# Exploratory Analysis and Sales Prediction with LSTM Models

June 9, 2024

## 1 Overview

This notebook conducts an exploratory analysis of sales data, including trend analysis and correlation examination. It also demonstrates the implementation of LSTM (Long Short-Term Memory) models for sales prediction. The models are trained and evaluated using historical sales data, and the notebook provides insights into model performance through visualizations and evaluation metrics.

---

[ ]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout

low_memory = False

# Load datasets
train = pd.read_csv('train.csv', header=0, parse_dates=['Date'])
test = pd.read_csv('test.csv', header=0, parse_dates=['Date'])
store = pd.read_csv('store.csv', header=0)

# Merge train and test data with store data
train = train.merge(store, on='Store', how='left')
test = test.merge(store, on='Store', how='left')

# Fill missing values
train.fillna(0, inplace=True)
test.fillna(0, inplace=True)

# Explore the data
print(train.head())
print(train.info())
```

```
/var/folders/38/q0d38hwd0m9g0ls_dbdnz5bh0000gn/T/ipykernel_11355/3732123763.py:1
2: DtypeWarning: Columns (7) have mixed types. Specify dtype option on import or
```

```
set low_memory=False.
  train = pd.read_csv('train.csv', header=0, parse_dates=['Date'])

   Store  DayOfWeek       Date  Sales  Customers  Open  Promo StateHoliday  \
0      1          5 2015-07-31   5263        555     1      1            0
1      2          5 2015-07-31   6064        625     1      1            0
2      3          5 2015-07-31   8314        821     1      1            0
3      4          5 2015-07-31  13995       1498     1      1            0
4      5          5 2015-07-31   4822        559     1      1            0

   SchoolHoliday StoreType Assortment  CompetitionDistance  \
0              1         c          a               1270.0
1              1         a          a                570.0
2              1         a          a              14130.0
3              1         c          c                620.0
4              1         a          a              29910.0

   CompetitionOpenSinceMonth  CompetitionOpenSinceYear  Promo2  \
0                        9.0                    2008.0       0
1                       11.0                    2007.0       1
2                       12.0                    2006.0       1
3                        9.0                    2009.0       0
4                        4.0                    2015.0       0

   Promo2SinceWeek  Promo2SinceYear    PromoInterval
0              0.0              0.0                0
1             13.0           2010.0  Jan,Apr,Jul,Oct
2             14.0           2011.0  Jan,Apr,Jul,Oct
3              0.0              0.0                0
4              0.0              0.0                0
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1017209 entries, 0 to 1017208
Data columns (total 18 columns):
 #   Column                     Non-Null Count    Dtype
---  ------                     --------------    -----
 0   Store                      1017209 non-null  int64
 1   DayOfWeek                  1017209 non-null  int64
 2   Date                       1017209 non-null  datetime64[ns]
 3   Sales                      1017209 non-null  int64
 4   Customers                  1017209 non-null  int64
 5   Open                       1017209 non-null  int64
 6   Promo                      1017209 non-null  int64
 7   StateHoliday               1017209 non-null  object
 8   SchoolHoliday              1017209 non-null  int64
 9   StoreType                  1017209 non-null  object
 10  Assortment                 1017209 non-null  object
 11  CompetitionDistance        1017209 non-null  float64
 12  CompetitionOpenSinceMonth  1017209 non-null  float64
```

```
 13  CompetitionOpenSinceYear   1017209 non-null  float64
 14  Promo2                     1017209 non-null  int64
 15  Promo2SinceWeek            1017209 non-null  float64
 16  Promo2SinceYear            1017209 non-null  float64
 17  PromoInterval              1017209 non-null  object
dtypes: datetime64[ns](1), float64(5), int64(8), object(4)
memory usage: 139.7+ MB
None
```

# 2 Explanation of Code

This code snippet demonstrates how to load and preprocess datasets using pandas and scikit-learn, and then build a LSTM (Long Short-Term Memory) neural network using Keras for predictive modeling.

## 2.1 Libraries Imported

- **pandas**: Used for data manipulation and analysis.
- **numpy**: Provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
- **matplotlib.pyplot**: A plotting library for creating static, interactive, and animated visualizations in Python.
- **seaborn**: Built on top of matplotlib, seaborn provides a high-level interface for drawing attractive and informative statistical graphics.
- **sklearn.preprocessing.MinMaxScaler**: Used for scaling numerical features to a specified range, in this case between 0 and 1.
- **keras.models.Sequential**: Provides the core functionality for defining a sequence of neural network layers.
- **keras.layers.LSTM, keras.layers.Dense, keras.layers.Dropout**: Layers used for building LSTM-based neural networks.

## 2.2 Loading and Preprocessing Data

1. **Load Datasets**: Three datasets are loaded from CSV files: `train.csv`, `test.csv`, and `store.csv`.
2. **Merge Data**: The `train` and `test` datasets are merged with the `store` dataset based on the 'Store' column.
3. **Fill Missing Values**: Any missing values in the datasets are filled with 0.

## 2.3 Exploring the Data

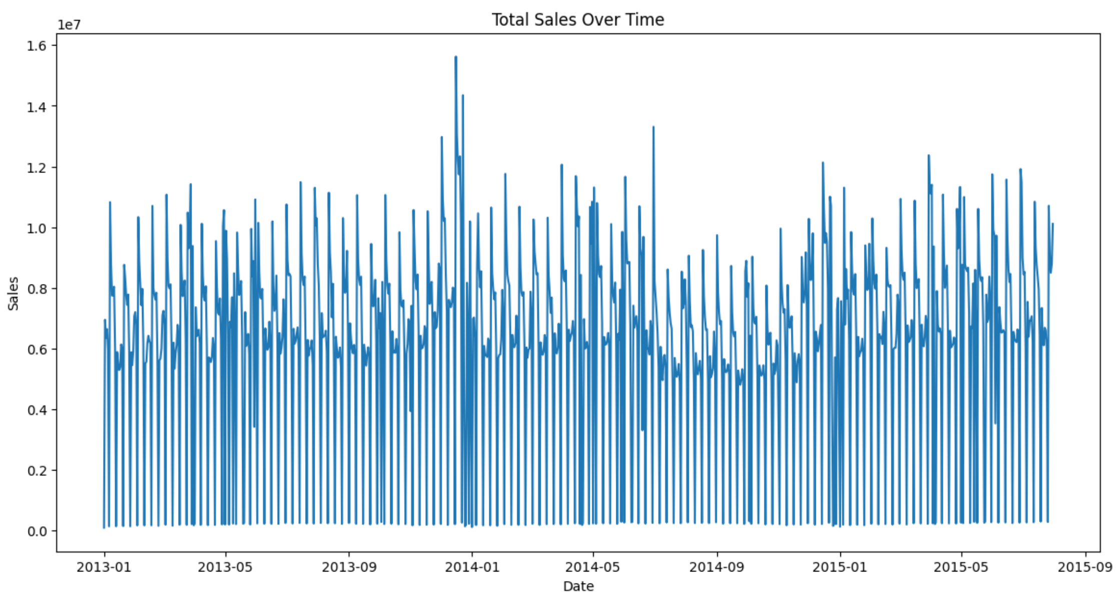
- `print(train.head())` displays the first few rows of the training dataset.
- `print(train.info())` provides information about the training dataset, including the data types and non-null counts of each column.

The code sets `low_memory` to `False`, indicating that pandas should not guess the dtype of each column in the CSV file. This can be useful for large datasets to avoid memory issues during file reading.

This code serves as a foundational step for data preprocessing and model building in a machine learning pipeline.

[ ]:
```
# Analyze sales trends, correlations, etc.
sales_summary = train.groupby('Date')['Sales'].sum()
print(sales_summary.head())

