5. Funciones de Activación:

Con el fin de romper la linealidad, y favorecer el aprendizaje de algoritmos complejos, estos modelos de IA van a estar trabajando con funciones de activación. Las opciones que se han barajado son las que se pueden apreciar en la tabla 3.2.5.1. En dicha tabla podemos observar como trabajan cada una de ellas, comenzando por *ReLU*, la cual transforma los números negativos a 0, y poco a poco llegando a extremos de radicalización más rápida, donde la función Tangente Hiperbólica es la que más rápido varía de extremo de números negativos a extremos de números positivos.

Mi objetivo a la hora de realizar este estudio es el de comprobar cuánto favorece la existencia de números negativos a la toma de decisiones final. Por ello considero que las funciones que más me permitirán ver el cuánto podrían afectar en los negativos sin que ello conlleve un cambio en los positivos, son la *ReLU* y la *LeakyReLU*.

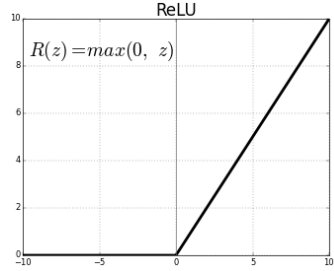
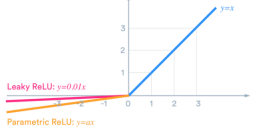
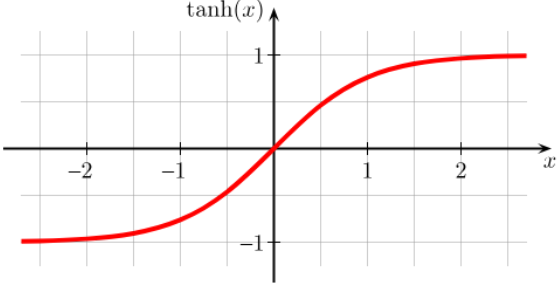
Función de activación	Gráfica
ReLU	 <p>ReLU</p> $R(z) = \max(0, z)$
LeakyReLU	$f(x) = 0,01x \text{ si } x < 0$ $f(x) = x \text{ si } x \geq 0$  <p>Leaky ReLU: $y=0,01x$ Parametric ReLU: $y=x$</p>
Tangente Hiperbólica (tanh)	 <p>$\tanh(x)$</p>

Tabla 3.2.5.1. Comparativa de funciones de activación

3.2.6. Conclusión:

Tras haber estudiado alguno de los diferentes modelos que podríamos usar para nuestro problema, considero que tanto las opciones de *LSTM* y *FNN* son las opciones que tienen mejor cabida en este problema.

De nuevo y similar a lo realizado con los Modelos de Aprendizaje Automático Clásicos, la decisión de cuál es mejor la haremos comparando y analizando los resultados empíricos

obtenidos con ambos, basándonos en precisión con respecto al coste computacional que suponen.

3.3. Implementación de Modelos de Aprendizaje Clásico:

Durante este apartado nos centraremos en analizar cómo se desarrolló la implementación de los algoritmos, las pruebas que se han hecho y las conclusiones que podemos derivar de las mismas con el fin de completar la disyuntiva generada en el análisis teórico.

3.3.1. Clase de Modelos de Regresión:

Con el fin de mantener los análisis de la manera más similar posible y evitando repeticiones innecesarias del código de manera que el mismo se presente de la manera más limpia posible, comencé declarando una clase abstracta ***RegressionModels***. En esta clase se implementa toda la lógica básica, a excepción del entrenamiento de los modelos y el mostrado de gráficos, estas son todas las funciones relacionadas con preparar los datos, normalizarlos, verificarlos y hacer las pruebas de precisión.

Esta clase cuenta con únicamente dos métodos públicos, un método **compute()** y otro método **show_plot()** y mientras que el segundo es un método abstracto dependiente del modelo implementado, el primer método es general para todas las diferentes implementaciones. (Véase la imagen 3.3.1)

```
# Método público que realiza todos los cálculos
def compute(self, data:pd.DataFrame) → float:
    self._verify_data(data)
    self._prepare_data()
    self._regression()
    return self._get_precision()

# Método público y abstracto encargado de hacer el plot del último dataframe o si no de uno nuevo
@abstractmethod
def show_plot(self, data:pd.DataFrame | None = None):
    pass
```

Imagen 3.3.1. Métodos públicos de la clase ***RegressionModels***

3.3.2. Clase Modelo de Regresión Lineal

Como hemos visto antes, esta clase, definida en código como ***LinearRegression***, es una implementación de la clase abstracta ***RegressionModels***. En ella se detalla la implementación

del método `_regression()` el cual gestiona el entrenamiento específico de los datos, y el método `show_plot()`, haciendo únicos los gráficos de esta clase.

De cara a la implementación del primer método, nos limitaremos a un caso básico de entreno para el módulo `sklearn.linear_models`, haciendo uso de su clase `LinearRegressor` como se puede apreciar en la imagen 3.3.2. En esta dónde definimos el regresor lineal, lo entrenamos en base a los datos de entrenamiento y actualizamos el modelo de la clase para que sea el regresor entrenado.

