jrhfm23fj

June 9, 2024

1 Equipo C

2 Predicción diaria del precio de las acciones de la minera FSM utilizando Redes Neuronales Radiales (RBF)

En este notebook, realizaremos la predicción diaria del precio de cierre de las acciones de la minera FSM (Fortuna Silver Mines Inc.) utilizando una Red Neuronal Radial (RBF).

2.1 1. Importación de bibliotecas

[1]:
```
import yfinance as yf  # Para descargar datos de Yahoo Finance
import numpy as np  # Para operaciones matemáticas
import matplotlib.pyplot as plt  # Para graficar
from sklearn.preprocessing import MinMaxScaler  # Para normalizar datos
from sklearn.model_selection import train_test_split  # Para dividir datos en␣
 ↪conjuntos de entrenamiento y prueba
from sklearn.metrics import mean_squared_error  # Para calcular el error␣
 ↪cuadrático medio
from scipy.linalg import pinv  # Para calcular la pseudo-inversa de una matriz
import seaborn as sns  # Para gráficos estadísticos
import matplotlib.dates as mdates  # Para manipulación de fechas en gráficos
```

2.2 2. Descargar datos

Descargamos los datos históricos de las acciones de FSM desde Yahoo Finance.

[2]:
```
# Descargar datos de Yahoo Finance
data = yf.download("FSM", start="2019-01-01", end="2023-12-31")

# Mostrar las primeras filas de los datos
data.head()
```

```
[**********************100%%**********************]  1 of 1 completed
```

[2]:

Date	Open	High	Low	Close	Adj Close	Volume
2019-01-02	3.67	3.77	3.62	3.68	3.68	969400

```
2019-01-03  3.71  3.82  3.68   3.79       3.79  848300
2019-01-04  3.77  3.84  3.73   3.78       3.78  770400
2019-01-07  3.80  3.84  3.67   3.70       3.70  633800
2019-01-08  3.69  3.87  3.67   3.85       3.85  738800
```

2.3 3. Exploración y preprocesamiento de datos

Exploramos los datos descargados y realizamos preprocesamiento si es necesario.

[3]:
```
# Mostrar todos los datos
data
```

[3]:
```
            Open  High   Low  Close  Adj Close   Volume
Date
2019-01-02  3.67  3.77  3.62   3.68       3.68   969400
2019-01-03  3.71  3.82  3.68   3.79       3.79   848300
2019-01-04  3.77  3.84  3.73   3.78       3.78   770400
2019-01-07  3.80  3.84  3.67   3.70       3.70   633800
2019-01-08  3.69  3.87  3.67   3.85       3.85   738800
…           …     …     …      …          …      …
2023-12-22  4.03  4.15  3.96   3.96       3.96  5210000
2023-12-26  3.98  4.00  3.90   3.98       3.98  1946400
2023-12-27  4.01  4.11  3.97   4.00       4.00  4162500
2023-12-28  3.98  4.01  3.89   3.89       3.89  3853300
2023-12-29  3.84  3.92  3.75   3.86       3.86  4770200

[1258 rows x 6 columns]
```

[4]:
```
# Obtener la forma de los datos (número de filas y columnas)
data.shape
```

[4]:
```
(1258, 6)
```

2.3.1 Información general de los datos

[5]:
```
# Mostrar información general de los datos
print("Información general de los datos:")
print(data.info())
```

```
Información general de los datos:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1258 entries, 2019-01-02 to 2023-12-29
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Open       1258 non-null   float64
 1   High       1258 non-null   float64
```

```
 2   Low        1258 non-null   float64
 3   Close      1258 non-null   float64
 4   Adj Close  1258 non-null   float64
 5   Volume     1258 non-null   int64
dtypes: float64(5), int64(1)
memory usage: 68.8 KB
None
```

2.3.2 Estadísticas descriptivas de los datos

[6]:
```
# Mostrar estadísticas descriptivas de los datos
print("\nEstadísticas descriptivas de los datos:")
print(data.describe())
```