# Plot sales trends
plt.figure(figsize=(14, 7))
plt.plot(sales_summary)
plt.title('Total Sales Over Time')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.show()
```

```
Date
2013-01-01      97235
2013-01-02    6949829
2013-01-03    6347820
2013-01-04    6638954
2013-01-05    5951593
Name: Sales, dtype: int64
```




# 3 Analyzing Sales Trends

This code snippet analyzes sales trends over time using the training dataset.

### 3.1 Calculating Sales Summary

- `sales_summary = train.groupby('Date')['Sales'].sum()` aggregates the sales data by date, summing up the sales for each date. This creates a pandas Series object where the index represents dates and the values represent total sales for each date.
- `print(sales_summary.head())` displays the first few entries of the sales summary, providing a glimpse of the aggregated sales data.

### 3.2 Plotting Sales Trends

- `plt.figure(figsize=(14, 7))` initializes a matplotlib figure with a specified size (width: 14 inches, height: 7 inches) to ensure the plot is visually appealing.
- `plt.plot(sales_summary)` plots the sales summary data on the initialized figure. The x-axis represents dates, and the y-axis represents total sales.
- `plt.title('Total Sales Over Time')` sets the title of the plot to 'Total Sales Over Time'.
- `plt.xlabel('Date')` and `plt.ylabel('Sales')` label the x-axis and y-axis, respectively.
- `plt.show()` displays the plot.

This visualization provides insight into the overall sales trends over time, allowing for the identification of patterns, seasonality, and potential anomalies in the sales data.

[ ]:
```
import tensorflow as tf

# Seeds for reproductivity
np.random.seed(42)
tf.random.set_seed(42)

# Filter out closed stores
train = train[train['Open'] == 1]

# Sort by date
train.sort_values('Date', inplace=True)

# Extract the sales data
sales_data = train[['Date', 'Sales']].set_index('Date')

# Resample to daily frequency and fill missing values
sales_data = sales_data.resample('D').sum().fillna(0)


# Normalize the sales data for LSTM
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_sales = scaler.fit_transform(sales_data)
```

# 4 Preprocessing Sales Data for LSTM Model

This code snippet demonstrates the preprocessing steps required to prepare sales data for training an LSTM (Long Short-Term Memory) model using TensorFlow.

### 4.1 Setting Seeds for Reproducibility

- `np.random.seed(42)` and `tf.random.set_seed(42)` set the random seeds for NumPy and TensorFlow, respectively. This ensures that the random number generation is reproducible across different runs of the code, which is important for debugging and comparing results.

### 4.2 Filtering Closed Stores and Sorting by Date

- `train = train[train['Open'] == 1]` filters out closed stores from the training dataset, retaining only the data for open stores.
- `train.sort_values('Date', inplace=True)` sorts the training data by date in ascending order. This is important for time-series data analysis and modeling.

### 4.3 Extracting and Resampling Sales Data

- `sales_data = train[['Date', 'Sales']].set_index('Date')` extracts the sales data from the training dataset and sets the date column as the index.
- `sales_data = sales_data.resample('D').sum().fillna(0)` resamples the sales data to a daily frequency (using 'D' for day) and fills any missing values with 0. This ensures that there is a sales value for every day, even if it's 0.

### 4.4 Normalizing Sales Data for LSTM

- `scaler = MinMaxScaler(feature_range=(0, 1))` initializes a MinMaxScaler object with a feature range of 0 to 1, which is commonly used for normalizing data for neural networks.
- `scaled_sales = scaler.fit_transform(sales_data)` normalizes the sales data using the MinMaxScaler, scaling the values to the specified feature range.

These preprocessing steps are essential for preparing the sales data to be fed into the LSTM model, ensuring that it is properly formatted and scaled for training.

[ ]:
```
# Prepare the LSTM input format (features and labels)
def create_dataset(data, time_step=1):
    X, Y = [], []
    for i in range(len(data) - time_step - 1):
        a = data[i:(i + time_step), 0]
        X.append(a)
        Y.append(data[i + time_step, 0])
    return np.array(X), np.array(Y)