Por otra parte y en cuanto a la representación de gráficos, haremos uso de la librería `matplotlib` y su módulo `pyplot`. Con ella haremos un *scatterplot* de los resultados reales contra los predichos por el modelo y mostraremos la línea de regresión de la misma a través de un plot. Un ejemplo de resultado puede verse en la imagen 3.3.3.

```
# Implementación del método de regresión
def _regression(self) → None:
    # Creamos el regresador
    lin_reg = lm.LinearRegression()
    # Lo entrenamos
    lin_reg.fit(self._train_data, self._train_results)
    # Lo establecemos como modelo de la clase
    self._model = lin_reg
```

Imagen 3.3.2. Implementación del método regresión lineal

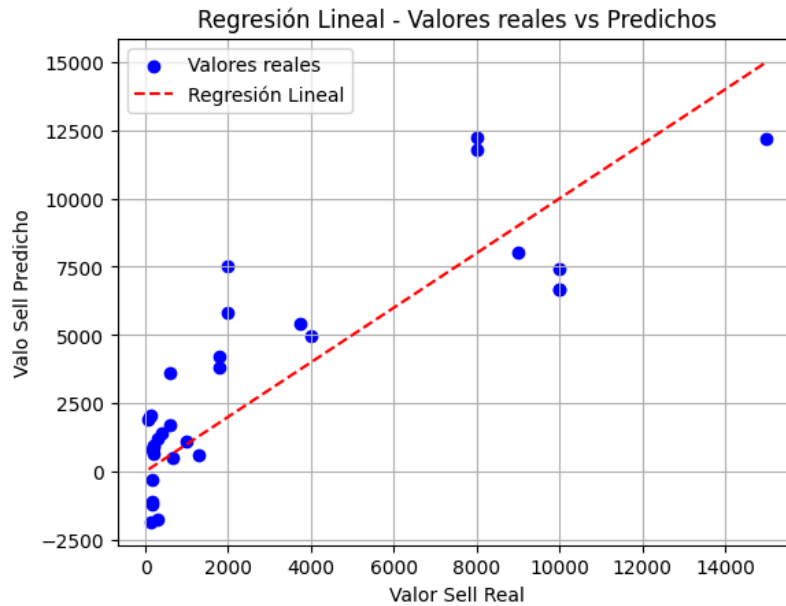


Imagen 3.3.3. Ejemplo de gráfico de regresión lineal

3.3.1 Clase de Modelo de Regresión Polinomial

Siguiendo la tendencia de la regresión lineal, esta clase, definida en código como ***PolynomialRegression***, es una implementación de la clase abstracta ***RegressionModels***. En ella se detalla la implementación del método **`_regression()`** el cual gestiona el entrenamiento específico de los datos, y el método **`show_plot()`**, haciendo únicos los gráficos de esta clase.

De cara a la implementación del primer método, traté de hacer pruebas con el método por defecto que viene en el módulo **`sklearn.linear_models`**. Este se haría con una transformación de los datos con el método **`sklearn.preprocessing.PolyFit`** y luego un entrenamiento con **`LinearRegressor`**. Tras implementarlo, los resultados que estaba obteniendo de precisión eran negativos, indicando que una aproximación mediante la media alcanzaría una precisión superior.

Tras una investigación en el tema, descubrí otro modelo que empleaba el concepto matemático de [mínimos cuadrados](#) para obtener una precisión superior. Este empleaba el módulo **`sklearn.preprocessing`** para tratar los datos de entrenamiento y pruebas, de manera que usando el modelo **`Ridge()`** del módulo **`sklearn.linear_models`**, pudiéramos obtener resultados naturales para polinomios de distintos grados. La implementación puede verse en la imagen 3.3.4.

Finalmente en cuanto a la muestra de los resultados mediante gráficos, nos limitaremos a hacer un *scatterplot* de los resultados reales contra los predichos sin profundizar en el dibujo de la línea de tendencia ya que este nuevo método no proporciona una fórmula general que permita su dibujo de una manera sencilla. Un ejemplo de resultado puede verse en la imagen 3.3.5.

```
# Implementación del método de regresión
def _regression(self)→ None:
    # Definimos un transformador para operar con polinomios
    self._poly_features = preprocessing.PolynomialFeatures(self._degree)

    # Transformar los datos
    self._train_data = self._poly_features.fit_transform(self._train_data)
    self._test_data = self._poly_features.transform(self._test_data)

    # Usar regresión ridge para evitar overfitting (Basada en mínimos cuadrados)
    pol_reg = lm.Ridge(alpha=1.0)
    # Entrenamos el modelo
    pol_reg.fit(self._train_data, self._train_results)

    # Definimos el modelo de la clase para predicciones
    self._model = pol_reg
```

