

23 DE MAYO DE 2023

# PRÁCTICA 12

Pronóstico - Árboles de Decisión - Bosques Aleatorios



## Objetivo de la práctica

Por Angel Damian Monroy Mendoza

No. de Cuenta: 316040707

Generar un modelo de pronóstico de las acciones de una determinada empresa mexicana a través de dos algoritmo de aprendizaje automático, en este caso, un modelo de árboles de decisión y otro de bosques aleatorios.



## Características:

- Yahoo Finance ofrece una amplia variedad de datos de mercado sobre acciones, bonos, divisas y criptomonedas.
- También proporciona informes de noticias con varios puntos de vista sobre diferentes mercados de todo el mundo, todos accesibles a través de la biblioteca yfinance.
- En el comercio de acciones, 'alto' y 'bajo' se refieren a los precios máximos y mínimos en un período determinado.
- 'Apertura' y 'cierre' son los precios en los que una acción comenzó y terminó cotizando en el mismo período.
- El 'volumen' es la cantidad total de la actividad comercial.
- Los valores ajustados tienen en cuenta las acciones corporativas, como los 'dividendos', la 'división de acciones' y la emisión de nuevas acciones.

# Desarrollo

Para el desarrollo de esta práctica se obtuvieron dos modelos diferentes. En cada modelo se dividió el proceso en cinco secciones:

1. Importación de las bibliotecas necesarias y datos.
2. Descripción de la estructura de los datos.
3. Aplicación del algoritmo - Árboles de Decisión.
4. Aplicación del algoritmo - Bosques Aleatorios.
5. Comparación.

## 1. Bibliotecas y datos

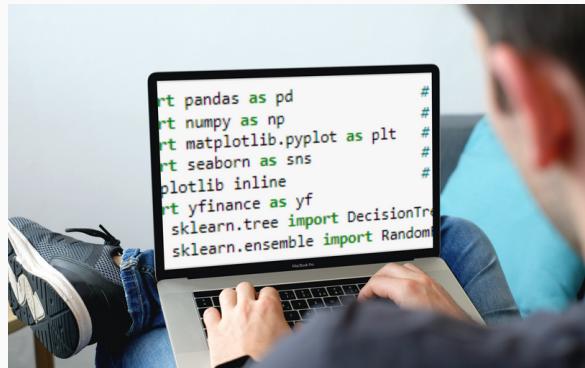
En este apartado se instalaron las bibliotecas necesarias para la manipulación de los datos, creación de vectores y matrices, generación de gráficas, la extracción de datos de yahoo finance y, posteriormente, en la sección de *Aplicación del Algoritmo*, se instalaron las bibliotecas para los Árboles de Decisión y Bosques Aleatorios, así como lo necesario para saber la eficiencia del modelo y su ajuste con *mean\_squared\_error*, *mean\_absolute\_error*, *r2\_score* y *model\_selection*.

Vale la pena comentar un poco de la biblioteca de *yahoo finance* (*yfinance*), que nos permite utilizar datos financieros de empresas que cotizan en diversas bolsas de valores. Esta biblioteca nos permite obtener datos históricos de las acciones de una empresa, pues sólo basta con saber el 'Ticker' de la empresa que se desea analizar y colocarlo dentro del atributo que tiene *yfinance*.

En este caso se analiza la empresa CEMEX con *Ticker = 'CX'* y luego, se obtiene el historial del precio de las acciones, en donde se definió el comienzo del periodo con el 01 - 01 - 2019 y un final del periodo el 16 - 05 - 2023, con un intervalo diario.

```
# Para CEMEX
DataCX = yf.Ticker('CX')

CXHist = DataCX.history(start = '2019-1-1', end = '2023-05-16', interval='1d')
CXHist
```



