

Forecasting in Time-Series Data: An implementation in Python

Prahald Siwakoti

March 17, 2025

Time series data are data from observations that are collected over time [1]. These kinds of data are prevalent in various fields such as economics, finance, weather forecasting, and many others. One of the main characteristics of time series data is the serial dependence of the observations. This is in violation of the assumption of independence that many standard statistical methods are built on. Therefore, specialized methods are required to analyze and forecast time series data.

The objectives of time series analysis commonly include Inference, Forecasting, and Smoothing. Inference aims to understand the underlying process that generated the data. Forecasting involves predicting future values of the time series based on historical observations. Smoothing is used to extract the underlying trend from the data and to fill in missing values.

In this document, we will discuss forecasting from time-series data and explore some models to achieve this:

- Time Series models: These models leverages the serial dependence of the data for predictions. We will implement the Seasonal ARIMA model with exogenous variables. See Appendix 6.1 for a brief review of the ARIMA model.
- Decision-tree-based regression models: These models extrapolate trends to forecast future values. We will implement the XgBoost [2] model, which is a gradient boosted decision tree algorithm known for its efficiency and accuracy.
- Neural network based models: Particularly Long Short-Term Memory (LSTM) networks, a variant of Recurrent Neural Networks (RNN) that are designed to capture long-term dependencies in the data. See Appendix 6.3 for more details.

In practice, the choice of forecasting method depends on the data and the goals of the analysis. Often times, a combination of methods is used to improve the accuracy of the forecast.

Using a simple time series revenue example data from prophet, we will implement the above mentioned models in python. We will compare the preformance between these models using performance metrics. We will then visualize the forecasted values with the help of Python libraries such as matplotlib and seaborn. The complete code is available at GitHub.

1 Data preparation

The data is loaded using the following code:

```
import pandas as pd
url = 'https://github.com/facebook/prophet/blob/main/examples/\example_retail_sales.csv?raw=true'
data = pd.read_csv(url)
```

The data consists of two columns: 'ds' and 'y'. The 'ds' column contains the date of the observation, and the 'y' column contains the revenue for that date. The data is in a long format, which is suitable for time series analysis.

In the next few sections, we will discuss preprocessing steps for each of the three models we used for forecasting: ARIMA, XGBoost, and LSTM. For consistency of comparison, we will use the same splitting method for training and testing sets (split at 80% of the length of the data without shuffling the data (to preserve time dependence)). The training set will be used to fit the model, and the testing set will be used to evaluate the performance of the model.

1.1 ARIMA Model

For ARIMA models, as the data is already in the long format, we do not need to do much preprocessing. However, we need to convert the 'ds' column to a datetime format. This can be done using the following code:

```
data['ds'] = pd.to_datetime(data['ds'])
# Convert the 'ds' column to datetime
```

1.2 XgBoost Model

For the regression model, we will create some new features in addition to the revenue values. We will create two lag features: 'lag1' and 'lag2', which are the revenue values from the previous day and the day before that respectively. We will also create features for the month and the year, which are treated as categorical variables.

The 'lag' features are obtained using the handy `shift()` method in pandas. The following code implements this feature engineering:

```
data['lag1'] = data['y'].shift(1)
data['lag2'] = data['y'].shift(2)
data['month'] = data['ds'].dt.month
data['year'] = data['ds'].dt.year
```

2 LSTM Model

The Data preprocessing for LSTM is a little involved. First, we want to generate some features from the data. I chose to use month variable from the date column. Unlike the decision tree-based models, LSTM models process data in a sequential manner. This means that it is important for us to feed the data in proper order. Since, the model cannot recognize months 1-12 as a complete cycle, we need to convert the month variable into a sine and cosine function. This is done using the following code:

```
import numpy as np
data['month_sin'] = np.sin(2 * np.pi * data['month'] / 12)
data['month_cos'] = np.cos(2 * np.pi * data['month'] / 12)
```

If more features are available, the implementation can be easily extended to add more engineered features. This often results in improved accuracy of the model.

Next, a linear regression is fitted to the data to obtain the trend. The linear trend is then subtracted from the data to obtain the residuals. The residuals are then scaled using the MinMaxScaler from sklearn. This is done to ensure that the data is in the range [0, 1] before feeding it to the model.