Imagen 3.3.4. Implementación del método de regresión polinomial

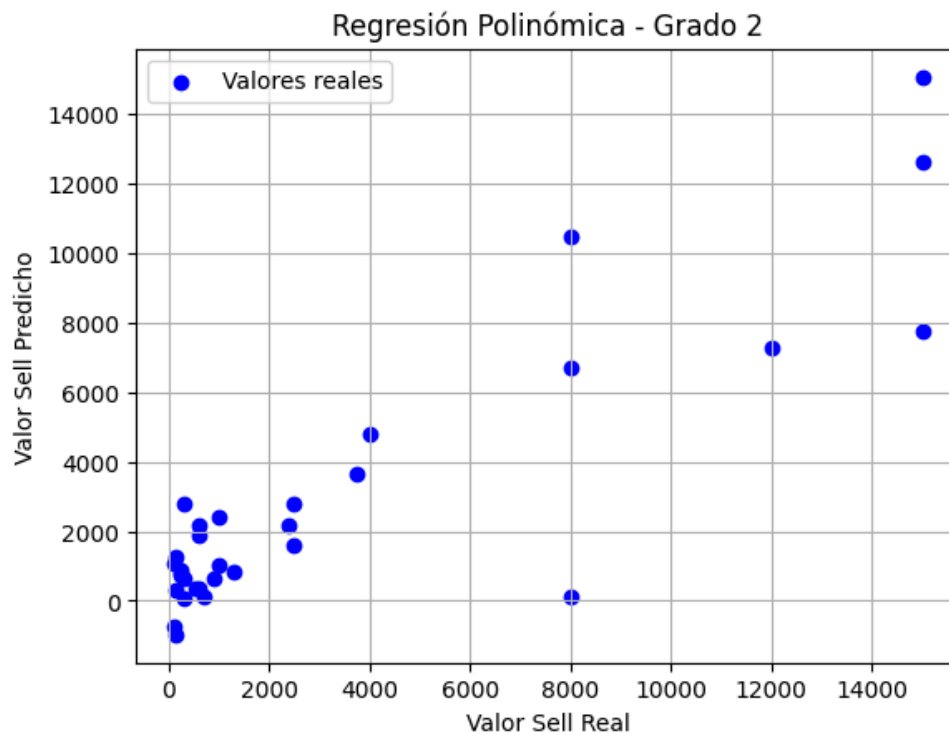


Imagen 3.3.5. Ejemplo gráfico de regresión polinomial

3.3.4. Análisis y Conclusión

Finalmente, con las clases definidas, me dispuse a realizar pruebas sobre un único modelo de regresión lineal y sobre tres modelos diferentes de regresión polinomial, cambiando el grado a 2, 5 y 7, con el fin de corroborar si, como dije anteriormente en la teoría, el modelo iba decayendo al subir el grado, o si para mi sorpresa se iba afinando.

Inicialmente realicé una única prueba por cada uno de los modelos, mostrando sus datos de porcentaje de aciertos, error cuadrático medio y error absoluto medio (imagen 3.3.6), sin embargo noté que los resultados varían de manera que derivar alguna conclusión solo con la información mostrada no serían muestras representativas. Con el fin de tratar de minimizar este componente aparentemente aleatorio, desarrollé un script que ejecutará cada prueba 100 veces y sacaría la media de precisión y el error medio absoluto de las muestras así como su tiempo tomado, de manera que los resultados representan mejor el modelo correspondiente. Podemos apreciar los resultados en la imagen 3.3.7.

```
=====
Precision: 72.92%
Error cuadrático medio: 4584255.64
Error absoluto medio: 1623.01
```

Imagen 3.3.6. Ejemplo de salida de entrenamiento de muestra

```
===== Regresión Lineal =====
Precisión media: 72.92%
Error absoluto medio: 1623.01u
Tiempo tomado: 0.62s

===== Regresión Polinomial (2) =====
Precisión media: 81.46%
Error absoluto medio: 1408.91u
Tiempo tomado: 1.84s

===== Regresión Polinomial (5) =====
Precisión media: 52.45%
Error absoluto medio: 1971.50u
Tiempo tomado: 11.83s

===== Regresión Polinomial (7) =====
Precisión media: 49.24%
Error absoluto medio: 1986.00u
Tiempo tomado: 72.98s
```

Imagen 3.3.7. Resultados de los modelos tras 100 iteraciones

De estos resultados podemos concluir que si bien la regresión polinomial obtiene mejores resultados que la lineal, esto sólo ocurriría con ciertas configuraciones específicas e incrementando el tiempo de entreno para márgenes de precisión de entorno al 1.5~10% dependiendo de la semilla o *random_state*.

Por otra parte, podemos ver que la teoría está en lo cierto y que, en general, a mayor grado usado en la regresión polinomial, menor es la precisión del resultado y sobre todo menor es la precisión entre los segundos tomados para el entrenamiento.

Con toda esta información, considero que para el caso específico de mis datos de entrenamiento, el modelo de regresión polinomial con grados bajos, en concreto 2, sería el modelo más preciso. Esta consideración se basa en que si bien es verdad lo que hemos dicho de que incrementa el tiempo que le tomaría a una regresión lineal por un incremento de precisión muy sutil en algunos casos, las cifras temporales en las que nos movemos son de un orden de magnitud muy bajo (apenas alcanzando el segundos realizando 100 entrenamientos y pruebas diferentes), por lo que priorizaremos maximizar la precisión sobre el tiempo.

3.4. Implementación de Modelos de Aprendizaje Moderno:

A continuación pasaremos a explicar la implementación de los modelos de I.A. modernos implementados. Antes de comenzar, mencionar que el escalador de normalización que vamos a estar empleando, *RobustScaler*, no funciona de la manera habitual ya que opera sobre el rango entre cuartiles (IQR). Esta forma de escalado no garantiza que todos los números sean positivos, por lo que *LeakyReLU* contaría con ventaja con respecto a *ReLU*, pero me he decantado por implementar esta forma ya que escalando con *MinMaxScaler*, sólo conseguía obtener precisiones de hasta el 62%.

Por otra parte las capas ocultas (*hidden_layers*) y la tasa de aprendizaje (*learning_rate*), son variables fijas e iguales para ambas clases, las cuales he ajustado varias veces para obtener los coeficientes que maximicen, dentro de las pruebas hechas, los resultados.

3.4.1. Implementación de FNN

De cara a esta primera implementación de los modelos de IA modernos, he decidido crear una clase la cual permitirá ejecutar el modelo con varias configuraciones de una manera sencilla. Para ello comenzaremos con un constructor donde definiremos la potencia del dropout (0 si queremos ninguno y 1 si queremos que todos), el cual por defecto está a 0.1. En este mismo constructor también definiremos la función de activación a través de un valor booleano **using_relu**, el cual en caso de ser verdadero usará *ReLU* y en su defecto *LeakyReLU*.