Date	Open	High	Low	Close	Volume
2019-01-02 00:00:00-05:00	4.686156	4.9229	4.686156	4.9229	1.000000
2019-01-03 00:00:00-05:00	4.883468	4.8834	4.883468	4.883468	1.000000
2019-01-04 00:00:00-05:00	4.844005	4.9031	4.844005	4.9031	1.000000
2019-01-07 00:00:00-05:00	4.893333	5.0413	4.893333	5.0413	1.000000
2019-01-08 00:00:00-05:00	5.051183	5.1399	5.051183	5.1399	1.000000
...	...	...	...	...	...
2023-05-09 00:00:00-04:00	6.500000	6.7700	6.500000	6.7700	1.000000
2023-05-10 00:00:00-04:00	6.820000	6.8500	6.820000	6.8500	1.000000
2023-05-11 00:00:00-04:00	6.570000	6.7000	6.570000	6.7000	1.000000
2023-05-12 00:00:00-04:00	6.680000	6.6800	6.680000	6.6800	1.000000
2023-05-15 00:00:00-04:00	6.550000	6.6400	6.550000	6.6400	1.000000

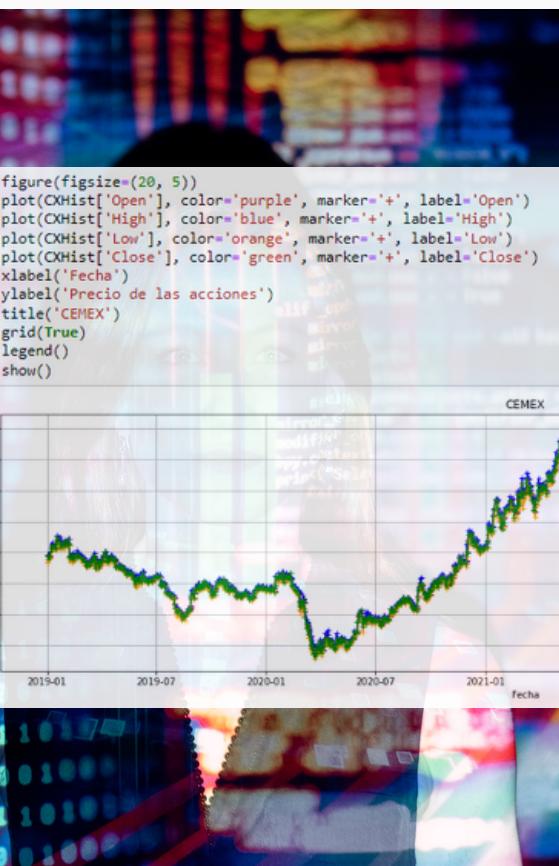
# Desarrollo

## 2. Descripción de la estructura de los datos.

```
DatetimeIndex: 1100 entries - 2019-01-02 00:00:00-05:00 to 2023-05-15 00:00:00-04:00
Data columns (total 7 columns):
 #   Column       Non-Null Count  Dtype  
--- 
 0   Open         1100 non-null    float64
 1   High        1100 non-null    float64
 2   Low          1100 non-null    float64
 3   Close        1100 non-null    float64
 4   Volume       1100 non-null    int64  
 5   Dividends    1100 non-null    float64
 6   Stock Splits 1100 non-null    int64  
dtypes: float64(5), int64(2)
memory usage: 68.0 KB
```

CXHist.describe()

	Open	High	Low	Close	Volume	Dividends	Stock Splits
count	1100.000000	1100.000000	1100.000000	1100.000000	1.000000e+03	1100.000000	1100.0
mean	4.995170	4.951115	4.779512	4.860024	7.609360e+06	0.000045	0.0
std	1.607927	1.626200	1.565193	1.655152	4.478540e+06	0.001504	0.0
min	1.740000	1.790000	1.550000	1.630000	1.050400e+06	0.000000	0.0
25%	3.78997	3.87000	3.736794	3.800000	4.460320e+06	0.000000	0.0
50%	4.47949	4.583750	4.419825	4.494355	6.552100e+06	0.000000	0.0
75%	5.932500	6.655000	5.842500	5.980000	9.598170e+06	0.000000	0.0
max	8.890000	9.090000	8.600000	8.890000	3.605800e+07	0.050000	0.0



En este apartado, se realizó un llamado al método de `.info()` para observar de una forma más condensada los tipos de datos del conjunto de datos, así como verificar que no existieran valores nulos.