Then, we need a method to feed this data sequentially to the model. This is done using a sliding window approach. That is, the data from time $t-w$ to time t , is used to predict the next time step ($t+1$). The size of window, w , governs how much data you are allowed to look at when you make the prediction. This is also called the look back period. The following code implements the sliding window approach:

```
def create_dataset(dataset, lookback):
    X, y = [], []
    for i in range(len(dataset)-lookback):
        feature = dataset[i:i+lookback]
        target = dataset[i+lookback]
        X.append(feature)
        y.append(target)
    return torch.tensor(X), torch.tensor(y)
```

Now, we use the DataLoader class from PyTorch to create batches of data from the dataset.

3 Model Implementation

In this section, we will discuss each model in detail and show their implementation in Python.

3.1 ARIMA Model

Some exploratory data analysis is done to determine the order of the ARIMA model. A differencing of order 1 is sufficient to make the data stationary as supported by the Dickey-Fuller test. The ACF and PACF plots, not shown here, are used to determine the order of the AR and MA terms for the differenced data. ACF plot shows signs of seasonality at lag 12 and cutoff at lag1, which suggests a seasonal AR term as well as an AR(1) term might be appropriate. The PACF plot shows gradual tailing off over the first few lags, with occasional spikes.

To make a proper assessment, we employed auto_arima method from pmdarima library.

- ARMA(1,1,2) is found to be best model. i.e. AR(1) and MA(2) with first order differencing.
- Seasonal differencing (D=1) with two seasonal AR terms (SAR(2)) and one seasonal MA term (SMA(1)) for the 12-month seasonality

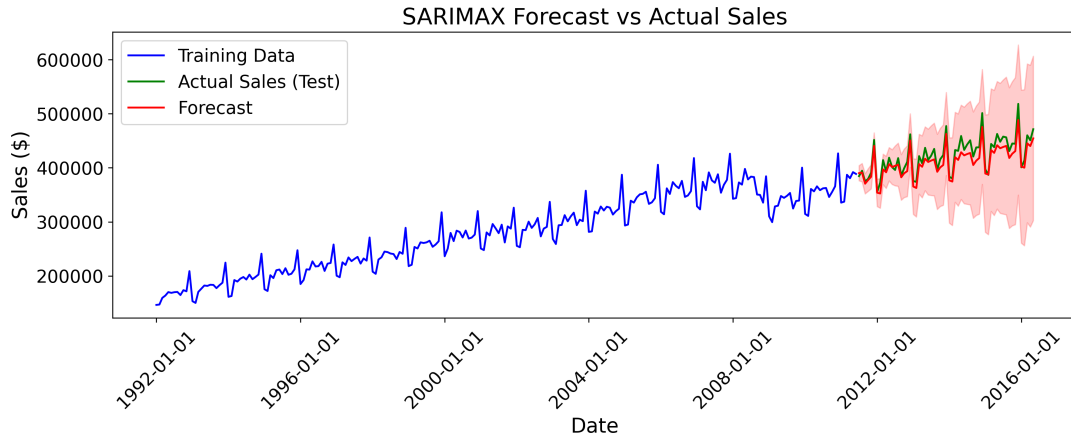


Figure 1: ARIMA Model Forecast

The following code implements the ARIMA model:

```
# Define the SARIMAX model
import statsmodels.api as sm
model = sm.tsa.SARIMAX(
    train['revenue'], # Use revenue data
    order=(1,1,2), # (p,d,q)
    seasonal_order=(2,1,1,12), # (P,D,Q,S) for 12-month seasonality
    enforce_stationarity=False,
    enforce_invertibility=False
)

# Fit the model
results = model.fit()

# Forecast
forecast_steps = len(test) # Forecast same number of steps as test set
forecast = results.get_forecast(steps=forecast_steps)
forecast_index = test.index

# Extract prediction and confidence intervals
forecast_mean = forecast.predicted_mean
forecast_ci = forecast.conf_int()
```

Figure 1 shows the forecasted values along with the confidence intervals. The forecasted values are in red, the actual test values are shown in green. The training data is shown in blue and also shown are the confidence intervals of the forecasted values. The model does very well in capturing the trend and seasonality of the data. The mean absolute error (MAE), defined as the average of the absolute differences between the forecasted and actual values, is 12,417.14 USD. The mean absolute percentage error (MAPE), defined as the average of the absolute percentage differences between the forecasted and actual values, is 2.86%.

