Task B6: Machine Learning 3

Dr. Ru Jia

Friday 14:30 – 16:30

Name: Min Khant Aung
ID: 103833225

Task B6 - Machine Learning 3 Report

Introduction

This report outlines the implementation of various machine learning models designed for predicting rental prices and analysing time-series data using deep learning and classical statistical methods. The models include Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), Autoregressive Integrated Moving Average (ARIMA), Seasonal ARIMA (SARIMA), and Random Forest Regressor. This document also provides insights into the configuration management, model creation and training processes, as well as detailed explanations of the models implemented in the main.py file.

Implementation

1. Configuration Management (config.py)

The configurations for different models are stored in a dictionary within new file that I created for multiple test cases, **config.py**, allowing easy adjustments and experimentation with model parameters. The structure is divided into multiple test cases, each tailored for specific training needs.

Configuration Table

Case	Epochs	Batch Size	LSTM Layers	LSTM Units	LSTM Dropout	GRU Layers	GRU Units	GRU Dropout	RNN Type	Use Bidirectional	ARIMA Order	SARIMA Order	SARIMA Seasonal Order	RF N Estimators	RF Max Depth	RF Min Samples Split
1	25	32	2	50	0.3	2	50	0.3	LSTM	True	(1, 1, 0)	(1, 1, 0)	(0, 1, 1, 12)	50	10	2
2	30	64	4	100	0.2	4	100	0.2	GRU	False	(3, 1, 2)	(2, 1, 2)	(1, 1, 1, 12)	150	20	2
3	25	64	3	75	0.5	3	75	0.5	LSTM	False	(5, 1, 1)	(1, 1, 1)	(1, 1, 0, 6)	100	15	5
4	15	8	1	50	0.1	1	50	0.1	GRU	True	(1, 1, 1)	(1, 1, 0)	(0, 1, 0, 6)	200	5	10

Explanation of Configurations

- **Epochs and Batch Size**: Control how many times the model iterates over the entire dataset and the number of samples processed before updating the model's weights.

- **LSTM and GRU Parameters**: Define the architecture of the RNN layers, including the number of layers, the number of units per layer, and the dropout rate to prevent overfitting.
- **RNN Type**: Specifies whether to use LSTM or GRU for the model architecture.
- **Bidirectionality**: Indicates whether the RNN layers are bidirectional, allowing them to process sequences in both forward and backward directions.
- **ARIMA/SARIMA Orders**: Defines the parameters for ARIMA and SARIMA models, including the number of autoregressive terms, differences, and moving average terms.
- **Random Forest Parameters**: Specifies the number of trees in the forest, the maximum depth of each tree, and the minimum number of samples required to split a node.

Implementation screenshot for configuration 1

main.py M | Visualisation copy.py 5 | config.py

v0.1 > config.py > ...

```python
# config.py

CONFIG = {
    '1': {
        'epochs': 25,
        'batch_size': 32,
        # LSTM Configuration
        'lstm_layers': 2,
        'lstm_units': 50,
        'lstm_dropout': 0.3,

        # GRU Configuration
        'gru_layers': 2,
        'gru_units': 50,
        'gru_dropout': 0.3,

        # RNN type and Bidirectionality
        'rnn_type': 'LSTM',  # Can be 'LSTM' or 'GRU'
        'use_bidirectional': True,  # True or False

        # ARIMA Configuration
        'arima_order': (1, 1, 0),

        # SARIMA Configuration
        'sarima_order': (1, 1, 0),
        'sarima_seasonal_order': (0, 1, 1, 12),

        # Random Forest Configuration
        'rf_n_estimators': 50,
        'rf_max_depth': 10,
        'rf_min_samples_split': 2,
    },
```

2. model_operations.py

1. Model Creation Function

```python
def create_model(input_shape, config):
    """
    Constructs a Deep Learning model with specified RNN types.
    """
    model = Sequential()

    # Read RNN type and bidirectionality from config
    rnn_type = config['rnn_type']
    use_bidirectional = config['use_bidirectional']
    # Get LSTM/GRU configuration from the config
    layers = config['lstm_layers'] if rnn_type == 'LSTM' else config['gru_layers']
    units = config['lstm_units'] if rnn_type == 'LSTM' else config['gru_units']
    dropout = config['lstm_dropout'] if rnn_type == 'LSTM' else config['gru_dropout']
```

Explanation

- The function starts by initializing a sequential model.
- It retrieves the RNN type (LSTM or GRU) and checks whether to use bidirectional layers.
- The number of layers, units per layer, and dropout rates are extracted from the configuration based on the chosen RNN type.