Asimismo, se realizó una llamada al método `.describe()` con el fin de observar de una forma condensada algunas características de nuestros datos como: media, desviación, valor mínimo, valor máximo, percentil inferior (25%), 50% y percentil superior (75%).

En este punto logramos descubrir que la variable de *Stock Splits* contiene valores de 0, pues propiedades como el promedio, la desviación estándar o el valor máximo cuentan con el valor de 0.

También se graficaron las características de *Open*, *High*, *Low* & *Close*, debido a que el objetivo era observar cuál fue el comportamiento de los precios de las acciones de la empresa, en el periodo establecido.

Aquí observamos que las cuatro características tienen un comportamiento bastante similar, excepto por determinados momentos puntuales en el '*High*' que subió más que las demás variables, así como en '*Low*', que tuvo bajadas más pronunciadas. Otro punto notable es que, al rededor de marzo de 2020, CEMEX sufrió una caída en el precio de sus acciones, ya que antes de esa fecha sus acciones rondaban los \$45 y posterior a ella cayó al rededor de los \$4, lo cual se explica indudablemente el efecto que tuvo la pandemia de Covid-19 en la sociedad.

A partir de este punto es necesario especificar que para ambos algoritmos se realizaron 2 modelos, uno en donde *no* se toma en cuenta la variable '*Volume*' y otro en donde *sí*, por lo que se referirán como '*1er Modelo*' o '*2do Modelo*'.

### 1er Modelo:

Se realizó una llamada al método `.drop` para eliminar las columnas que no formarían parte de nuestras variables independientes. En este caso se utilizaron las variables de: *Open*, *High*, *Low* & *Close*.

# Desarrollo

## 2do Modelo:

Se realizó una llamada al método `.drop` para eliminar las columnas que no formarían parte de nuestras variables independientes. En este caso se utilizaron las mismas variables del primer modelo más la variable de: **Volume**.

Por último, se realizó un `.dropna` con el fin de corroborar que no tuvieramos valores nulos en ambos modelos, pues cuando obtuvimos la información del `.describe()`, sólo pudimos afirmar que *Dividends* y *Stock Splits*, más no las demás variables.

```
X = np.array(MDatos[['Open',  
'High',  
'Low']])  
pd.DataFrame(X)
```

	0	1	2
0	4.686156	4.922930	4.686156
1	4.883468	4.883468	4.7555215
2	4.844005	4.903199	4.794677

## 3. Aplicación del algoritmo - Árboles de Decisión

Para este apartado se instalaron las bibliotecas necesarias para la aplicación del algoritmo de árboles de decisión explicadas al principio de la práctica. Para ambos modelos se seleccionan las variables predictoras (X) y la variable a pronosticar (Y)

### 1er Modelo:

Variables predictoras: *Open*, *High* & *Low*

Variable a pronosticar: *Close*

### 2do Modelo:

Variables predictoras: *Open*, *High*, *Low* & *Volume*

Variable a pronosticar: *Close*

Después, para ambos modelos se hace la división de los datos utilizando la función de `model_selection` con el método `train_test_split`, en donde se le dan las variables predictoras y la variable a pronosticar; el tamaño del conjunto de testeo, el cual en este caso fue de 20% de prueba (p) y 80% de entrenamiento (e), recordando que es un parámetro que podemos modificar a nuestro criterio, pero que usualmente se deja en estas combinaciones:

- 30% (p) y 70% (e).
- 25% (p) y 75% (e).
- 20% (p) y 80% (e). --> *Elegida*

```
X_train, X_test, Y_train, Y_test = model_selection.  
  
pd.DataFrame(X_test)
```

	0	1	2
0	3.14	3.19	3.06
1	4.64	4.70	4.41
2	4.61	4.70	4.56
3	8.12	8.24	8.08

# Desarrollo

## 3. Aplicación del algoritmo - Árboles de Decisión

Asimismo, se agrego un estado random para volver reproducible los mismos datos obtenidos todas las veces que corramos nuestro código y un `shuffle = 'True'`, que indica que los datos estén barajeados, es decir, que no agarre los primeros datos para el conjunto de entrenamiento y los últimos para el de prueba, sino que sea al azar la asignación de los datos de prueba y entrenamiento.