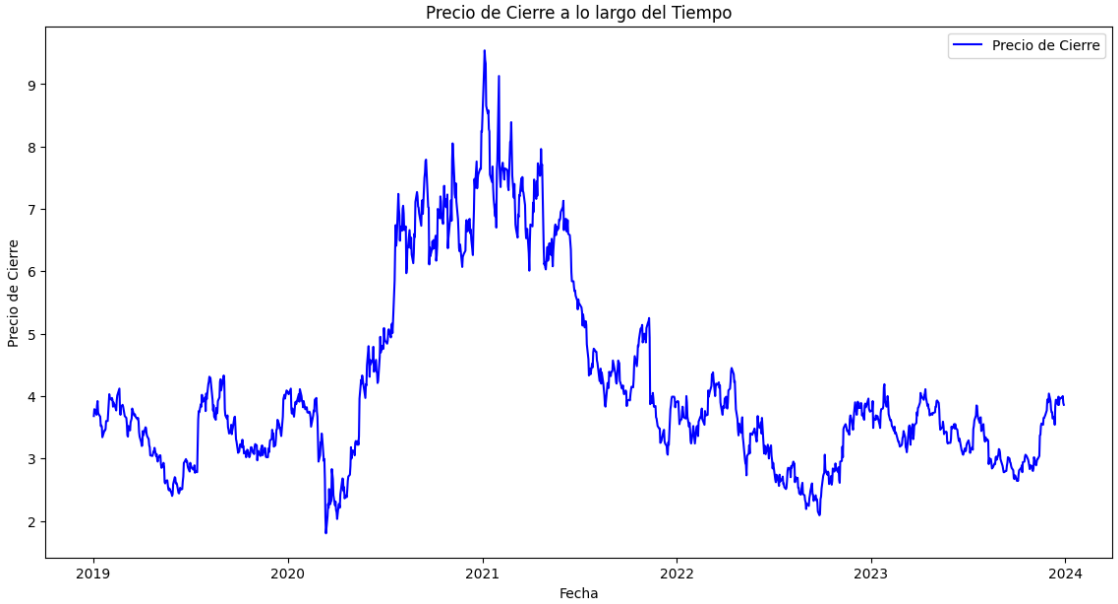
```
Estadísticas descriptivas de los datos:
              Open         High          Low        Close    Adj Close  \
count  1258.000000  1258.000000  1258.000000  1258.000000  1258.000000
mean      4.174300     4.272091     4.064094     4.164523     4.164523
std       1.513419     1.546329     1.466581     1.508172     1.508172
min       1.660000     2.180000     1.470000     1.800000     1.800000
25%       3.152500     3.220000     3.080000     3.160000     3.160000
50%       3.710000     3.780000     3.630000     3.700000     3.700000
75%       4.480000     4.620000     4.330000     4.460000     4.460000
max       9.480000     9.850000     9.120000     9.540000     9.540000

             Volume
count  1.258000e+03
mean   3.736393e+06
std    2.280113e+06
min    2.466000e+05
25%    2.343275e+06
50%    3.372650e+06
75%    4.658625e+06
max    3.585730e+07
```