3.2 XgBoost Model

Tree-based models, like Random Forests and XGBoost, split the data into subsets based on the values of the predictors. The goal in splitting, for regression problems, is to minimize the mean squared error. Each leaf node then makes a prediction based on the average of the observations in that node. These models are powerful and can capture complex relationships in the data.

Care must be taken when using tree-based models for time series forecasting. The data must be transformed into a format that the model can understand. This usually involves de-trending, removing seasonality, creating lagged variables etc. The model can then be trained on this data to make predictions.

```
# Fit a Linear Trend Model
dt['time_index'] = np.arange(len(dt)) # Create a time index for linear regression
trend_model = LinearRegression()
trend_model.fit(dt[['time_index']], dt['sales']) # Fit trend model
dt['trend'] = trend_model.predict(dt[['time_index']]) # Extract trend component

# Detrend the Data (Sales - Trend)
dt['sales_detrended'] = dt['sales'] - dt['trend']
# train - test split
test_period_index = int(len(dt) * 0.8)
train_data = dt.iloc[:test_period_index]
test_data = dt.iloc[test_period_index:]

X_train = train_data.drop(columns=['sales', 'sales_detrended', 'trend', 'time_index'])
y_train = train_data['sales_detrended'] # Train on detrended sales
X_test = test_data.drop(columns=['sales', 'sales_detrended', 'trend', 'time_index'])
y_test = test_data['sales_detrended']

# Fit the XGBoost model
model = XGBRegressor()
model.fit(X_train, y_train)

# Make predictions
y_train_pred_detrended = model.predict(X_train)
y_test_pred_detrended = model.predict(X_test)

y_train_pred = y_train_pred_detrended + train_data['trend'].values
y_test_pred = y_test_pred_detrended + test_data['trend'].values
```

Figure 2 shows the forecasted values along with the actual test values. The forecasted values are in red, the actual test values are shown in green. The predicted training data is shown in blue.

The model looks like it might be overfitting to the training data. Addition of exogenous variables might help remedy this. Regardless, the model is doing pretty well in capturing the trend and seasonality of the data. The mean absolute error (MAE) is 12321.65 USD, and the mean absolute percentage error (MAPE) is 2.85%.

One of the advantages of tree-based models like XgBoost is their ability to provide