# Define the time step
time_step = 60  # This means using the past 60 days to predict the next day

# Create the dataset
X, Y = create_dataset(scaled_sales, time_step)

# Reshape the input to be [samples, time steps, features] which is required for
 ↪LSTM
X = X.reshape(X.shape[0], X.shape[1], 1)
```

# 5 Creating LSTM Input Dataset

This code snippet demonstrates how to prepare the input dataset for training an LSTM (Long Short-Term Memory) model.

## 5.1 Creating Dataset Function

- `def create_dataset(data, time_step=1):` defines a function `create_dataset` that takes the scaled sales data and a time step as input parameters.
- The function iterates through the data to create sequences of input features (X) and corresponding labels (Y).
- For each time step, it creates an input sequence (X) by selecting the previous `time_step` days of sales data.
- It creates the corresponding label (Y) by selecting the sales value for the next day.
- The function returns numpy arrays `X` and `Y`, representing the input features and labels, respectively.

## 5.2 Defining Time Step

- `time_step = 60` sets the time step to 60 days, indicating that the model will use the past 60 days of sales data to predict the sales for the next day.

## 5.3 Creating Dataset

- `X, Y = create_dataset(scaled_sales, time_step)` calls the `create_dataset` function to create the input dataset using the scaled sales data and the specified time step.
- `X` contains the input features (sequences of past sales data), while `Y` contains the corresponding labels (sales values for the next day).

## 5.4 Reshaping Input Data

- `X = X.reshape(X.shape[0], X.shape[1], 1)` reshapes the input features `X` to match the required input format for LSTM models, which is `[samples, time steps, features]`.
- Here, `X.shape[0]` represents the number of samples (sequences), `X.shape[1]` represents the number of time steps, and `1` represents the number of features (sales data).

These steps prepare the input dataset in the appropriate format for training the LSTM model, enabling it to learn patterns from historical sales data and make predictions for future sales.

[ ]:
```
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout
from keras.callbacks import EarlyStopping


# Define a complex LSTM model
model = Sequential()
model.add(LSTM(100, return_sequences=True, input_shape=(time_step, X_train.
 ↪shape[2])))
model.add(LSTM(100, return_sequences=False))
model.add(Dropout(0.2))
```

```
model.add(Dense(50, activation='relu'))
model.add(Dense(1))

model.compile(optimizer='adam', loss='mean_squared_error')

# Increase the number of epochs and use early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=40,␣
↪restore_best_weights=True)

history = model.fit(X_train, Y_train, epochs=200, batch_size=64,␣
↪validation_data=(X_test, Y_test), callbacks=[early_stopping], verbose=1)
```