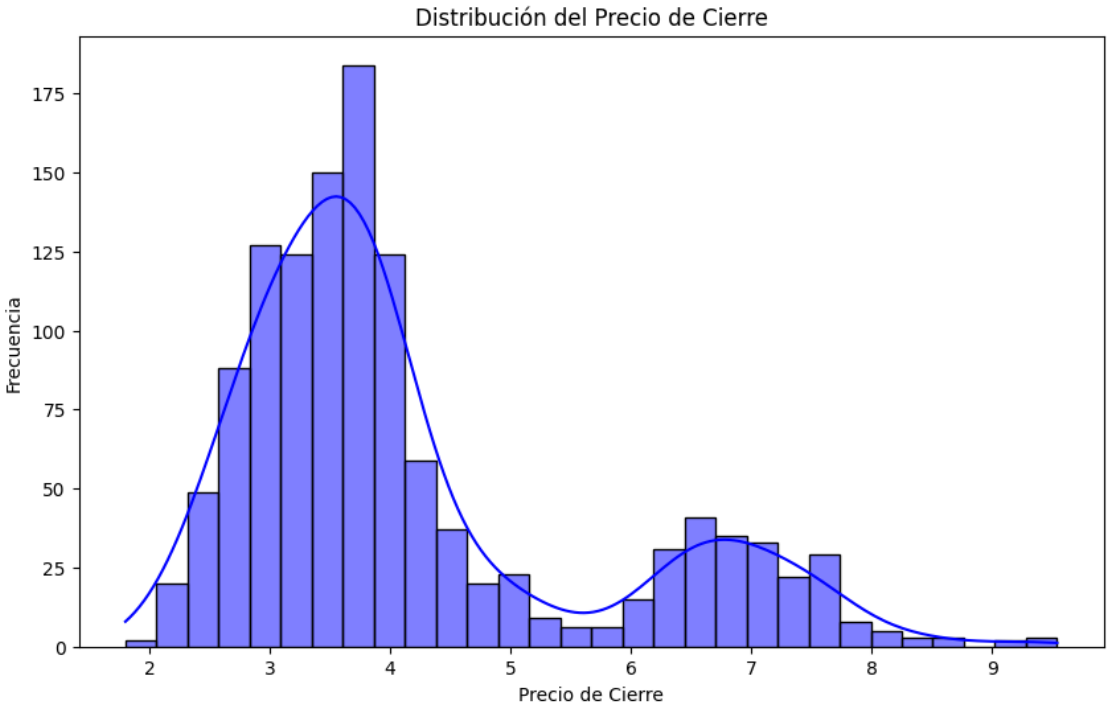
2.4 4. Visualización de datos

Visualizamos los datos para comprender mejor su distribución y tendencias.

[7]:
```
# Graficar el precio de cierre a lo largo del tiempo
plt.figure(figsize=(14, 7))
plt.plot(data.index, data['Close'], label='Precio de Cierre', color='blue')
plt.title('Precio de Cierre a lo largo del Tiempo')
plt.xlabel('Fecha')
plt.ylabel('Precio de Cierre')
plt.legend()
plt.show()
```



[8]:
```
# Graficar la distribución del precio de cierre
plt.figure(figsize=(10, 6))
sns.histplot(data['Close'], bins=30, kde=True, color='blue')
plt.title('Distribución del Precio de Cierre')
plt.xlabel('Precio de Cierre')
plt.ylabel('Frecuencia')
plt.show()
```