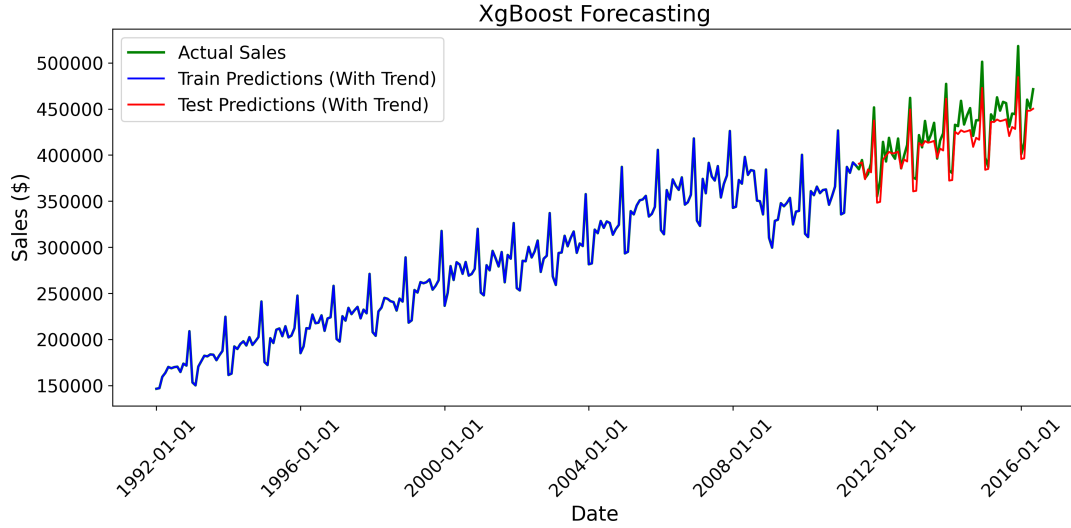


Figure 2: XgBoost Model Forecast

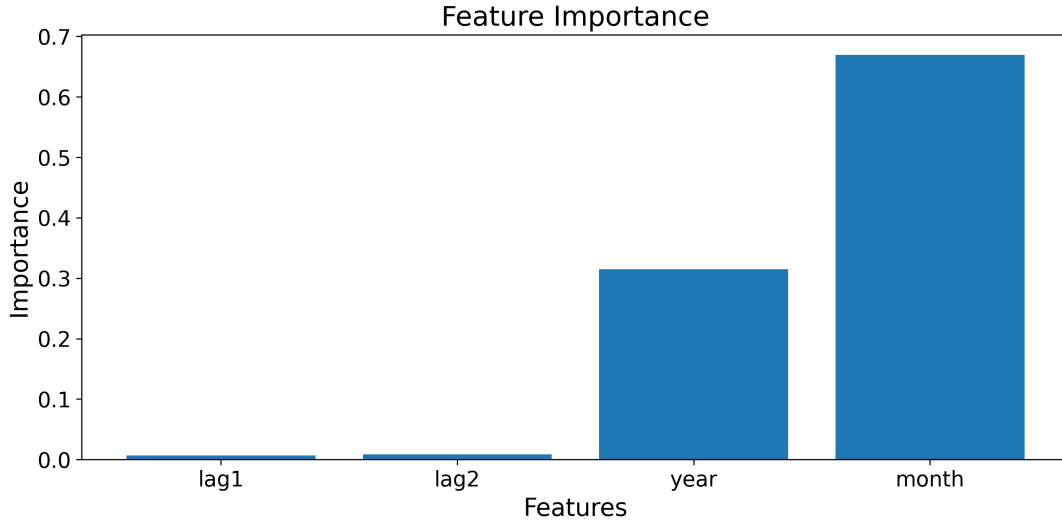


Figure 3: Feature Importance in XgBoost Model

feature importance scores. These scores indicate how much each feature contributes to the predictive power of the model.

Figure 3 shows the feature importance plot for the XgBoost model. 'month' and 'year' are the most important features, followed by 'lag2' and 'lag1'. This suggests that the month and year have a significant impact on the revenue. The lag features are also important, indicating that the revenue values from the previous days are useful in predicting future revenue but considerably less important than the month and year features.

3.3 LSTM Model

The LSTM model is implemented using the PyTorch library. The model consists of an bidirectional doubly stacked LSTM layer followed by a fully connected linear layer. The model is trained using the AdamW optimizer and the Mean Squared Error (MSE) loss

function. The model is trained for 500 epochs with a batch size of 4. The following code implements the LSTM model:

```
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.lstm = nn.LSTM(input_size=3, hidden_size=50, num_layers=2, \
batch_first=True, dropout=0.2, bidirectional=True)
        self.linear = nn.Linear(50*2, 3)

    def forward(self, x):
        x, _ = self.lstm(x)
        x = self.linear(x)
        return x

# train the model
model = Model()
optimizer = optim.AdamW(model.parameters(), lr=0.0001)
loss_fn = nn.MSELoss()

loader = data.DataLoader(data.TensorDataset(X_train, y_train), \
shuffle=False, batch_size=4)

n_epochs = 500
for epoch in range(n_epochs):
    model.train()
    for X_batch, y_batch in loader:
        X_batch = X_batch.float()
        y_batch = y_batch.float()
        y_pred = model(X_batch)
        y_pred = y_pred[:, -1, :] # Take only the last time step output
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    # Validation
    if epoch % 50 != 0:
        continue
    model.eval()
    with torch.no_grad():
        X_train = X_train.float()
        X_test = X_test.float()
        y_pred = model(X_train)
        y_pred = y_pred[:, -1, :] # Take only the last time step output
        train_rmse = np.sqrt(loss_fn(y_pred, y_train).item())
        y_pred = model(X_test)
        y_pred = y_pred[:, -1, :] # Take only the last time step output
        test_rmse = np.sqrt(loss_fn(y_pred, y_test).item())
    print("Epoch %d: train RMSE %.4f, test RMSE %.4f" % \
```