Seguidamente pasaremos a normalizar los datos, inicializar los pesos en base a la función de activación, usando *Kaiming Initization* y a definir las capas del modelo, las cuales toman la estructura que se puede apreciar en la imagen 3.4.1, dónde la capa de activación tomará el valor respectivo en función de la decisión tomada y la reducción del *dropout* se dará solo con valores **d > 0**. En este mismo esquema podemos apreciar que el *dropout* difiere en sus dos llamadas ya que la tasa de reducción de la segunda es menor. Esto decidí hacerlo así ya que, por una parte quería mantener el dropout en esas capas finales, pero eliminar demasiado podría afectar negativamente al modelo.

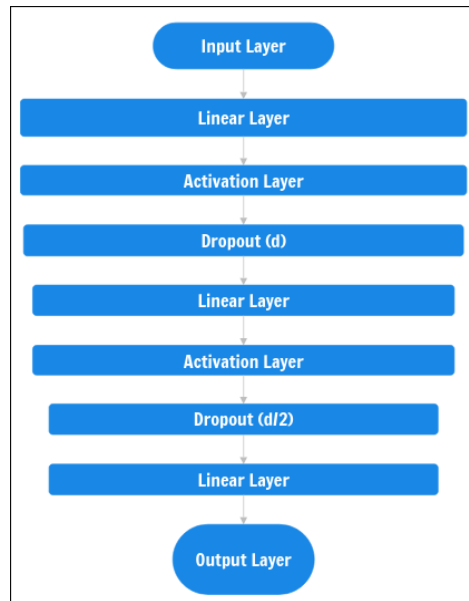


Imagen 3.4.1. Estructura FNN

Establecidas las capas, pasamos a implementar el modelo, y para ello la clase que hemos definido, hereda directamente de `torch.nn.Module`. Esta herencia nos permite realizar entrenos y evaluaciones de manera más sencilla, teniendo llamadas automáticas a métodos de corrección, como es en el caso del FNN el método *forward*. La función de entrenamiento, sigue una lógica sencilla: Por cada *epoch*, realizará una optimización, una pasada hacia delante con corrección (método *forward*), cálculo de pérdidas, y corrección en base a las pérdidas (método *backward* sobre las mismas). Por otra parte, la función de evaluación, emplea una llamada al método *forward* sobre la muestra de testeo, de manera que el resultado obtenido sea la predicción y en base a ella calcula la precisión.

```

Progreso general: 100%|██████████| 250/250 [15:40<00:00, 3.76s/iter]

===== RESULTADOS FINALES (250 iteraciones) =====

1. LeakyReLU, con Dropout: Precisión media 86.83% (Veces primero: 129, Veces último: 0)
2. ReLU, con Dropout: Precisión media 86.15% (Veces primero: 120, Veces último: 0)
3. ReLU, sin Dropout: Precisión media 81.02% (Veces primero: 0, Veces último: 124)
4. LeakyReLU, sin Dropout: Precisión media 80.52% (Veces primero: 1, Veces último: 126)
  
```

Imagen 3.4.2. Resultados de FNN

Finalmente, tras una evaluación de 250 iteraciones con 500 *epoch* cada una sobre todos los modelos, podemos obtener los resultados que se ven en la imagen 3.4.2. Ahí podemos apreciar que la función de activación con valores negativos *LeakyReLU*, obtiene la mejor

precisión y consistencia de todos, seguido muy de cerca pero detrás en toda comparativa por su contraparte. Considero que esto se debe en gran parte a la alta presencia de valores negativos que se obtienen tras la normalización, al haber hecho uso de una función de normalización que normaliza en base al IQR (***RobustScaler***) y no garantiza que todos los valores sean positivos. Por otra parte, el uso de *dropout* es indispensable para obtener resultados precisos, ya que mejora el rendimiento con ambas funciones de activación pero destaca en el caso de *ReLU* dónde al no contar con esta función ni la normalización de valores negativos .

Finalmente me gustaría destacar el tiempo por iteración de 3.76 segundos, el cual será una de las medidas a comparar con el LSTM.

3.4.2. Implementación de LSTM

Para la implementación de este modelo he decidido crear una clase la cual permitirá ejecutar el modelo con varias configuraciones de una manera sencilla. Para ello comenzaremos con un constructor donde definiremos la potencia del dropout (0 si queremos ninguno y 1 si queremos que todos), el cual por defecto está a 0.1. En este mismo constructor también definiremos la función de activación a través de un valor booleano **using_relu**, el cual en caso de ser verdadero usará *ReLU* y en su defecto *LeakyReLU*.

Seguidamente pasaremos a normalizar los datos y a definir las capas del modelo, las cuales toman la estructura que se puede apreciar en la imagen 3.4.3, dónde la capa de activación tomará el valor respectivo en función de la decisión tomada y la reducción del *dropout* se dará solo con valores $d > 0$. En este caso el paso de inicializar los pesos, los hace por defecto el LSTM sin nosotros tener que definirlo, a diferencia del modelo anterior.