Posteriormente, en ambos modelos se entrena el modelo a través de Árbol de Decisión Regresor; esto quiere decir que, creamos un objeto para instanciar la función `DecisionTreeRegressor()` junto con hiperparámetros ya ajustados (`max_depth = 9`, `min_samples_split=8`, `min_samples_leaf=4`, `random_state=0`) y luego, a ese objeto se le ajustan los datos de entrenamiento.

A continuación, con estos modelos ya entrenados se generan los pronósticos con el método `.predict` y las variables predictoras del testeo o prueba y se imprimen en un `DataFrame` de pandas.

Por último, se obtienen las características necesarias para poder armar nuestros modelos de árbol de decisión, así como para poder evaluar qué tan buenos son. En este sentido, se imprime el Criterio, la importancia de las variables, el Error Absoluto Medio (MAE), el Error Cuadrático Medio (MSE), la Raíz del Error Cuadrático Medio (RMSE) y el Score o Bondad de Ajuste, en donde se obtuvieron los siguientes valores:

### 1er Modelo:

Criterio: squared\_error

Importancia: [2.79e-04, 9.18e-01, 8.16e-02]

MAE: 0.0658

MSE: 0.0078

RMSE: 0.0883

Score (Bondad de ajuste): 0.9968

### 2do Modelo:

Criterio: squared\_error

Importancia: [2.52e-04, 9.17e-01, 8.16e-02, 1.05e-04]

MAE: 0.0665

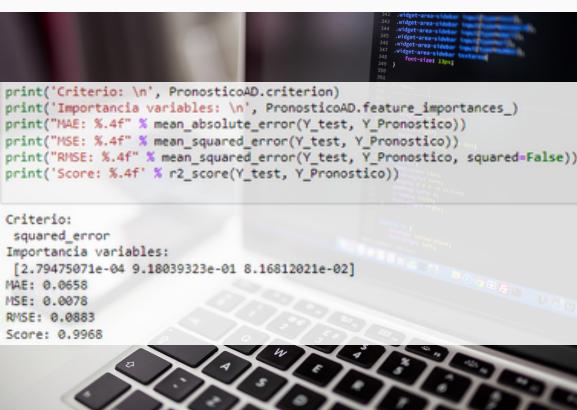
MSE: 0.0079

RMSE: 0.0891

Score (Bondad de ajuste): 0.9967

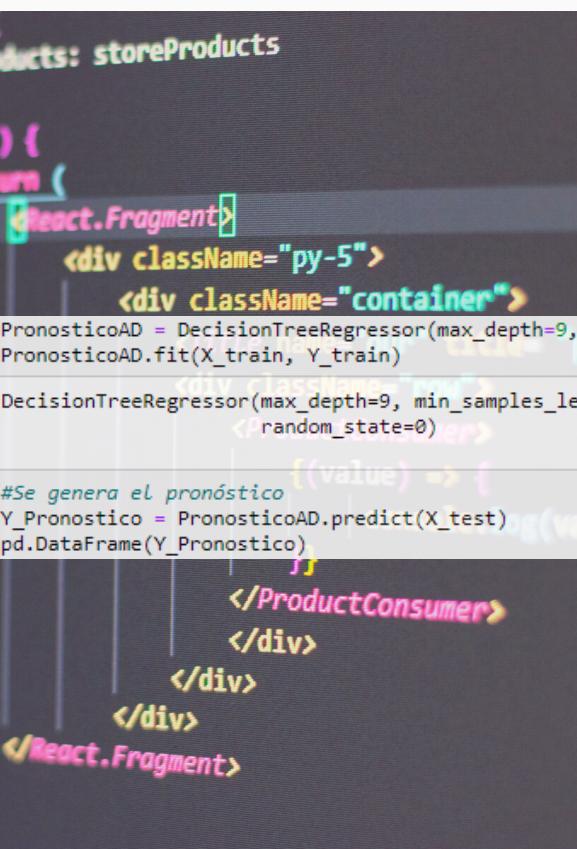
Variable	Importancia
1	High 0.918039
2	Low 0.081681
0	Open 0.000279