```
Epoch 1/200
```

```
/Users/michaelwilliams/anaconda3/envs/jupyter_env/lib/python3.11/site-
packages/keras/src/layers/rnn/rnn.py:204: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

```
10/10            2s 79ms/step -
loss: 0.1193 - val_loss: 0.0478
Epoch 2/200
10/10            1s 63ms/step -
loss: 0.0477 - val_loss: 0.0492
Epoch 3/200
10/10            1s 62ms/step -
loss: 0.0481 - val_loss: 0.0476
Epoch 4/200
10/10            1s 62ms/step -
loss: 0.0456 - val_loss: 0.0455
Epoch 5/200
10/10            1s 73ms/step -
loss: 0.0455 - val_loss: 0.0453
Epoch 6/200
10/10            1s 63ms/step -
loss: 0.0447 - val_loss: 0.0453
Epoch 7/200
10/10            1s 62ms/step -
loss: 0.0437 - val_loss: 0.0449
Epoch 8/200
10/10            1s 66ms/step -
loss: 0.0454 - val_loss: 0.0448
Epoch 9/200
10/10            1s 65ms/step -
loss: 0.0451 - val_loss: 0.0447
Epoch 10/200
10/10            1s 68ms/step -
loss: 0.0436 - val_loss: 0.0445
```

```
Epoch 11/200
10/10              1s 68ms/step -
loss: 0.0445 - val_loss: 0.0443
Epoch 12/200
10/10              1s 69ms/step -
loss: 0.0443 - val_loss: 0.0443
Epoch 13/200
10/10              1s 67ms/step -
loss: 0.0437 - val_loss: 0.0442
Epoch 14/200
10/10              1s 66ms/step -
loss: 0.0442 - val_loss: 0.0441
Epoch 15/200
10/10              1s 74ms/step -
loss: 0.0449 - val_loss: 0.0440
Epoch 16/200
10/10              1s 68ms/step -
loss: 0.0444 - val_loss: 0.0439
Epoch 17/200
10/10              1s 68ms/step -
loss: 0.0433 - val_loss: 0.0438
Epoch 18/200
10/10              1s 67ms/step -
loss: 0.0437 - val_loss: 0.0436
Epoch 19/200
10/10              1s 67ms/step -
loss: 0.0440 - val_loss: 0.0438
Epoch 20/200
10/10              1s 71ms/step -
loss: 0.0435 - val_loss: 0.0437
Epoch 21/200
10/10              1s 67ms/step -
loss: 0.0445 - val_loss: 0.0435
Epoch 22/200
10/10              1s 67ms/step -
loss: 0.0432 - val_loss: 0.0434
Epoch 23/200
10/10              1s 67ms/step -
loss: 0.0434 - val_loss: 0.0433
Epoch 24/200
10/10              1s 67ms/step -
loss: 0.0435 - val_loss: 0.0432
Epoch 25/200
10/10              1s 66ms/step -
loss: 0.0428 - val_loss: 0.0431
Epoch 26/200
10/10              1s 66ms/step -
loss: 0.0427 - val_loss: 0.0432
```

```
Epoch 27/200
10/10             1s 74ms/step -
loss: 0.0427 - val_loss: 0.0432
Epoch 28/200
10/10             1s 67ms/step -
loss: 0.0426 - val_loss: 0.0429
Epoch 29/200
10/10             1s 67ms/step -
loss: 0.0421 - val_loss: 0.0427
Epoch 30/200
10/10             1s 67ms/step -
loss: 0.0421 - val_loss: 0.0427
Epoch 31/200
10/10             1s 67ms/step -
loss: 0.0424 - val_loss: 0.0426
Epoch 32/200
10/10             1s 72ms/step -
loss: 0.0422 - val_loss: 0.0426
Epoch 33/200
10/10             1s 68ms/step -
loss: 0.0413 - val_loss: 0.0425
Epoch 34/200
10/10             1s 68ms/step -
loss: 0.0415 - val_loss: 0.0424
Epoch 35/200
10/10             1s 70ms/step -
loss: 0.0413 - val_loss: 0.0425
Epoch 36/200
10/10             1s 75ms/step -
loss: 0.0411 - val_loss: 0.0427
Epoch 37/200
10/10             1s 70ms/step -
loss: 0.0411 - val_loss: 0.0427
Epoch 38/200
10/10             1s 70ms/step -
loss: 0.0411 - val_loss: 0.0425
Epoch 39/200
10/10             1s 69ms/step -
loss: 0.0415 - val_loss: 0.0425
Epoch 40/200
10/10             1s 70ms/step -
loss: 0.0409 - val_loss: 0.0427
Epoch 41/200
10/10             1s 74ms/step -
loss: 0.0413 - val_loss: 0.0427
Epoch 42/200
10/10             1s 75ms/step -
loss: 0.0411 - val_loss: 0.0428
```

```
Epoch 43/200
10/10            1s 70ms/step -
loss: 0.0405 - val_loss: 0.0431
Epoch 44/200
10/10            1s 70ms/step -
loss: 0.0406 - val_loss: 0.0429
Epoch 45/200
10/10            1s 71ms/step -
loss: 0.0409 - val_loss: 0.0431
Epoch 46/200
10/10            1s 71ms/step -
loss: 0.0401 - val_loss: 0.0433
Epoch 47/200
10/10            1s 69ms/step -
loss: 0.0409 - val_loss: 0.0430
Epoch 48/200
10/10            1s 69ms/step -
loss: 0.0410 - val_loss: 0.0433
Epoch 49/200
10/10            1s 68ms/step -
loss: 0.0395 - val_loss: 0.0431
Epoch 50/200
10/10            1s 73ms/step -
loss: 0.0400 - val_loss: 0.0430
Epoch 51/200
10/10            1s 68ms/step -
loss: 0.0404 - val_loss: 0.0429
Epoch 52/200
10/10            1s 68ms/step -
loss: 0.0405 - val_loss: 0.0434
Epoch 53/200
10/10            1s 68ms/step -
loss: 0.0407 - val_loss: 0.0438
Epoch 54/200
10/10            1s 68ms/step -
loss: 0.0405 - val_loss: 0.0433
Epoch 55/200
10/10            1s 69ms/step -
loss: 0.0402 - val_loss: 0.0431
Epoch 56/200
10/10            1s 68ms/step -
loss: 0.0401 - val_loss: 0.0429
Epoch 57/200
10/10            1s 68ms/step -
loss: 0.0399 - val_loss: 0.0431
Epoch 58/200
10/10            1s 69ms/step -
loss: 0.0403 - val_loss: 0.0432
```

```
Epoch 59/200
10/10            1s 73ms/step -
loss: 0.0398 - val_loss: 0.0424
Epoch 60/200
10/10            1s 69ms/step -
loss: 0.0412 - val_loss: 0.0427
Epoch 61/200
10/10            1s 69ms/step -
loss: 0.0402 - val_loss: 0.0439
Epoch 62/200
10/10            1s 69ms/step -
loss: 0.0397 - val_loss: 0.0431
Epoch 63/200
10/10            1s 69ms/step -
loss: 0.0398 - val_loss: 0.0431
Epoch 64/200
10/10            1s 68ms/step -
loss: 0.0401 - val_loss: 0.0429
Epoch 65/200
10/10            1s 70ms/step -
loss: 0.0401 - val_loss: 0.0440
Epoch 66/200
10/10            1s 73ms/step -