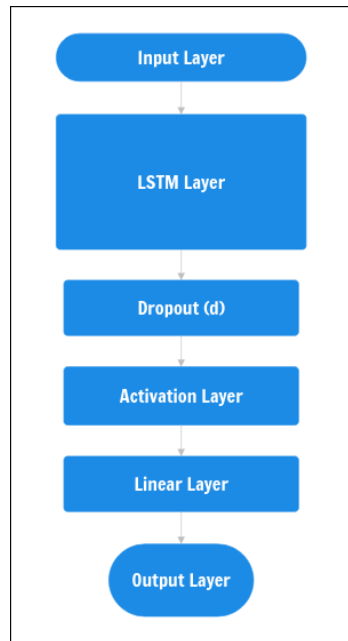


Imagen 3.4.3. Estructura de LSTM

Establecidas las capas, pasamos a implementar el modelo, y para ello la clase que hemos definido, hereda directamente de [torch.nn.Module](#). Esta herencia nos permite realizar entrenamientos y evaluaciones de manera más sencilla, teniendo llamadas automáticas a métodos de corrección. La función de entrenamiento, sigue una lógica sencilla: Por cada *epoch*, realizará una optimización, una pasada hacia delante con corrección (método **forward**), cálculo de pérdidas, y corrección en base a las pérdidas (método **backward** sobre las mismas). Por otra parte, la función de evaluación, emplea una llamada al método **forward** sobre la muestra de testeo, de manera que el resultado obtenido sea la predicción y en base a ella calcula la precisión. Muy similar a lo visto con el FNN pero cambiando la estructura de las capas y añadiendo memorización en aquella ligada al LSTM.

```
Progreso general: 100%|██████████| 250/250 [44:12<00:00, 10.61s/iter]

===== RESULTADOS FINALES (250 iteraciones) =====

1. LeakyReLU, sin Dropout: Precisión media 85.58% (Veces primero: 102, Veces último: 34)
2. LeakyReLU, con Dropout: Precisión media 84.32% (Veces primero: 51, Veces último: 35)
3. ReLU, con Dropout: Precisión media 84.06% (Veces primero: 19, Veces último: 121)
4. ReLU, sin Dropout: Precisión media 83.82% (Veces primero: 78, Veces último: 60)
```

Imagen 3.4.4. Resultados de LSTM

En cuanto a los resultados, esta vez para mi sorpresa, podemos apreciar que las versiones sin *dropout* dominan ante el resto, con una variación

3.4.3. Conclusiones

Si hacemos una comparativa de métricas entre ambos modelos (tabla 3.4.1), podemos observar que el FNN supera al LSTM en todas las métricas comparadas, y si bien la precisión más baja la ostenta una variante del FNN, la métrica más alta es superada por más de un 1% con un margen de mejora temporal de 2.82 veces. Por todo esto, considero que, para este caso específico, el FNN es superior al LSTM. De todas formas, con fines educativos, estudiaré la mejor muestra de cada campo mediante XAI.

Modelo	Precisión Media	Tiempo
FNN - LeakyReLU, Dropout	86.83%	3.76s/iter
FNN - ReLU, Dropout	86.15%	
FNN - LeakyReLU	81.02%	
FNN - ReLU	80.52%	
LSTM - LeakyReLU	85.58%	10.61s/iter
LSTM - LeakyReLU, Dropout	84.32%	
LSTM - ReLU, Dropout	84.06%	
LSTM - ReLU	83.82%	

Tabla 3.4.1. Comparativa de resultados: Mejor FNN, mejor LSTM, mejor y peor precisión y mejor y peor tiempo

De todas formas estas son las medidas de esta ejecución general, y aunque seguramente tengamos resultados diferentes, especialmente cambiando la muestra de test y entreno, esto recogería una imagen general. Estos resultados reflejan cómo la teoría de *deep learning* se manifiesta empíricamente: no hay una configuración universal, pero entender cómo opera cada sistema puede ayudar a elegir la mejor combinación.

Atendiendo a la teoría, si es verdad que por su funcionamiento, es más eficiente y más veloz el FNN que el LSTM, aunque ambos deberían recoger una precisión similar. De todas

formas, en el caso de trabajar con modelos con más entradas, la facultad de memorización del LSTM podría ponerle en cabeza con respecto al FNN.

4. Empleo de Modelos de XAI:

Para este último apartado, estaré trabajando con dos modelos de XAI con los que buscaré entender el funcionamiento de los modelos con los que hemos trabajado anteriormente. En específico analizaremos el *FNN - LeakyReLU con dropout* y el *LSTM - LeakyReLU con dropout* de los modelos de I.A. moderna. Esta elección se llevó a cabo no solo basándose en las precisiones de cada uno, si no que también en el porcentaje de veces que fueron el mejor modelo en su entreno.

Por otra parte, también considero que un estudio sobre los modelos de regresión, en particular la regresión lineal y la polinomial de grado 2 podría ser interesante, aunque considero que es más fácil entenderlas en caso de hacer un estudio meticuloso sobre las mismas.

Los modelos de XAI a implementar serán los siguientes: ALE para muestras generales y SHAP para muestras más específicas.

4.1. Implementación de ALE

Para la implementación de este primer modelo de IA explicable, con el cual estudiaremos el marco general de los resultados obtenidos, emplearemos un modelo simple de esta tecnología. Con ello conseguiremos garantizar que los 4 modelos que se vayan a emplear sigan una estructura similar: Se define y entrena el modelo, se crea un predictor que permita al ALE trabajar con *Dataframes*, se aíslan los nombres de las columnas, se define el ALE con el *predictor* y los nombres de las columnas como *feature_names*, y se pide una explicación.

Con el objetivo de implementar lo anterior descrito, comencé creando un método simple el cual recibe un modelo de IA y un *predictor*, ambos definidos y en el caso del modelo también entrenado, de manera que sintetice este proceso en un único lugar. Luego continué definiendo los predictores adaptándolos a cada uno de los modelos y finalmente llamé a la función. Los resultados obtenidos pueden apreciarse en las imágenes 4.1 - 4.4.