Variable	Importancia
1	High 0.917980
2	Low 0.081662
0	Open 0.000253
3	Volume 0.000105



```
print('Criterio: \n', PronosticoAD.criterion)
print('Importancia variables: \n', PronosticoAD.feature_importances_)
print("MAE: %.4f" % mean_absolute_error(Y_test, Y_Pronostico))
print("MSE: %.4f" % mean_squared_error(Y_test, Y_Pronostico))
print("RMSE: %.4f" % mean_squared_error(Y_test, Y_Pronostico, squared=False))
print("Score: %.4f" % r2_score(Y_test, Y_Pronostico))

Criterio:
squared_error
Importancia variables:
[2.79475071e-04 9.18039323e-01 8.16812021e-02]
MAE: 0.0658
MSE: 0.0078
RMSE: 0.0883
Score: 0.9968
```



```
products: storeProducts

)
  ( 
    <React.Fragment>
      <div className="py-5">
        <div className="container">
          PronosticoAD = DecisionTreeRegressor(max_depth=9,
          PronosticoAD.fit(X_train, Y_train)

          DecisionTreeRegressor(max_depth=9, min_samples_le
            <random_state=0>

          #Se genera el pronóstico
          Y_Pronostico = PronosticoAD.predict(X_test)
          pd.DataFrame(Y_Pronostico)
          </value>
        </ProductConsumer>
      </div>
    </div>
  </React.Fragment>
```



# Desarrollo

## 4. Aplicación del algoritmo - Bosques Aleatorios

Para la aplicación del algoritmo de Bosque Aleatorio se realizó un proceso muy similar, pues también se crea un modelo para instanciar la función `RandomForestRegressor()` junto con hiperparámetros ya ajustados (`n_estimators = 105`, `max_depth = 8`, `min_samples_split = 8`, `min_samples_leaf = 4`, `random_state = 0`) y luego, a ese objeto se le ajustan los datos de entrenamiento.

De igual forma se generan los pronósticos con el método `.predict` y las variables predictoras del testeo o prueba y se imprimen en un `DataFrame` de pandas.

Asimismo, se obtienen las mismas métricas que se mencionan arriba de los árboles de decisión:

### 1er Modelo:

Criterio: squared\_error

Importancia: [0.0114139 0.79752895 0.19105715]

MAE: 0.0538

MSE: 0.0049

RMSE: 0.0702

Score (Bondad de ajuste): 0.998

### 2do Modelo:

Criterio: squared\_error

Importancia: [1.18e-02, 7.93e-01, 1.94e-01, 9.93e-05]

MAE: 0.0541

MSE: 0.0050

RMSE: 0.0706

Score (Bondad de ajuste): 0.9979

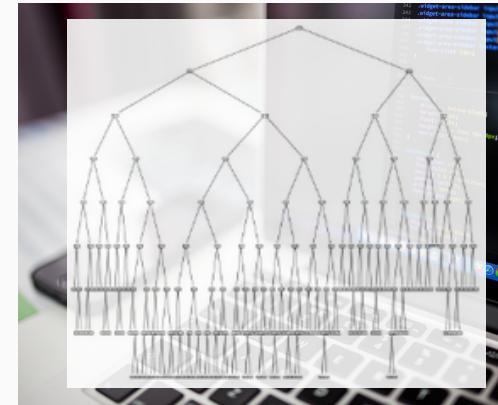
Por último, cabe recalcar que para cada modelo se obtuvo la conformación del Árbol, así como sus respectivas reglas. Cabe mencionar que para el caso de Bosques Aleatorios, no se puede generar directamente el árbol y las reglas, pues se tiene que elegir uno del total de 'estimadores' para poder imprimir estos objetos. Para ambos casos de Bosques Aleatorios se escogió el estimador 50.