loss: 0.0397 - val_loss: 0.0419
Epoch 67/200
10/10            1s 69ms/step -
loss: 0.0410 - val_loss: 0.0426
Epoch 68/200
10/10            1s 68ms/step -
loss: 0.0408 - val_loss: 0.0435
Epoch 69/200
10/10            1s 69ms/step -
loss: 0.0400 - val_loss: 0.0433
Epoch 70/200
10/10            1s 69ms/step -
loss: 0.0400 - val_loss: 0.0438
Epoch 71/200
10/10            1s 68ms/step -
loss: 0.0403 - val_loss: 0.0437
Epoch 72/200
10/10            1s 68ms/step -
loss: 0.0401 - val_loss: 0.0435
Epoch 73/200
10/10            1s 68ms/step -
loss: 0.0396 - val_loss: 0.0431
Epoch 74/200
10/10            1s 68ms/step -
loss: 0.0400 - val_loss: 0.0431
```

```
Epoch 75/200
10/10           1s 73ms/step -
loss: 0.0392 - val_loss: 0.0429
Epoch 76/200
10/10           1s 69ms/step -
loss: 0.0392 - val_loss: 0.0428
Epoch 77/200
10/10           1s 68ms/step -
loss: 0.0395 - val_loss: 0.0428
Epoch 78/200
10/10           1s 68ms/step -
loss: 0.0392 - val_loss: 0.0423
Epoch 79/200
10/10           1s 68ms/step -
loss: 0.0396 - val_loss: 0.0422
Epoch 80/200
10/10           1s 68ms/step -
loss: 0.0399 - val_loss: 0.0417
Epoch 81/200
10/10           1s 68ms/step -
loss: 0.0392 - val_loss: 0.0411
Epoch 82/200
10/10           1s 68ms/step -
loss: 0.0391 - val_loss: 0.0416
Epoch 83/200
10/10           1s 68ms/step -
loss: 0.0394 - val_loss: 0.0415
Epoch 84/200
10/10           1s 73ms/step -
loss: 0.0383 - val_loss: 0.0413
Epoch 85/200
10/10           1s 68ms/step -
loss: 0.0397 - val_loss: 0.0416
Epoch 86/200
10/10           1s 68ms/step -
loss: 0.0397 - val_loss: 0.0415
Epoch 87/200
10/10           1s 68ms/step -
loss: 0.0383 - val_loss: 0.0397
Epoch 88/200
10/10           1s 68ms/step -
loss: 0.0413 - val_loss: 0.0422
Epoch 89/200
10/10           1s 68ms/step -
loss: 0.0383 - val_loss: 0.0409
Epoch 90/200
10/10           1s 68ms/step -
loss: 0.0385 - val_loss: 0.0406
```

```
Epoch 91/200
10/10              1s 68ms/step -
loss: 0.0375 - val_loss: 0.0399
Epoch 92/200
10/10              1s 71ms/step -
loss: 0.0374 - val_loss: 0.0395
Epoch 93/200
10/10              1s 74ms/step -
loss: 0.0377 - val_loss: 0.0389
Epoch 94/200
10/10              1s 68ms/step -
loss: 0.0383 - val_loss: 0.0388
Epoch 95/200
10/10              1s 68ms/step -
loss: 0.0376 - val_loss: 0.0390
Epoch 96/200
10/10              1s 69ms/step -
loss: 0.0365 - val_loss: 0.0380
Epoch 97/200
10/10              1s 68ms/step -
loss: 0.0372 - val_loss: 0.0386
Epoch 98/200
10/10              1s 68ms/step -
loss: 0.0373 - val_loss: 0.0381
Epoch 99/200
10/10              1s 69ms/step -
loss: 0.0364 - val_loss: 0.0383
Epoch 100/200
10/10              1s 68ms/step -
loss: 0.0358 - val_loss: 0.0380
Epoch 101/200
10/10              1s 73ms/step -
loss: 0.0366 - val_loss: 0.0384
Epoch 102/200
10/10              1s 68ms/step -
loss: 0.0369 - val_loss: 0.0370
Epoch 103/200
10/10              1s 69ms/step -
loss: 0.0357 - val_loss: 0.0367
Epoch 104/200
10/10              1s 68ms/step -
loss: 0.0345 - val_loss: 0.0362
Epoch 105/200
10/10              1s 68ms/step -
loss: 0.0341 - val_loss: 0.0362
Epoch 106/200
10/10              1s 69ms/step -
loss: 0.0338 - val_loss: 0.0370
```

```
Epoch 107/200
10/10            1s 68ms/step -
loss: 0.0349 - val_loss: 0.0365
Epoch 108/200
10/10            1s 69ms/step -
loss: 0.0341 - val_loss: 0.0356
Epoch 109/200
10/10            1s 68ms/step -
loss: 0.0335 - val_loss: 0.0358
Epoch 110/200
10/10            1s 73ms/step -
loss: 0.0324 - val_loss: 0.0356
Epoch 111/200
10/10            1s 68ms/step -
loss: 0.0323 - val_loss: 0.0348
Epoch 112/200
10/10            1s 68ms/step -
loss: 0.0326 - val_loss: 0.0369
Epoch 113/200
10/10            1s 69ms/step -
loss: 0.0321 - val_loss: 0.0348
Epoch 114/200
10/10            1s 68ms/step -
loss: 0.0307 - val_loss: 0.0344
Epoch 115/200
10/10            1s 68ms/step -
loss: 0.0303 - val_loss: 0.0359
Epoch 116/200
10/10            1s 68ms/step -
loss: 0.0314 - val_loss: 0.0377
Epoch 117/200
10/10            1s 69ms/step -
loss: 0.0306 - val_loss: 0.0355
Epoch 118/200
10/10            1s 73ms/step -
loss: 0.0311 - val_loss: 0.0349
Epoch 119/200
10/10            1s 68ms/step -
loss: 0.0286 - val_loss: 0.0337
Epoch 120/200
10/10            1s 69ms/step -
loss: 0.0289 - val_loss: 0.0346
Epoch 121/200
10/10            1s 68ms/step -
loss: 0.0294 - val_loss: 0.0343
Epoch 122/200
10/10            1s 68ms/step -
loss: 0.0286 - val_loss: 0.0313
```

```
Epoch 123/200
10/10            1s 70ms/step -
loss: 0.0279 - val_loss: 0.0308
Epoch 124/200
10/10            1s 77ms/step -
loss: 0.0271 - val_loss: 0.0309
Epoch 125/200
10/10            1s 69ms/step -
loss: 0.0259 - val_loss: 0.0299
Epoch 126/200
10/10            1s 75ms/step -
loss: 0.0253 - val_loss: 0.0311
Epoch 127/200
10/10            1s 69ms/step -
loss: 0.0251 - val_loss: 0.0300
Epoch 128/200
10/10            1s 68ms/step -
loss: 0.0221 - val_loss: 0.0329
Epoch 129/200
10/10            1s 73ms/step -
loss: 0.0211 - val_loss: 0.0279
Epoch 130/200
10/10            1s 67ms/step -
loss: 0.0214 - val_loss: 0.0297
Epoch 131/200
10/10            1s 69ms/step -
loss: 0.0224 - val_loss: 0.0275
Epoch 132/200
10/10            1s 68ms/step -
loss: 0.0221 - val_loss: 0.0268
Epoch 133/200
10/10            1s 79ms/step -
loss: 0.0189 - val_loss: 0.0272
Epoch 134/200