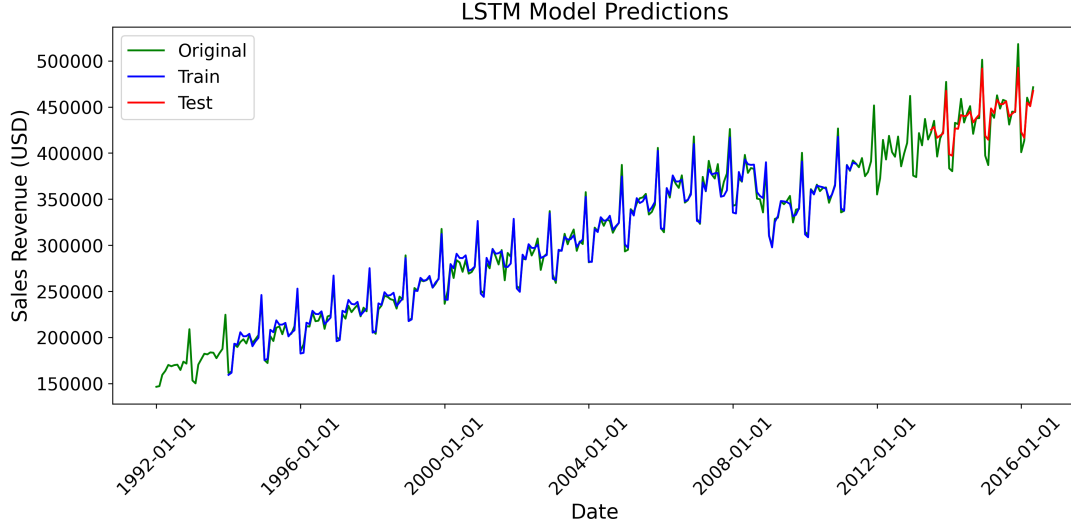


Figure 4: LSTM Model Forecast

(epoch, train_rmse, test_rmse))

Figure 4 shows the forecasted values along with the actual test values. The forecasted values are in red, the actual test values are shown in green. The predicted training data is shown in blue. The model does a good job of capturing the trend and seasonality of the data. Unlike the XgBoost model, the LSTM model does not seem to be overfitting to the training data. The mean absolute error (MAE) on test data is 8322.45 USD, and the mean absolute percentage error (MAPE) is 1.95%.

4 Model Comparison

Table 1 shows the performance of the three models on the test data. The LSTM model has the lowest MAE and MAPE, indicating that it is the best performing model. The XgBoost model and the Seasonal ARIMA models are close in performance.

I should note that for the LSTM model, there is a window where no prediction is obtained as the model works with a sliding window approach. This is why the LSTM model has fewer predictions than the other models. This might lower the MAE and MAPE values as the model is not penalized for these missing values. However, the LSTM model is still the best performing model in this comparison.

| Model | MAE in USD | MAPE (%) |
|---------|------------|----------|
| SARIMA | 12417.14 | 2.86% |
| XgBoost | 12321.65 | 2.85% |
| LSTM | 8322.45 | 1.95% |

Table 1: Model Comparison

5 Conclusion

In this document, we have discussed three methods for forecasting time series data: ARIMA, XGBoost, and LSTM. We have implemented these methods using Python and

compared their performance using the Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE) as evaluation metrics. The results show that all three methods can be used for time series forecasting, but the choice of method depends on the data and the goals of the analysis.

Interpretability is an important aspect of time series forecasting models, as it allows us to understand how the model makes predictions and to trust the results. Different models offer varying levels of interpretability.

Statistical models, such as ARIMA, are generally more interpretable compared to machine learning models. These models have clearly defined parameters that can be directly understood in terms of their influence on the forecast. On the other hand, tree-based methods and neural networks are often considered "black-box" models due to their complexity. However, techniques like feature importance, partial dependence plots, and SHAP (SHapley Additive exPlanations) values can enhance their interpretability. Among these, neural networks are the least interpretable. While methods exist to highlight significant features and break down predictions into contributions from each input feature, careful implementation is required, especially when interpretability is a critical requirement.