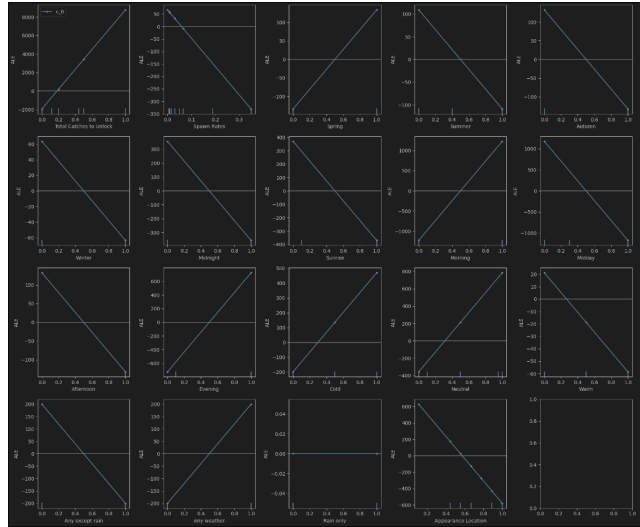


Imagen 4.1. Resultados de ALE de Regresión Lineal

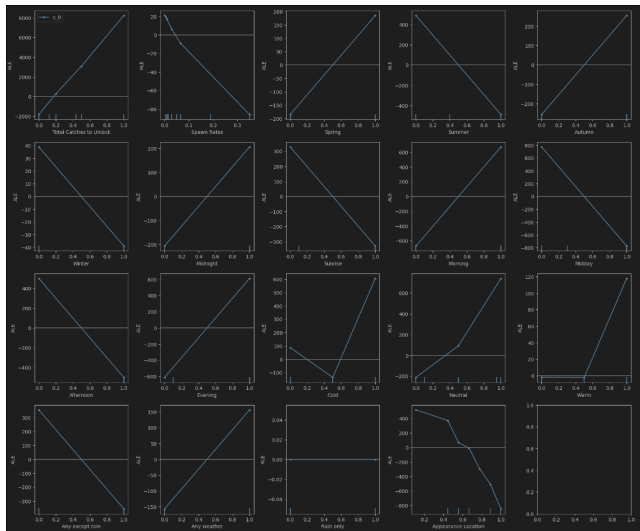


Imagen 4.2. Resultados de ALE en Regresión Polinomial

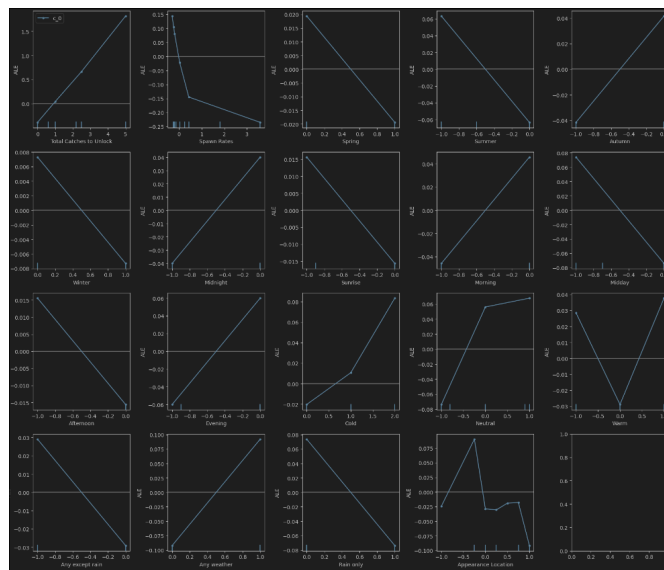


Imagen 4.3. Resultados de ALE en FNN

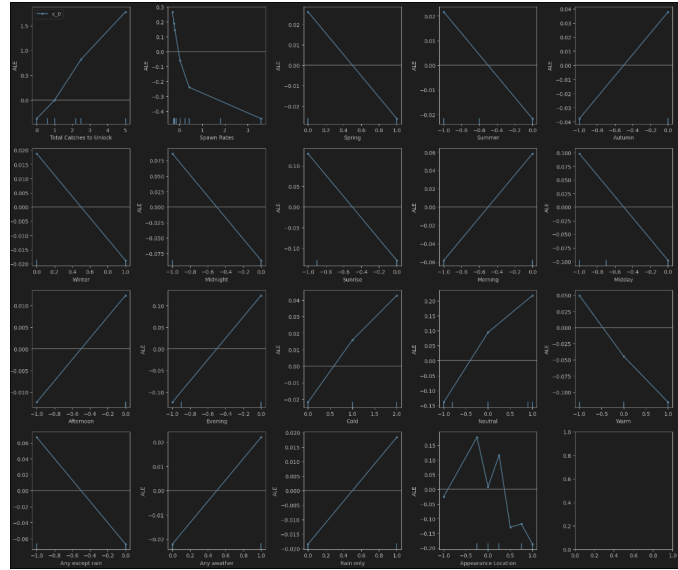


Imagen 4.4. Resultados de ALE en LSTM

Y aunque en las imágenes, debido a su tamaño, no es fácil apreciar detalles, si podemos observar diferentes tendencias a lo largo de todos los modelos. Por ejemplo es fácil apreciar que las estaciones (gráficas 2, 3, 4, 5 siguiendo el orden de cuadrícula siendo el de arriba a la izquierda el 1), a excepción de en el modelo de regresión lineal, mantienen una misma forma en una escala similar. Lo mismo ocurre para la característica “*total_catches_to_unlock*” (gráfica 1), la cual va recibiendo ligeras transformaciones.