10/10            1s 69ms/step -
loss: 0.0189 - val_loss: 0.0248
Epoch 135/200
10/10            1s 68ms/step -
loss: 0.0187 - val_loss: 0.0231
Epoch 136/200
10/10            1s 69ms/step -
loss: 0.0170 - val_loss: 0.0255
Epoch 137/200
10/10            1s 69ms/step -
loss: 0.0158 - val_loss: 0.0226
Epoch 138/200
10/10            1s 69ms/step -
loss: 0.0160 - val_loss: 0.0224
```

```
Epoch 139/200
10/10              1s 68ms/step -
loss: 0.0160 - val_loss: 0.0240
Epoch 140/200
10/10              1s 68ms/step -
loss: 0.0150 - val_loss: 0.0238
Epoch 141/200
10/10              1s 75ms/step -
loss: 0.0181 - val_loss: 0.0284
Epoch 142/200
10/10              1s 68ms/step -
loss: 0.0182 - val_loss: 0.0292
Epoch 143/200
10/10              1s 87ms/step -
loss: 0.0165 - val_loss: 0.0256
Epoch 144/200
10/10              1s 79ms/step -
loss: 0.0146 - val_loss: 0.0272
Epoch 145/200
10/10              1s 68ms/step -
loss: 0.0148 - val_loss: 0.0219
Epoch 146/200
10/10              1s 68ms/step -
loss: 0.0149 - val_loss: 0.0256
Epoch 147/200
10/10              1s 68ms/step -
loss: 0.0129 - val_loss: 0.0224
Epoch 148/200
10/10              1s 80ms/step -
loss: 0.0132 - val_loss: 0.0246
Epoch 149/200
10/10              1s 69ms/step -
loss: 0.0133 - val_loss: 0.0251
Epoch 150/200
10/10              1s 81ms/step -
loss: 0.0122 - val_loss: 0.0209
Epoch 151/200
10/10              1s 80ms/step -
loss: 0.0124 - val_loss: 0.0237
Epoch 152/200
10/10              1s 69ms/step -
loss: 0.0128 - val_loss: 0.0222
Epoch 153/200
10/10              1s 68ms/step -
loss: 0.0124 - val_loss: 0.0282
Epoch 154/200
10/10              1s 68ms/step -
loss: 0.0122 - val_loss: 0.0214
```

```
Epoch 155/200
10/10            1s 68ms/step -
loss: 0.0125 - val_loss: 0.0246
Epoch 156/200
10/10            1s 75ms/step -
loss: 0.0113 - val_loss: 0.0218
Epoch 157/200
10/10            1s 68ms/step -
loss: 0.0108 - val_loss: 0.0233
Epoch 158/200
10/10            1s 68ms/step -
loss: 0.0112 - val_loss: 0.0212
Epoch 159/200
10/10            1s 69ms/step -
loss: 0.0150 - val_loss: 0.0247
Epoch 160/200
10/10            1s 68ms/step -
loss: 0.0136 - val_loss: 0.0235
Epoch 161/200
10/10            1s 67ms/step -
loss: 0.0108 - val_loss: 0.0247
Epoch 162/200
10/10            1s 74ms/step -
loss: 0.0116 - val_loss: 0.0237
Epoch 163/200
10/10            1s 69ms/step -
loss: 0.0111 - val_loss: 0.0217
Epoch 164/200
10/10            1s 69ms/step -
loss: 0.0111 - val_loss: 0.0273
Epoch 165/200
10/10            1s 69ms/step -
loss: 0.0109 - val_loss: 0.0233
Epoch 166/200
10/10            1s 74ms/step -
loss: 0.0106 - val_loss: 0.0194
Epoch 167/200
10/10            1s 70ms/step -
loss: 0.0093 - val_loss: 0.0222
Epoch 168/200
10/10            1s 70ms/step -
loss: 0.0096 - val_loss: 0.0197
Epoch 169/200
10/10            1s 83ms/step -
loss: 0.0109 - val_loss: 0.0223
Epoch 170/200
10/10            1s 74ms/step -
loss: 0.0102 - val_loss: 0.0228
```

```
Epoch 171/200
10/10            1s 72ms/step -
loss: 0.0102 - val_loss: 0.0214
Epoch 172/200
10/10            1s 71ms/step -
loss: 0.0100 - val_loss: 0.0218
Epoch 173/200
10/10            1s 65ms/step -
loss: 0.0100 - val_loss: 0.0286
Epoch 174/200
10/10            1s 74ms/step -
loss: 0.0118 - val_loss: 0.0249
Epoch 175/200
10/10            1s 69ms/step -
loss: 0.0103 - val_loss: 0.0204
Epoch 176/200
10/10            1s 68ms/step -
loss: 0.0102 - val_loss: 0.0257
Epoch 177/200
10/10            1s 67ms/step -
loss: 0.0119 - val_loss: 0.0260
Epoch 178/200
10/10            1s 69ms/step -
loss: 0.0109 - val_loss: 0.0254
Epoch 179/200
10/10            1s 69ms/step -
loss: 0.0118 - val_loss: 0.0242
Epoch 180/200
10/10            1s 70ms/step -
loss: 0.0100 - val_loss: 0.0202
Epoch 181/200
10/10            1s 69ms/step -
loss: 0.0119 - val_loss: 0.0254
Epoch 182/200
10/10            1s 70ms/step -
loss: 0.0127 - val_loss: 0.0240
Epoch 183/200
10/10            1s 71ms/step -
loss: 0.0112 - val_loss: 0.0213
Epoch 184/200
10/10            1s 70ms/step -
loss: 0.0098 - val_loss: 0.0231
Epoch 185/200
10/10            1s 67ms/step -
loss: 0.0109 - val_loss: 0.0223
Epoch 186/200
10/10            1s 67ms/step -
loss: 0.0099 - val_loss: 0.0213
```

```
Epoch 187/200
10/10              1s 72ms/step -
loss: 0.0093 - val_loss: 0.0225
Epoch 188/200
10/10              1s 68ms/step -
loss: 0.0087 - val_loss: 0.0205
Epoch 189/200
10/10              1s 75ms/step -
loss: 0.0093 - val_loss: 0.0208
Epoch 190/200
10/10              1s 66ms/step -
loss: 0.0083 - val_loss: 0.0202
Epoch 191/200
10/10              1s 73ms/step -
loss: 0.0092 - val_loss: 0.0204
Epoch 192/200
10/10              1s 69ms/step -
loss: 0.0096 - val_loss: 0.0244
Epoch 193/200
10/10              1s 68ms/step -
loss: 0.0096 - val_loss: 0.0201
Epoch 194/200
10/10              1s 70ms/step -
loss: 0.0081 - val_loss: 0.0218
Epoch 195/200
10/10              1s 69ms/step -
loss: 0.0082 - val_loss: 0.0193
Epoch 196/200
10/10              1s 69ms/step -
loss: 0.0091 - val_loss: 0.0209
Epoch 197/200
10/10              1s 67ms/step -
loss: 0.0080 - val_loss: 0.0204
Epoch 198/200
10/10              1s 67ms/step -
loss: 0.0085 - val_loss: 0.0191
Epoch 199/200
10/10              1s 69ms/step -
loss: 0.0082 - val_loss: 0.0211
Epoch 200/200
10/10              1s 69ms/step -
loss: 0.0086 - val_loss: 0.0211
```