In conclusion, while more complex models may offer better predictive performance, they often come at the cost of interpretability. It is important to balance the need for accurate forecasts with the need for understanding and trust in the model's predictions.

6 Appendix

6.1 Time Series Analysis (Overview)

Time series Models are statistical models that are used to analyze and forecast time series data. They are based on the assumption that the data is generated by a stochastic process, which means that the future values of the time series depend on its past values.

A general approach is to identify if the time series data is stationary. A stationary time series is one whose statistical properties such as mean, variance, and autocorrelation do not change over time. This is important because many time series models assume stationarity.

If the data are stationary, one needs to verify serial dependence. This is usually done by examining the autocorrelation function (ACF) and partial autocorrelation function (PACF) plots. These plots help identify the order of the autoregressive (AR) and moving average (MA) terms in the model.

If the data are not stationary, one can try to coerce data into stationarity. This can be done in one of three ways: differencing, transformation, or modeling the trend. Differencing involves taking the difference between consecutive observations. Transformation involves applying a function to the data to stabilize the variance. Modeling the trend involves fitting a model to the trend and then modeling the residuals.

There are several time series models that can be used for forecasting for stationary data. We will discuss some of the most common ones:

6.1.1 Autoregressive Model

The autoregressive (AR) model assumes that the current value of the time series is a linear combination of the previous values. The AR model is denoted as AR(p), where p

is the order of the model. The model can be written as:

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + w_t$$

where x_t is the value of the time series at time t , ϕ_i are the parameters of the model, and w_t is the white noise term. The parameters to estimate are the ϕ_i values and σ_w^2 , the variance of the white noise term. The requirement that x_t is stationary imposes constraints on the values of the ϕ_i parameters. The model can be estimated using the method of least squares or maximum likelihood estimation.

6.1.2 Moving Average Model

A moving average model MA(q) assumes that the current value of the time series is a linear combination of the previous forecast errors. The model can be written as:

$$x_t = w_t + \theta_1 w_{t-1} + \theta_2 w_{t-2} + \cdots + \theta_q w_{t-q}$$

where w_t is the white noise term, θ_i are the parameters of the model, and q is the order of the model. The MA model is stationary for all values of θ_i .

6.1.3 Autoregressive Moving Average (ARMA) Model

The ARMA model combines the AR and MA models. The ARMA(p, q) model is given by:

$$x_t = \alpha + \sum_{i=1}^p \phi_i x_{t-i} + \sum_{j=1}^q \theta_j w_{t-j} + w_t$$

for $\phi_i > 0$, $\theta_j > 0$ and $\sigma_w^2 > 0$ and the model is causal and invertible. The ARMA model can be estimated using the method of least squares or maximum likelihood estimation.

Choosing the Order of these Models

There are two steps to choosing the order of the AR and MA models.

Exploratory Data Analysis Prior to fitting models, the data can be analyzed using ACF and PACF plots[refer to Appendix 6.4]. A general behavior of the ACF and PACF plots for AR and MA models is as follows:

| Model | ACF | PACF |
|----------------|------------------------|------------------------|
| AR(p) | Tails off | Cuts off after lag p |
| MA(q) | Cuts off after lag q | Tails off |
| ARMA(p, q) | Tails off | Tails off |

Model Selection The order of the AR and MA models can be selected using information criteria such as AIC, BIC, or cross-validation. These criteria balance the goodness of fit of the model with the complexity of the model. The model with the lowest information criterion is selected.

6.1.4 Autoregressive Integrated Moving Average (ARIMA) Model

Backshift Operator (B) The backshift operator B is defined as:

$$Bx_t = x_{t-1}$$

So, In general, $B^k x_t = x_{t-k}$. The differencing operator ∇ can be written as:

$$\begin{aligned}\nabla x_t &= x_t - x_{t-1} \\ \nabla^2 x_t &= (1 - B)x_t = x_t - 2x_{t-1} + x_{t-2}\end{aligned}$$

For differencing of order d : $\nabla^d x_t = (1 - B)^d x_t$

ARIMA Model The ARIMA model is a generalization of the ARMA model that can handle non-stationary data. The ARIMA(p, d, q) model is given by:

$$\phi(B)\nabla^d x_t = \alpha + \theta(B)w_t$$

where $\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \dots - \phi_p B^p$ is the autoregressive operator, $\theta(B) = 1 + \theta_1 B + \theta_2 B^2 + \dots + \theta_q B^q$ is the moving average operator, and $\alpha = \delta(1 - \phi_1 - \phi_2 - \dots - \phi_p)$ and $\delta = E[\nabla^d x_t]$.

