ML Models

Dalton Bailey

2023-05-14

LinearRegressor

The `LinearRegressor` class in this Python code is a machine learning model designed to perform regression tasks. It's built using the Sequential API of Keras, a popular deep learning library in Python. The model is initialized with no layers, and these are added in the `train` method.

In the `train` method, the data is preprocessed using a `preProcess` function, which is not provided in this code. The target variables (`Close` and `Adj Close`) are separated from the feature variables. A MinMaxScaler is used to normalize both the feature and target data. This scaling is necessary because the range of values of raw data varies widely, and scaling ensures that the input features have the same scale, improving the stability and performance of the model.

The model architecture consists of three Dense layers with 128, 64, and 32 neurons respectively, and 'relu' activation functions. The output layer has a single neuron, as is common for regression tasks. The model uses the 'adam' optimizer and mean squared error as the loss function, which is typical for regression problems.

The model is trained for 100 epochs, with a verbosity level of 2, which means that the model will output detailed information about the training process, including the loss and any metrics specified during the compilation of the model.

After training, the model makes predictions on the test data, and these predictions are inversely transformed to bring them back to the original scale of the target variable. The model then calculates and prints the Mean Squared Error, Mean Absolute Error, and the R^2 score of these predictions. These metrics provide a quantitative measure of the model's performance.

Finally, the model is saved to a .h5 file, a common format for storing Keras models. This allows the model to be loaded and used later without needing to be retrained.

In the `predict` method, the model is loaded from the .h5 file, and it makes predictions on new data. The new data is preprocessed in the same way as the training data, and the predictions are inversely transformed before being returned.

RNN Model

The `RNN` class in this Python code represents a recurrent neural network (RNN) model. This model is built using Keras, a popular deep learning library in Python. Like the `LinearRegressor` class, this class is initialized with an empty Sequential model.

The `train` method begins with the same preprocessing steps as in `LinearRegressor`, with the data being passed through a `preProcess` function and the features and target variable being separated and scaled using a MinMaxScaler.

A significant difference in the `train` method of `RNN` is the introduction of a `look_back` parameter. This parameter determines the number of previous time steps to use as input features for the model. It is set to 10 in this code, meaning that the model will use the values from the 10 previous time steps to make a

prediction for the current time step. This is a common technique in time series prediction tasks, where the value at a given time is often highly dependent on recent values.

The architecture of the model consists of three SimpleRNN layers with 50 units each, with Dropout layers in between to prevent overfitting. The output layer is a Dense layer with a single neuron, as this is a regression task. The model is compiled with the 'adam' optimizer and the mean squared error loss function, and it is trained for 100 epochs.

After training, the model makes predictions on the test data and these predictions are inversely transformed to bring them back to the original scale of the target variable. The model then calculates and prints the Mean Squared Error, Mean Absolute Error, and the R^2 score of these predictions. These metrics provide a quantitative measure of the model's performance.

The model is then saved to a .h5 file, allowing it to be loaded and used later without needing to be retrained.

In the `predict` method, the model is loaded from the .h5 file and used to make predictions on new data. The new data is preprocessed in the same way as the training data, and the predictions are inversely transformed before being returned.

The `create_dataset` method is a utility function that is used to reshape the input data into the appropriate format for the RNN model. It takes the dataset and a look_back parameter and returns two numpy arrays, X and Y, which represent the input features and the target variable, respectively.

LTSM Model

The `LSTMModel` class in this Python code represents a Long Short-Term Memory (LSTM) model. This model is also built using Keras. Like the `RNN` class, this class is initialized with an empty Sequential model.

The `train` method follows similar preprocessing steps as in the `RNN` and `LinearRegressor` classes. The data is passed through a `preProcess` function, and the features and target variable are separated and scaled using a MinMaxScaler.

The `look_back` parameter is also used in this class. This parameter determines the number of previous time steps to use as input features for the LSTM model. In this code, it is set to 10, implying that the model will use the values from the 10 previous time steps to predict the current time step.

The architecture of the LSTM model consists of two LSTM layers with 50 units each, with Dropout layers in between to prevent overfitting. The output layer is a Dense layer with a single neuron, indicating a regression task. The model is compiled with the 'adam' optimizer and the mean squared error loss function, and it is trained for 100 epochs.

After training, the model makes predictions on the test data. These predictions are inversely transformed to bring them back to the original scale of the target variable. The model then calculates and prints the Mean Squared Error, Mean Absolute Error, and the R^2 score of these predictions. These metrics provide a quantitative measure of the model's performance.

The model is then saved to a .h5 file, enabling it to be loaded and used later without needing to be retrained.

In the `predict` method, the model is loaded from the .h5 file and used to make predictions on new data. The new data is preprocessed in the same way as the training data, and the predictions are inversely transformed before being returned.

The `create_dataset` method is a utility function that is used to reshape the input data into the appropriate format for the LSTM model. It takes the dataset and a look_back parameter and returns two numpy arrays, X and Y, which represent the input features and the target variable, respectively.

HMM Model

This Python class, `HMM`, represents a Hidden Markov Model (HMM). HMMs are statistical models that are used when our observations are believed to be driven by an underlying process that we cannot directly observe (the hidden part). This particular HMM is being trained to model the daily closing prices of a stock.

The `__init__` method initializes the class with an empty model.

The `train` method preprocesses the input data using the `preProcess` function, similar to the previous classes. It then trains a GaussianHMM with three hidden states, "full" covariance type, and 1000 iterations using the closing prices. GaussianHMM is a type of HMM with Gaussian emissions, which means it assumes the data (in this case, closing prices) is normally distributed. The trained model is saved to a .pkl file using `joblib.dump`.

The `hiddenStates` method loads the trained model from the .pkl file and predicts the hidden states of the closing prices. These hidden states represent the underlying process that we believe is driving the changes in closing prices.

The `predict` method also loads the trained model and uses it to predict the closing prices for a specified number of future days. It does this by starting with the last observed closing price and predicting the next state and emission (closing price) for each future day. The predicted closing prices are returned as a numpy array.

The `print_details` method prints out the details of the trained model. This includes the transition matrix (which shows the probabilities of transitioning between each pair of states) and the mean and covariance of each state (which represent the parameters of the Gaussian distributions for each state's emissions). If the model hasn't been trained yet, it instead prints a message to that effect.

ARIMA Model

This Python class, `ARIMAModel`, represents an AutoRegressive Integrated Moving Average (ARIMA) model. ARIMA models are a popular choice for time series forecasting.

The `__init__` method initializes the class and sets the order of the ARIMA model, which represents the (p,d,q) parameters where:

- p is the number of autoregressive terms,
- d is the number of nonseasonal differences needed for stationarity, and
- q is the number of lagged forecast errors in the prediction equation.

The `train` method preprocesses the input data using the `preProcess` function, similar to the previous classes. It then splits the data into training and testing sets. The ARIMA model is then fit on the training data.

The fit model is used to predict values for the testing data. The model's performance is evaluated by calculating the root mean square error (RMSE) between the actual and predicted values.

The `predict` method uses the fit model to make predictions on new input data. It starts by ensuring that the input `X` drops the 'Close' and 'Adj Close' columns. It then selects columns that are not of 'object' data type and makes predictions on this processed data. These predictions are then returned by the method.