Variable	Importancia
1	High 0.797529
2	Low 0.191057
0	Open 0.011414

Variable	Importancia
1	High 0.793732
2	Low 0.194304
0	Open 0.011865
3	Volume 0.000099

```
Estimador = PronosticoBA.estimators_[50]
Estimador

DecisionTreeRegressor(max_depth=8, max_features='auto', min_samples_leaf=4,
min_samples_split=8, random_state=600956192)
```



```
EstimadorBA_M2 = PronosticoBA_M2.estimators_[50]
ReporteBA_M2 = export_text(EstimadorBA_M2, feature_names)
print(ReporteBA_M2)

--- High <= 5.55
|--- High <= 3.80
|   |--- High <= 3.00
|   |   |--- Low <= 2.25
|   |   |   |--- Low <= 1.95
|   |   |   |   |--- High <= 1.97
|   |   |   |   |   |--- value: [1.77]
|   |   |   |   |   |--- High > 1.97
|   |   |   |   |   |--- Low <= 1.88
|   |   |   |   |   |   |--- value: [1.99]
|   |   |   |   |   |--- Low > 1.88
|   |   |   |   |   |   |--- value: [1.93]
|--- Low > 1.95
|   |--- Low <= 2.12
|   |   |--- Volume <= 7185050.00
|   |   |   |--- value: [2.12]
|   |   |   |--- Volume > 7185050.00
|   |   |   |   |--- value: [2.05]
|   |   |   |--- Low > 2.12
|   |   |   |   |--- value: [2.02]

</div>
<React.Fragment>
```

\*Nota: no se presentaron los análisis de las configuraciones debido a que éstas se encuentran en el cuaderno.

# Desarrollo

## Nuevos pronósticos.

Por último, para ambos modelos se realizó un pronóstico de una acción ficticia con ciertos valores de entrada (variables predictoras) y con los modelos que habíamos guardado en los objetos de *PronosticoAD*, *PronosticoAD\_M2*, *PronosticoBA\_M1* y *PronosticoBA\_M2*, respectivamente, se pronosticó la variable objetivo.

## 5. Comparación y Conclusiones

De esta práctica podemos concluir que los modelos de árboles de decisión y bosques aleatorios suelen ser muy eficientes ante conjunto de datos que se logran ajustar bien y el comportamiento de sus variables son parecidos. Asimismo, tal y como comentamos en clase, es importante reconocer que, a pesar de que un árbol de decisión genera buenos rendimientos, es común observar que un bosque aleatorio presente un mejor rendimiento que un árbol debido a que el bosque es una mejora directa de este algoritmo. Esto se logra apreciar en la comparación que se realizó de los 4 modelos. Aquí observamos dos cosas importantes, la primera que la variable 'volume' empeora el rendimiento de los modelos aunque no de forma considerable y, que se confirma que el bosque aleatorio tiene mejor rendimiento que el árbol.

Modelo & Algoritmo	R^2 "Score"	MAE	MSE	RMSE
2 Solo - BA	0.997993	0.053850	0.004924	0.070172
3 Con "Volume" - BA	0.997968	0.054075	0.004985	0.070607
0 Solo - AD	0.996819	0.065840	0.007805	0.088348
1 Con "Volume" - AD	0.996766	0.066549	0.007934	0.089071

```
PrecioAccion = pd.DataFrame({'Open': [6.55],  
                            'High': [6.64],  
                            'Low': [6.5]})  
PronosticoAD.predict(PrecioAccion)  
  
C:\Users\Principal\anaconda3\lib\site-packages  
was fitted without feature names  
warnings.warn(  
array([6.58749998])  
  
PrecioAccionBA_M2 = pd.DataFrame({'Open': [6.55],  
                                  'High': [6.64],  
                                  'Low': [6.5],  
                                  'Volume': [4615500]})  
PronosticoBA_M2.predict(PrecioAccionBA_M2)  
  
C:\Users\Principal\anaconda3\lib\site-packages  
was fitted without feature names  
warnings.warn(  
array([6.57916726])
```