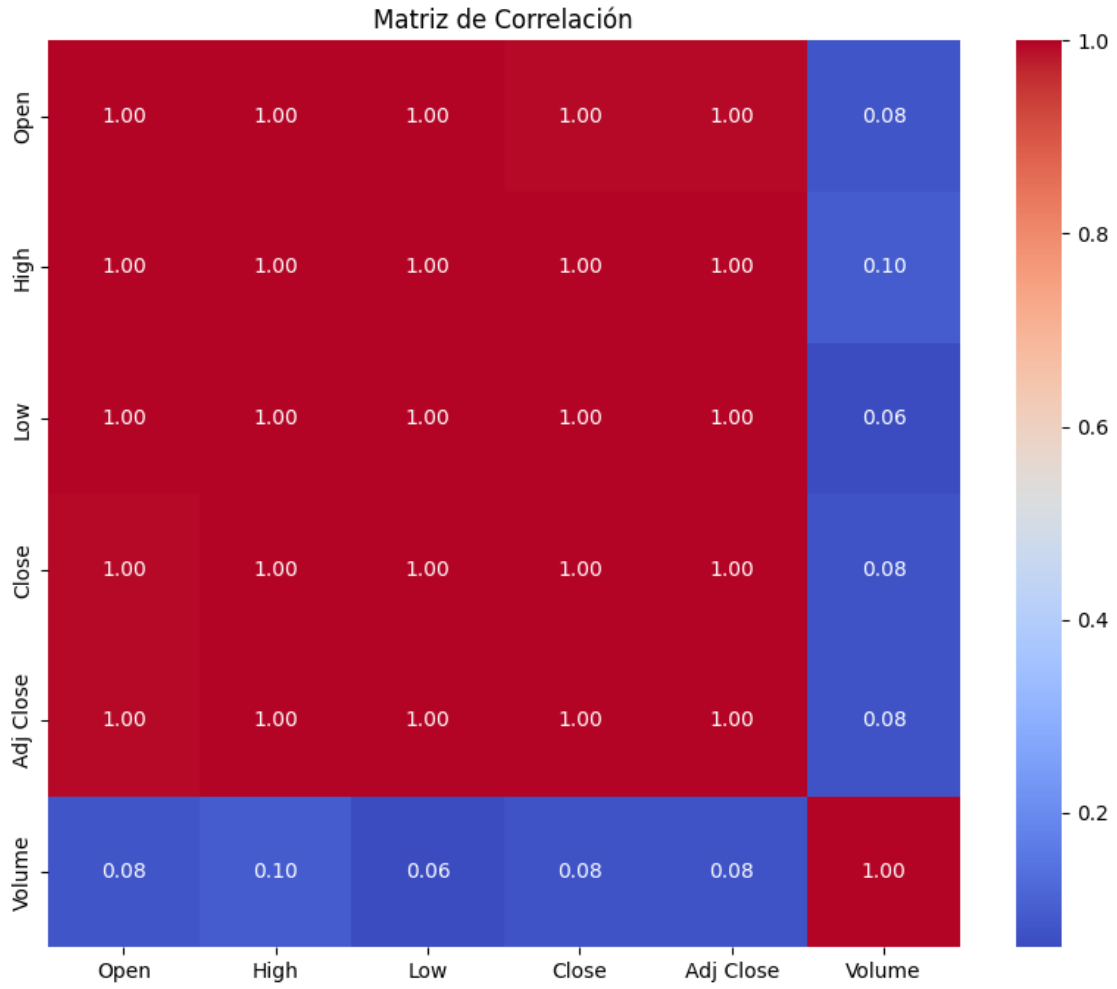


[9]:
```
# Calcular la matriz de correlación entre las variables del conjunto de datos
data.corr()
```

[9]:

	Open	High	Low	Close	Adj Close	Volume
Open	1.000000	0.998105	0.997833	0.995443	0.995443	0.082039
High	0.998105	1.000000	0.997531	0.997901	0.997901	0.096713
Low	0.997833	0.997531	1.000000	0.998080	0.998080	0.060505
Close	0.995443	0.997901	0.998080	1.000000	1.000000	0.075692
Adj Close	0.995443	0.997901	0.998080	1.000000	1.000000	0.075692
Volume	0.082039	0.096713	0.060505	0.075692	0.075692	1.000000

[10]:
```
# Graficar la matriz de correlación
plt.figure(figsize=(10, 8))  # Establecer el tamaño de la figura
sns.heatmap(data.corr(), annot=True, cmap='coolwarm', fmt=".2f")  # Crear el␣
↪mapa de calor con anotaciones y formato decimal de dos lugares
plt.title('Matriz de Correlación')  # Establecer el título del gráfico
plt.show()  # Mostrar el gráfico
```



2.5 5. Preparación de datos para el modelo

Preparamos los datos para el entrenamiento del modelo, incluyendo la normalización y la división en conjuntos de entrenamiento y prueba.