# 6 Building and Training a Complex LSTM Model

This code snippet demonstrates how to define and train a complex LSTM (Long Short-Term Memory) model using Keras.

### 6.1 Importing Required Modules

- `from keras.models import Sequential`: Imports the `Sequential` class from Keras, which allows for building sequential neural network models.
- `from keras.layers import LSTM, Dense, Dropout`: Imports the LSTM, Dense, and Dropout layers, which are essential components of the neural network architecture.
- `from keras.callbacks import EarlyStopping`: Imports the EarlyStopping callback, which monitors the validation loss during training and stops training early if the loss stops decreasing.

### 6.2 Defining the Model Architecture

- `model = Sequential()`: Initializes a sequential model.
- `model.add(LSTM(100, return_sequences=True, input_shape=(time_step, X_train.shape[2])))`: Adds an LSTM layer with 100 units, returning sequences (required for subsequent LSTM layers), and specifies the input shape.
- `model.add(LSTM(100, return_sequences=False))`: Adds another LSTM layer with 100 units, but this time does not return sequences.
- `model.add(Dropout(0.2))`: Adds a dropout layer with a dropout rate of 0.2 to prevent overfitting.
- `model.add(Dense(50, activation='relu'))`: Adds a dense layer with 50 units and a ReLU activation function.
- `model.add(Dense(1))`: Adds a dense layer with a single unit, which is the output layer.

### 6.3 Compiling the Model

- `model.compile(optimizer='adam', loss='mean_squared_error')`: Compiles the model, specifying the Adam optimizer and mean squared error loss function.

### 6.4 Training the Model

- `early_stopping = EarlyStopping(monitor='val_loss', patience=40, restore_best_weights=True)`: Initializes an EarlyStopping callback to monitor the validation loss and stop training if it does not improve after 40 epochs.
- `history = model.fit(X_train, Y_train, epochs=200, batch_size=64, validation_data=(X_test, Y_test), callbacks=[early_stopping], verbose=1)`: Trains the model using the training data (`X_train` and `Y_train`) for 200 epochs with a batch size of 64. The validation data (`X_test` and `Y_test`) is used to monitor the model's performance during training, and the EarlyStopping callback is applied to prevent overfitting.

This code snippet illustrates the process of building and training a complex LSTM model for time-series forecasting tasks.

[ ]:
```
# Make predictions
predictions = model.predict(X_test)
predictions = scaler.inverse_transform(predictions)

# Evaluate the model
train_predict = model.predict(X_train)
test_predict = model.predict(X_test)
```

```python
# Transform back to original scale
train_predict = scaler.inverse_transform(train_predict)
test_predict = scaler.inverse_transform(test_predict)


# Calculate RMSE for training and testing sets
train_score = np.sqrt(np.mean((train_predict - scaler.inverse_transform(Y_train.
    ↪reshape(-1, 1)))**2))
test_score = np.sqrt(np.mean((test_predict - scaler.inverse_transform(Y_test.
    ↪reshape(-1, 1)))**2))

print(f'Train RMSE: {train_score:.2f}')
print(f'Test RMSE: {test_score:.2f}')

# Check the lengths for debugging
print(f"Length of scaled_sales: {len(scaled_sales)}")
print(f"Length of train_predict: {len(train_predict)}")
print(f"Length of test_predict: {len(test_predict)}")
```

```
9/9              0s 29ms/step
20/20              0s 12ms/step
9/9              0s 11ms/step
Train RMSE: 1384679.14
Test RMSE: 2145640.40
Length of scaled_sales: 942
Length of train_predict: 616
Length of test_predict: 265
```

# 7 Making Predictions and Evaluating the Model

This code snippet demonstrates how to make predictions using a trained LSTM model and evaluate its performance.

## 7.1 Making Predictions

- `predictions = model.predict(X_test)`: Uses the trained LSTM model to make predictions on the test dataset (`X_test`).
- `predictions = scaler.inverse_transform(predictions)`: Inverse transforms the scaled predictions back to the original scale using the MinMaxScaler (`scaler`).

## 7.2 Evaluating the Model

- `train_predict = model.predict(X_train)` and `test_predict = model.predict(X_test)`: Makes predictions on both the training and test datasets to evaluate the model's performance.
- `train_predict = scaler.inverse_transform(train_predict)` and `test_predict = scaler.inverse_transform(test_predict)`: Inverse transforms the scaled predictions on

both training and test datasets back to the original scale.

### 7.3 Calculating RMSE (Root Mean Squared Error)

- `train_score = np.sqrt(np.mean((train_predict - scaler.inverse_transform(Y_train.reshape(-1, 1)))**2))` and `test_score = np.sqrt(np.mean((test_predict - scaler.inverse_transform(Y_test.reshape(-1, 1)))**2))`: Calculates the RMSE for both the training and testing sets. It compares the predicted sales values (`train_predict` and `test_predict`) with the actual sales values (`Y_train` and `Y_test`) after inverse transformation.
- The RMSE measures the average magnitude of the errors between predicted and actual values, with lower values indicating better model performance.