Por otra parte también tenemos casos como “*spawn_rates*” la cual va perdiendo importancia según visualizamos los modelos, o “*cold*” y “*appearance_location*” los cuales tienen representaciones (y con ello importancias) bastante diferentes en los 4 modelos.

Ahora ya analizando en detalle las métricas, y no solo las formas de las representaciones, podemos observar lo siguiente:

En todos los modelos la característica más definitoria es la “*total_catches_to_unlock*”, la cual opera en márgenes bastante distantes de los vistos en el resto de gráficas (aunque diferentes para cada modelo según la normalización y desnormalización). Esto encaja perfectamente sabiendo que estamos analizando los datos de un juego en el cual se va progresando y con el progreso aumenta la cantidad de dinero que necesita el jugador.

Una de las características que destaca sobre todo en los modelos de *deep learning* aunque opere en una escala 10 veces menor en otro, tenemos “*spawn_rates*”. Esta, a diferencia de la anteriormente mencionada, afecta de una manera más positiva que negativa, en otras palabras, que un animal sea raro (números bajos en la gráfica) no modifica apenas la predicción (ni positiva ni negativamente), sin embargo si el animal es común, tiene una repercusión negativamente fuerte. De nuevo, esto tiene sentido tanto en la vida real como en el ámbito de estudio y en lo personal me sorprendió que fuera así y no tuviera, en su lugar, una afección positiva.

En cuanto a las estaciones, de los análisis podemos inferir que los peces de Primavera y Otoño tienen en general un precio de venta mayor, contrastando con los encontrados en Verano e Invierno, lo cual puede ser que se diseñara de esta forma ya que la gente suele jugar menos en las primeras estaciones mencionadas.

En cuanto a las horas podemos ver que siguen una estructura de pirámide (a excepción de “*evening*”), con picos alcistas y bajistas en las horas de cambio de momento del día, lo cual me ha llamado la atención, pero no creo que tenga mucha influencia.

En cuanto a los estados temporales (temperatura y lluvia) no puedo concluir nada general debido a las variaciones que se observan en cada modelo, pero sí es interesante cómo cada modelo interpreta y opera con estos datos de una manera diferente.

Y finalmente en los lugares de aparición podemos observar que los modelos de regresión igual se veían afectados por la prioridad establecida durante el tratamiento de los datos que por razones más intrínsecas del modelo y que por ello tiene una importancia muy fuerte en estos casos, mientras que los modelos de IA moderna, o *deep learning*, si iban infiriendo relaciones más estrechas con cada punto, compartiendo un pico en el punto 5 (Aguas Interiores).

4.2. Implementación de SHAP

La implementación del SHAP fue sencilla y común para todos los modelos, debido al cuidado que se llevó a cabo durante su construcción y al empleo de clases durante la misma, donde valores comunes recibieron nombres comunes. Esta fue diseñada para operar por

defecto sobre el primer valor de la muestra de test (igual para todos los modelos al compartir semilla), o, si se define, sobre el primer valor de un nuevo *Dataframe*, aunque este ha de mantener la misma estructura que los modelos de entreno y no contar con la variable “*Sell*”.

Antes de mostrar y comentar los resultados, me gustaría mencionar que existe un problema con el reescalado de los números del resultado que han trabajado con *RobustScaler*, es decir, el FNN y LSTM. Estos devuelven resultados coherentes pero en una escala demasiado baja, y según mis investigaciones, podría ser un problema intrínseco de la clase al contar con varios outliers.

Por otra parte, por su naturaleza y fuerte dependencia a determinadas características, los modelos de regresión son muy radicales, posibilitando la opción de obtener negativos y tendiendo a valores absolutos altos.