Fitting and ARIMA(p, d, q) model is equivalent to fitting an ARMA(p, q) model to the data that is differenced to the order d .

6.2 Regression based Methods

A regular regression model is of the form:

$$y_t = \beta_0 + \beta_1 x_{1,t} + \beta_2 x_{2,t} + \dots + \beta_k x_{k,t} + \epsilon_t$$

where y_t is a linear function of the k - predictor variables $(x_{1,t}, x_{2,t}, \dots, x_{k,t})$, and ϵ_t is white noise. This model, while allows for the inclusion of a lot of relevant information in the form of predictor variables, does not account for the serial dependence like for instance ARIMA models do. A workaround is to include lagged values of the dependent variable as predictors. This is known as autoregressive distributed lag (ADL) models.

Yet another approach is to allow the errors from a regression model to contain autocorrelation. This is often referred to as **dynamic regression models**. To frame this problem, we will assume that the residuals follow ARIMA(p, d, q) process. The model can be written as:

$$y_t = \beta_0 + \beta_1 x_{1,t} + \beta_2 x_{2,t} + \dots + \beta_k x_{k,t} + \eta_t$$

$$(1 - B)^d \phi(B)\eta_t = \theta(B)\epsilon_t$$

where ϵ_t is the white noise term. The error from the regression model (η_t) is modeled with ARIMA process.

6.3 Neural Networks

6.3.1 A brief introduction to Neural Networks

Standard linear regression assumes that the true models are a linear function of the input variables as discussed in 6.2. This can be written as:

$$f(\mathbf{x}; \boldsymbol{\theta}) = \sum_{i=1}^k \theta_i x_i = \boldsymbol{\theta}^T \mathbf{x}$$

where \mathbf{x} is the input vector, $\boldsymbol{\theta}$ is the parameter vector, and k is the number of input variables. This model does not capture non-linear relationships. A basic approach to capture non-linear relations is to assume the models are linear functions of a set of nonlinear basis functions:

$$f(\mathbf{x}; \boldsymbol{\theta}) = \sum_{i=1}^k \theta_i \phi_i(\mathbf{x}) = \boldsymbol{\theta}^T \boldsymbol{\phi}(\mathbf{x})$$

where $\phi_i(\mathbf{x})$ are nonlinear basis functions and should capture the important non-linear information regarding the inputs. The choice of basis functions is crucial for the model's performance. For example, a polynomial or a kernel basis function can cater to very specific non-linear relationships.

Neural networks are a generalization of this idea. The basic idea is to use a set of adaptive basis functions that are learned from the data. We begin by assuming the basis function has some parameter and then estimate them by minimizing the loss function. Specifically, in neural networks, each basis function $\phi_i(\mathbf{x})$ is assumed to have the following form:

$$\phi_i(\mathbf{x}) = \sigma(\mathbf{w}_i^T \mathbf{x} + b_i)$$

where \mathbf{w}_i and b_i are the parameters of the basis function, and σ is the activation function. The activation function is a non-linear function that introduces non-linearity into the model. Common activation functions include the sigmoid function, the hyperbolic tangent function, and the rectified linear unit (ReLU) function.

These basis functions $\phi_i(\mathbf{x})$ are called neurons and the parameter \mathbf{w}_i and b_i are called weights and biases, respectively.

One can then arrange these neurons, as shown in figure 5, in layers to form a neural network. The first layer is the input layer, the last layer is the output layer, and the layers in between are called hidden layers. This kind of neural network is called a **feedforward neural network**.

6.3.2 Neural Networks for Time Series Forecasting

A feedforward neural network is a poor choice for time series forecasting because it does not account for the serial dependence in the data. A better choice is to use a recurrent neural network (RNN) (see figure 6). In a recurrent neural network, the input (\mathbf{x}) goes through a hidden layer (\mathbf{h}) and then produces two outputs. The first gets fed back to the hidden layer and is used at the next timestamp. The second is the output from the network (or to the next layer).