### 7.4 Debugging Information

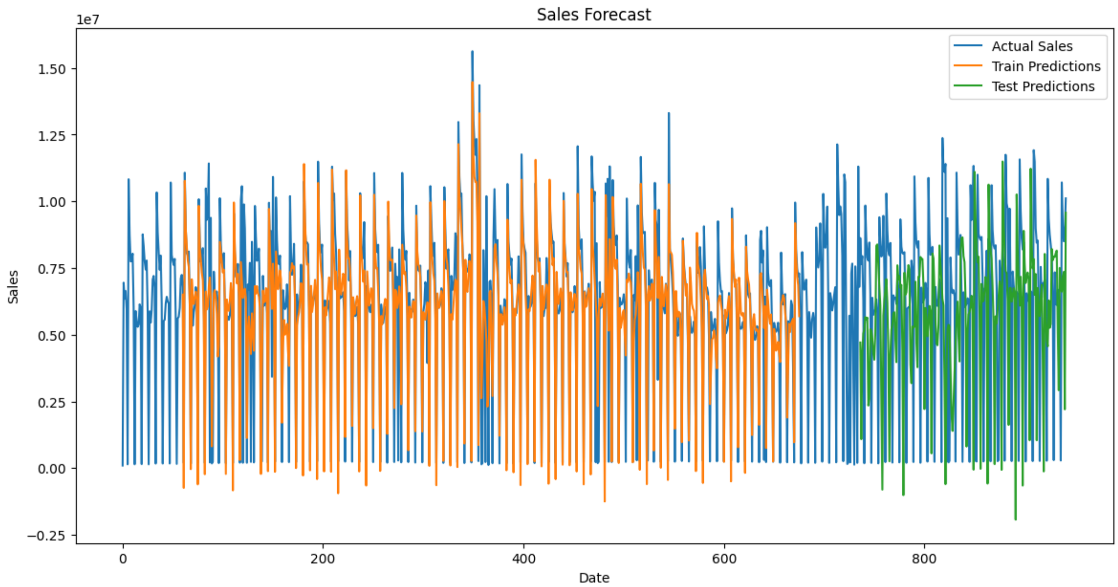
- `print(f"Length of scaled_sales: {len(scaled_sales)}")`: Prints the length of the scaled sales data, providing information about the size of the dataset used for training the model.
- `print(f"Length of train_predict: {len(train_predict)}")` and `print(f"Length of test_predict: {len(test_predict)}")`: Prints the lengths of the predicted sales values on the training and test datasets, which can be useful for debugging purposes.

This code snippet completes the process of making predictions, evaluating the model's performance, and providing debugging information to assess the effectiveness of the LSTM model for time-series forecasting.

[ ]:
```
# Create empty arrays for plotting predictions
train_plot = np.empty_like(scaled_sales)
train_plot[:, :] = np.nan
train_plot[time_step:len(train_predict) + time_step, :] = train_predict

test_plot = np.empty_like(scaled_sales)
test_plot[:, :] = np.nan

# Correct indexing for test predictions
# Align test predictions after the training predictions
test_start_index = len(train_predict) + (time_step * 2)
test_end_index = test_start_index + len(test_predict)

# Ensure indices are valid for the length of test_plot
if test_end_index <= len(test_plot):
    test_plot[test_start_index:test_end_index, :] = test_predict
else:
    # Adjust if the calculated index is out of bounds
    valid_length = len(test_plot) - test_start_index
    print(f"Adjusting test prediction length from {len(test_predict)} to␣
 ↪{valid_length}")
    test_plot[test_start_index:test_start_index + valid_length, :] =␣
 ↪test_predict[:valid_length]
```

```
# Plot the results
plt.figure(figsize=(14, 7))
plt.plot(scaler.inverse_transform(scaled_sales), label='Actual Sales')
plt.plot(train_plot, label='Train Predictions')
plt.plot(test_plot, label='Test Predictions')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.title('Sales Forecast')
plt.legend()
plt.show()
```

```
Adjusting test prediction length from 265 to 206
```




# 8 Plotting Predictions

This code snippet plots the actual sales data along with the predictions made by the LSTM model on both the training and test datasets.

## 8.1 Creating Empty Arrays for Plotting Predictions

- `train_plot = np.empty_like(scaled_sales)`: Creates an empty numpy array (`train_plot`) with the same shape as the scaled sales data.
- `train_plot[:, :] = np.nan`: Fills the `train_plot` array with NaN (Not a Number) values to represent missing values.
- `train_plot[time_step:len(train_predict) + time_step, :] = train_predict`: Inserts the predictions made on the training dataset (`train_predict`) into the `train_plot`

array. The predictions start from the time step and continue for the length of the training predictions.

- `test_plot = np.empty_like(scaled_sales)`: Creates another empty numpy array (`test_plot`) with the same shape as the scaled sales data.
- `test_plot[:, :] = np.nan`: Fills the `test_plot` array with NaN values.

### 8.2 Correcting Indexing for Test Predictions

- Calculates the start and end indices for inserting the test predictions into the `test_plot` array, ensuring that they align correctly with the training predictions.
- Checks if the calculated end index is within the valid range of `test_plot`. If it exceeds the length of `test_plot`, it adjusts the length of the test predictions accordingly.

### 8.3 Plotting the Results

- `plt.figure(figsize=(14, 7))`: Initializes a matplotlib figure with a specified size for plotting.
- `plt.plot(scaler.inverse_transform(scaled_sales), label='Actual Sales')`: Plots the actual sales data after inverse transformation to the original scale using the MinMaxScaler.
- `plt.plot(train_plot, label='Train Predictions')`: Plots the predictions made on the training dataset (`train_plot`).
- `plt.plot(test_plot, label='Test Predictions')`: Plots the predictions made on the test dataset (`test_plot`).
- `plt.xlabel('Date')` and `plt.ylabel('Sales')`: Labels the x-axis and y-axis of the plot, respectively.
- `plt.title('Sales Forecast')`: Sets the title of the plot.
- `plt.legend()`: Displays the legend on the plot to distinguish between actual sales and predictions.
- `plt.show()`: Shows the plot.

This code snippet visualizes the actual sales data along with the predictions made by the LSTM model, providing insights into the model's performance in forecasting sales.