[11]:
```
# Seleccionar las variables 'Open', 'High', 'Low' y 'Close' del conjunto de␣
↪datos y convertirlas en un arreglo de numpy
features = data[['Open', 'High', 'Low', 'Close']].values
features
```

[11]:
```
array([[3.67000008, 3.76999998, 3.61999989, 3.68000007],
       [3.71000004, 3.81999993, 3.68000007, 3.78999996],
       [3.76999998, 3.83999991, 3.73000002, 3.77999997],
       …,
       [4.01000023, 4.11000013, 3.97000003, 4.        ],
       [3.98000002, 4.01000023, 3.8900001 , 3.8900001 ],
```

```
       [3.83999991, 3.92000008, 3.75      , 3.8599999 ]])
```

[12]:
```
# Imprimir la forma (shape) del arreglo de características
print(features.shape)
```

```
(1258, 4)
```

[13]:
```
# Seleccionar la cuarta columna (índice 3) del arreglo de características y␣
 ↪remodelarla para tener una sola característica
features_close = features[:, 3].reshape(-1, 1)
```

[14]:
```
# Normalizar las características entre 0 y 1
scaler_features = MinMaxScaler(feature_range=(0, 1))  # Escalador para las␣
 ↪características
scaler_target = MinMaxScaler(feature_range=(0, 1))  # Escalador para la␣
 ↪variable objetivo

# Escalar todas las características
features_scaled = scaler_features.fit_transform(features)

# Escalar solo la columna 'Close' (variable objetivo)
# Seleccionamos la columna 'Close' (índice 3) y la remodelamos para que tenga␣
 ↪una sola característica (-1, 1)
target_scaled = scaler_target.fit_transform(features[:, 3].reshape(-1, 1))
```

[15]:
```
# Mostrar las características escaladas
print(features_scaled)
```

```
[[0.25703328 0.20730115 0.28104574 0.24289407]
 [0.26214836 0.21382005 0.2888889  0.25710594]
 [0.26982099 0.21642761 0.29542484 0.25581396]
 …
 [0.30051156 0.25162972 0.32679739 0.28423773]
 [0.29667521 0.23859193 0.31633989 0.27002586]
 [0.27877239 0.22685788 0.29803922 0.26614986]]
```

[16]:
```
# Mostrar la variable objetivo escalada
print(target_scaled)
```

```
[[0.24289407]
 [0.25710594]
 [0.25581396]
 …
 [0.28423773]
 [0.27002586]
 [0.26614986]]
```

[17]:
```
# Separar los datos en características (X) y variable objetivo (y)
# Utilizamos los datos hasta el día anterior como características
X = features_scaled[:-1]
# Utilizamos el Close del día siguiente como variable objetivo
y = target_scaled[1:]
```

[18]:
```
# Matriz de características (X)
print(X)
```

```
[[0.25703328 0.20730115 0.28104574 0.24289407]
 [0.26214836 0.21382005 0.2888889  0.25710594]
 [0.26982099 0.21642761 0.29542484 0.25581396]
 …
 [0.29667521 0.23728812 0.31764707 0.28165376]
 [0.30051156 0.25162972 0.32679739 0.28423773]
 [0.29667521 0.23859193 0.31633989 0.27002586]]
```

[19]:
```
# Imprimir la forma (shape) de la matriz de características X
print(X.shape)
```

```
(1257, 4)
```

[20]:
```
# Vector de variable objetivo (y)
print(y)
```

```
[[0.25710594]
 [0.25581396]
 [0.24547805]
 …
 [0.28423773]
 [0.27002586]
 [0.26614986]]
```

[21]:
```
# Imprimir la forma (shape) del vector de variable objetivo y
print(y.shape)
```

```
(1257, 1)
```

[22]:
```
# Separar los datos en conjuntos de entrenamiento y prueba
# Se utiliza un 80% de los datos para entrenamiento y un 20% para prueba
# Los datos no se mezclan (shuffle=False) para mantener el orden temporal
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪shuffle=False)
```

[23]:
```
# Imprimir la forma (shape) de la matriz de características X_train (conjunto␣
 ↪de entrenamiento)
print(X_train.shape)
```

```
(1005, 4)
```

[24]:
```
# Imprimir la forma (shape) del vector de la variable objetivo y_train␣
↪(conjunto de entrenamiento)
print(y_train.shape)
```

```
(1005, 1)
```

[25]:
```
# Imprimir la forma (shape) de la matriz de características X_test (conjunto de␣
↪prueba)
print(X_test.shape)
```

```
(252, 4)
```

[26]:
```
# Imprimir la forma (shape) del conjunto de prueba de la variable objetivo␣
↪(y_test)
print(y_test.shape)
```

```
(252, 1)
```

2.6 6. Definición y entrenamiento del modelo RBF

Definimos una clase RBF para implementar la Red Neuronal Radial y la entrenamos con los datos.