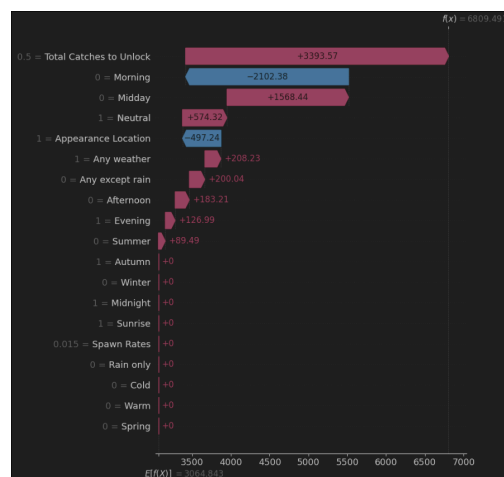


Imagen 4.5. Resultados SHAP en Regresión Lineal

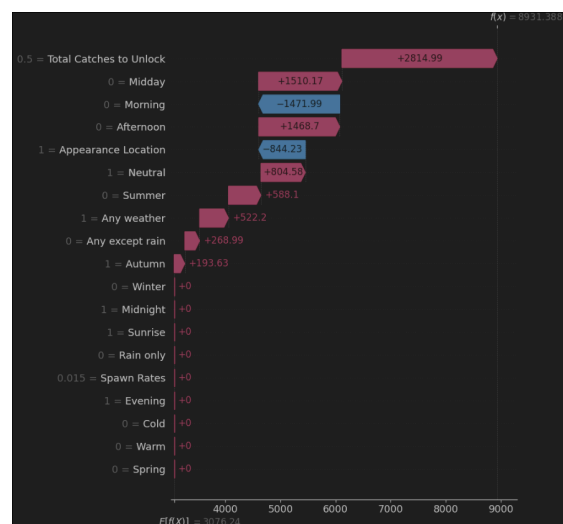


Imagen 4.6. Resultados SHAP en Regresión Polinomial

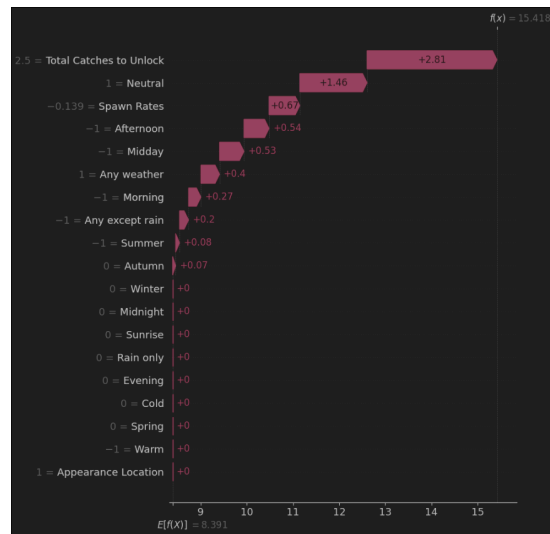


Imagen 4.7. Resultados SHAP en FNN

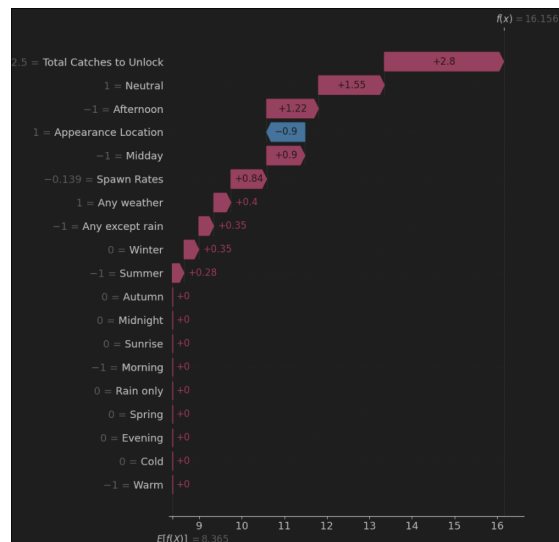


Imagen 4.8. Resultados SHAP en LSTM

En las imágenes 4.5 - 4.8, podemos observar los diferentes resultados de emplear SHAP en los diferentes modelos de IA sobre el mismo parámetro. Si hacemos una comparativa entre todos los modelos podemos observar que independientemente de cual estemos empleando, obtenemos que la característica “*Total Catches to unlock*” es la más definitoria, de acuerdo a lo visto anteriormente en el ALE. A continuación de esta, las características van siendo independientes entre todas las muestras, aunque comparando los resultados brutos que recibimos (6809, 8931, 15, 16) entendiendo las escalas de cada uno (los dos últimos no se han reescalado al haber sido frutos de *RobustScaler*), podemos observar que dan un resultado bastante similar comparándolos 2 a 2.

No contento con estos resultados, tomé la libertad de desarrollar un sistema, simple pero efectivo, de probar diferentes configuraciones.

5. Pruebas de datos

Antes de comenzar me gustaría aclarar lo siguiente: Es completamente necesario haber ejecutado todas las celdas de definición del dataset y de definición de clases y funciones (no las de prueba), para poder ejecutar sin problema esta función.

Dicho esto, podemos proceder a mostrar este rústico pero funcional sistema. Para probarlo, necesitaremos ejecutar la última celda. Una vez hecho eso se te irán haciendo varias preguntas con el fin de “crear” una criatura para predecir su valor.

Finalmente se te pedirá que elijas uno de los 4 modelos con los que hemos trabajado más, se te dirá el valor que predice y el resultado de SHAP para esa muestra.

Con este sistema podemos probar todos los valores que queramos, y es aquí donde se saca a relucir los problemas en los extremos que presentan los modelos de regresión.

6. Conclusiones

Con este nuevo sistema estuve probando diferentes configuraciones sobre todo en los extremos que el ALE plantea en los diferentes modelos. Tenía pensado crear un nuevo apartado donde mi objetivo era comentar resultados de manera específica pero las restricciones temporales me han impedido desarrollar esa labor, al igual que proponer modelos de IA más complejos como pueden ser transformadores.

En lo personal he disfrutado mucho durante todo el proceso de investigación, transformación y manejo de datos, barajando opciones para cada uno de los datos, investigando, planteando e implementando los diferentes modelos de IA y sobre todo analizando y trasteando con diferentes valores en la etapa final.

De nuevo, el tiempo ha sido mi peor enemigo, el cual juntado con trabajos y exámenes para otras asignaturas, han impedido que extendiera más este cotizado modelo. Cotizado, al menos, por mí y algún que otro fan friki de Animal Crossing.

Como nota, también me gustaría agradecer a Marta, mi profesora, la cual me ha enseñado todo lo que este trabajo tiene que ofrecer y me ha resuelto todas mis dudas y dado ideas de cómo proceder en el proyecto.

7. Referencias

https://nookea.com/es_es/ac/MwHcZBfqCJ5vJzRqZ

<https://www.kaggle.com/datasets/jessicali9530/animal-crossing-new-horizons-nookplaza-data-set>

https://en.wikipedia.org/wiki/Least_squares

https://ml4a.github.io/ml4a/es/neural_networks/

<https://jacar.es/la-funcion-leaky-relu-y-su-papel-en-las-redes-neuronales/>

<https://interactivechaos.com/es/manual/tutorial-de-machine-learning/la-funcion-sigmoide>

https://es.wikipedia.org/wiki/Tangente_hiperb%C3%B3lica

<https://www.geeksforgeeks.org/kaiming-initialization-in-deep-learning/>

<https://app.smartdraw.com>