Another variant of RNN is the Long Short-Term Memory (LSTM) network. LSTM networks are a type of RNN that are designed to capture long-term dependencies in

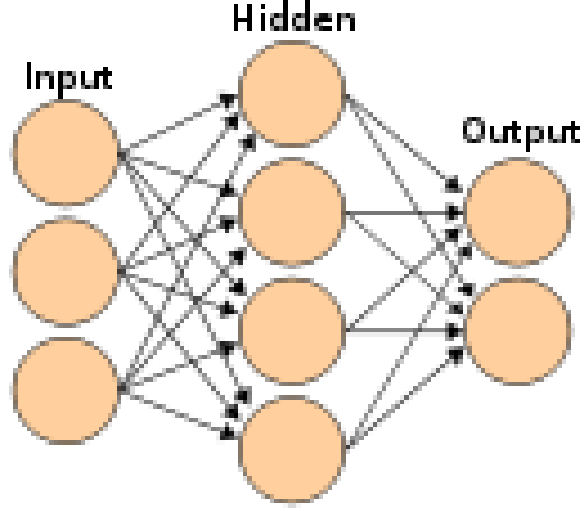


Figure 5: A two-layer neural network

the data. They have a more complex architecture than standard RNNs and include mechanisms to remember and forget information over time. LSTM networks have been shown to be effective for time series forecasting tasks. In this document we will not delve deep into the details of the architecture but rather look at how one can implement an LSTM network for time series forecasting.

6.4 Autocovariance and Autocorrelation Functions

The autocovariance function $\gamma_X(t)$ of a time series X_t is defined as:

$$\gamma_X(s, t) = E[(X_s - \mu_s)(X_t - \mu_t)] \quad (1)$$

for all s and t and μ is the mean of the time series.

The autocorrelation function $\rho_X(t)$ is defined as:

$$\rho_X(s, t) = \frac{\gamma_X(s, t)}{\gamma_X(s, s)\gamma_X(t, t)} \quad (2)$$

Partial autocorrelation function (PACF) is the correlation between two variables after removing the effect of other variables. For a stationary process, x_t , PACF (ϕ_{hh}) for $h = 1, 2, \dots$, is defined as:

$$\phi_{11} = \text{Cor}[x_1, x_0] = \rho(1) \quad (3)$$

and

$$\phi_{hh} = \text{Cor}[x_h - \hat{x}_h, x_0 - \hat{x}_0], \quad h \geq 2 \quad (4)$$

where \hat{x}_h is the regression of x_h on $\{x_1, x_2, \dots, x_{h-1}\}$ and \hat{x}_0 is the regression of x_0 on $\{x_1, x_2, \dots, x_{h-1}\}$.

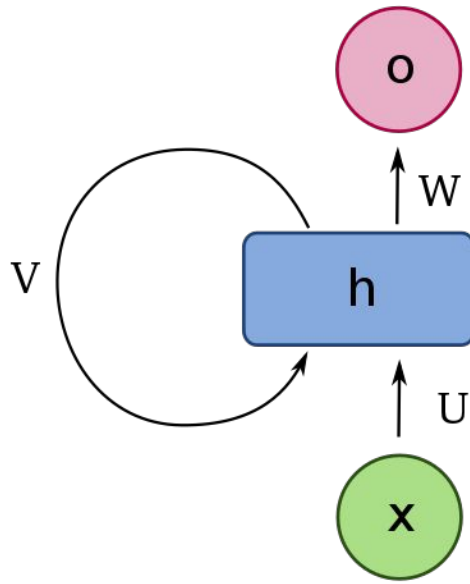


Figure 6: A Recurrent Neural Network

References

- [1] Robert H Shumway, David S Stoffer, and David S Stoffer. *Time series analysis and its applications*, volume 3. Springer, 2000.
- [2] <https://xgboost.readthedocs.io/en/stable/>.
- [3] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018.
- [4] Adrian Tam. Long short-term memory (lstm) for time series prediction in pytorch. <https://machinelearningmastery.com/lstm-for-time-series-prediction-in-pytorch/>, 2023.