[27]:
```
class RBF:
    def __init__(self, num_hidden, sigma=1.0):
        """
        Inicializa la clase RBF.

        Parámetros:
        - num_hidden: Número de neuronas ocultas en la capa RBF.
        - sigma: Parámetro de dispersión de las funciones gaussianas.
        """
        self.num_hidden = num_hidden
        self.sigma = sigma
        self.centers = None  # Centros de las neuronas RBF
        self.weights = None  # Pesos de salida de la red RBF

    def _gaussian(self, X, centers, sigma):
        """
        Calcula el valor de una función gaussiana.

        Parámetros:
        - X: Datos de entrada.
        - centers: Centros de las funciones gaussianas.
        - sigma: Parámetro de dispersión de las funciones gaussianas.
        """
        return np.exp(-np.linalg.norm(X - centers)**2 / (2 * (sigma ** 2)))
```

```
    def _calculate_interpolation_matrix(self, X):
        """
        Calcula la matriz de interpolación para los datos de entrada.

        Parámetros:
        - X: Datos de entrada.

        Retorna:
        - Matriz de interpolación.
        """
        num_samples = X.shape[0]
        distances = np.zeros((num_samples, self.num_hidden))
        for i in range(num_samples):
            for j in range(self.num_hidden):
                distances[i, j] = np.linalg.norm(X[i] - self.centers[j])
        return np.exp(- (distances ** 2) / (2 * self.sigma ** 2))

    def fit(self, X, y):
        """
        Entrena la red RBF.

        Parámetros:
        - X: Datos de entrada.
        - y: Etiquetas de salida.
        """
        # Seleccionar aleatoriamente los centros de las neuronas
        self.centers = X[np.random.choice(
            X.shape[0], self.num_hidden, replace=False)]
        # Calcular la matriz de interpolación
        Z = self._calculate_interpolation_matrix(X)
        # Calcular los pesos de salida utilizando la pseudo-inversa
        self.weights = np.dot(pinv(Z), y)

    def predict(self, X):
        """
        Realiza predicciones utilizando la red RBF.

        Parámetros:
        - X: Datos de entrada.

        Retorna:
        - Predicciones de salida.
        """
        # Calcular la matriz de interpolación para los nuevos datos
        Z = self._calculate_interpolation_matrix(X)
```

```
        # Realizar las predicciones multiplicando la matriz de interpolación␣
 ↪por los pesos
        return np.dot(Z, self.weights)
```

[28]:
```
# Crear y entrenar el modelo RBF
num_hidden = 10  # Número de neuronas ocultas en la capa RBF (puedes ajustar␣
 ↪este valor según sea necesario)

# Instanciar el modelo RBF con el número de neuronas ocultas especificado
rbf_model = RBF(num_hidden=num_hidden)

# Entrenar el modelo RBF utilizando los datos de entrenamiento
rbf_model.fit(X_train, y_train)
```

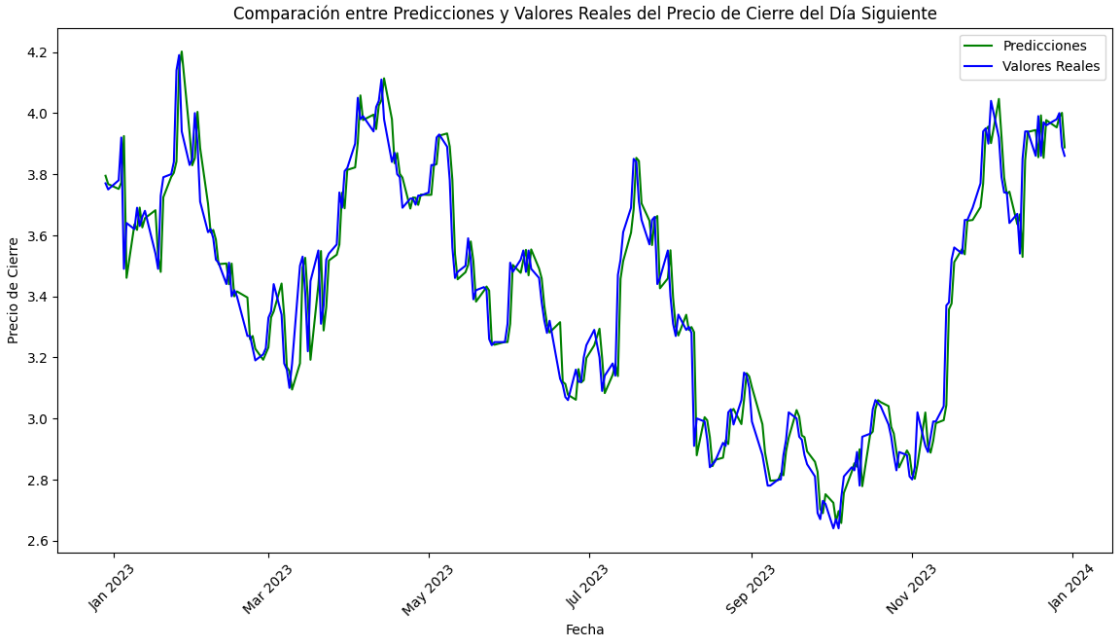
2.7 7. Evaluación del modelo

Evaluamos el modelo utilizando métricas como el MAPE y el RMSE.

[29]:
```
# Realizar predicciones utilizando el modelo RBF entrenado
predictions = rbf_model.predict(X_test)
```

[30]:
```
# Desnormalizar las predicciones y los valores reales solo para la columna␣
 ↪'Close'
predictions = scaler_target.inverse_transform(predictions.reshape(-1, 1))  #␣
 ↪Desnormalizar las predicciones
y_test = scaler_target.inverse_transform(y_test)  # Desnormalizar los valores␣
 ↪reales
```

[31]:
```
# Visualización de los resultados
plt.figure(figsize=(14, 7))  # Tamaño de la figura
plt.plot(data.index[-len(predictions):], predictions,  # Graficar predicciones
         label='Predicciones', color='green')
plt.plot(data.index[-len(y_test):], y_test,  # Graficar valores reales
         label='Valores Reales', color='blue')
plt.title('Comparación entre Predicciones y Valores Reales del Precio de Cierre␣
 ↪del Día Siguiente')  # Título del gráfico
plt.xlabel('Fecha')  # Etiqueta del eje x
plt.ylabel('Precio de Cierre')  # Etiqueta del eje y
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%b %Y'))  # Formato␣
 ↪de las fechas en el eje x
plt.xticks(rotation=45)  # Rotación de las etiquetas del eje x para una mejor␣
 ↪visualización
plt.legend()  # Mostrar leyenda
plt.show()  # Mostrar el gráfico
```



[32]:
```
from sklearn.metrics import mean_absolute_percentage_error

# Importar la función mean_absolute_percentage_error para calcular el MAPE

# Calcular el MAPE para el modelo RBF
mape_rbf = mean_absolute_percentage_error(y_test, predictions)

# Calcular el RMSE para el modelo RBF
rmse_rbf = np.sqrt(mean_squared_error(y_test, predictions))

# Imprimir los resultados de evaluación
print(f"MAPE para el modelo RBF: {mape_rbf}")
print(f"RMSE para el modelo RBF: {rmse_rbf}")
```

```
MAPE para el modelo RBF: 0.021492766331758076
RMSE para el modelo RBF: 0.10195650978486316
```