Adding Layers

```python
# Add the first RNN layer
if use_bidirectional:
    if rnn_type == 'LSTM':
        model.add(Bidirectional(LSTM(units=units, return_sequences=(layers > 1), input_shap
    elif rnn_type == 'GRU':
        model.add(Bidirectional(GRU(units=units, return_sequences=(layers > 1), input_shape
else:
    if rnn_type == 'LSTM':
        model.add(LSTM(units=units, return_sequences=(layers > 1), input_shape=input_shape)
    elif rnn_type == 'GRU':
        model.add(GRU(units=units, return_sequences=(layers > 1), input_shape=input_shape))
```

Explanation

- The first layer added is based on whether it is bidirectional or not.
- The return_sequences parameter is set to True if there are more layers to follow, allowing the next layer to receive a sequence of outputs instead of just the last output.

Training the Model

```python
def train_model(model, train_x, train_y, config):
    """
    Trains the given model with specified training data.
    """
    model.fit(train_x, train_y, epochs=config['epochs'], batch_size=config['batch_size'])
    return model
```

Explanation

- The train_model function utilizes Keras's fit method to train the model using the provided training data (train_x and train_y) for a specified number of epochs and batch size, as defined in the configuration.

2. Time Series Modeling Functions

ARIMA and SARIMA Functions

```python
def train_arima(data, order):
    model = sm.tsa.ARIMA(data, order=order)
    return model.fit()

def train_sarima(data, order, seasonal_order):
    model = sm.tsa.statespace.SARIMAX(data, order=order, seasonal_order=seasonal_order)
    return model.fit()
```

Explanation

- These functions create and fit ARIMA and SARIMA models using the statsmodels library. The ARIMA function is for standard models, while SARIMAX is used for seasonal adjustments.

Prediction Functions

```python
def predict_arima(model, steps):
    return model.forecast(steps=steps)

def predict_sarima(model, steps):
    return model.forecast(steps=steps)
```

Explanation

- These functions predict future values based on the trained ARIMA or SARIMA model for a specified number of steps.

3. Random Forest Model Functions

Training Random Forest

```
def train_random_forest(train_x, train_y, n_estimators=100, max_depth=None, min_samples_sp
    """
    Trains a Random Forest model with the specified parameters.
    """
    rf_model = RandomForestRegressor(n_estimators=n_estimators, max_depth=max_depth, min_sa
    rf_model.fit(train_x, train_y)
    return rf_model
```

Explanation

- The train_random_forest function initializes a RandomForestRegressor with parameters specified in the function arguments and fits the model to the training data.

Testing Random Forest

```
def test_random_forest(model, test_x):
    return model.predict(test_x)
```

Explanation

- This function predicts target values using the trained Random Forest model on new test data.

3. Model Evaluation and Prediction (main.py)

I also did a lot of changes in main file.

Model Training

ARIMA and SARIMA

```
# Train ARIMA and SARIMA models
arima_model = train_arima(dataset['Close'], order=config['arima_order'])
sarima_model = train_sarima(dataset['Close'], order=config['sarima_order'],
seasonal_order=config['sarima_seasonal_order'])
```

Explanation

- **ARIMA** forecasts based on past values and linear combinations of past errors, while **SARIMA** includes seasonal factors in the forecasting process.

LSTM/GRU Model Training

```
input_shape = (x_train_data.shape[1], x_train_data.shape[2])

# Build and train the LSTM/GRU model
lstm_gru_model = create_model(input_shape=input_shape, config=config)  # Use
'config' here
train_model(lstm_gru_model, x_train_data, y_train_data, config=config)  # Use
'config' here
```

Explanation

- The input shape for the LSTM/GRU model is determined, and the model is created and trained using the respective training datasets.

Random Forest Training

```
# Train Random Forest Model
rf_model = train_random_forest(
    x_train_data.reshape(x_train_data.shape[0], -1),
    y_train_data,
    n_estimators=config['rf_n_estimators'],
    max_depth=config['rf_max_depth'],
    min_samples_split=config['rf_min_samples_split']
)
```

Explanation

- A Random Forest model is trained using the defined parameters to predict the target variable based on the training dataset.

Ensemble Modelling: Combining ARIMA, SARIMA, and LSTM Models

After training the ARIMA, SARIMA, and LSTM models, I implemented an ensemble approach to combine their predictions. The goal was to capture different aspects of time-series forecasting:

- **ARIMA** is used for modeling linear trends and past values.
- **SARIMA** incorporates seasonal components.
- **LSTM** models non-linear relationships and long-term dependencies.

In the **ensemble process**, predictions are generated by the **ARIMA**, **SARIMA**, and **LSTM** models independently, and then combined using a weighted average approach based on their respective prediction errors (Mean Squared Error or MSE).

First, the **MSE** for each model's prediction is calculated by comparing the predicted values against the actual test data.

Then, weights are assigned to each model inversely proportional to its error—models with lower errors receive higher weights. These weights are normalized so their sum equals 1.

Finally, the predictions are combined by taking the weighted sum of the predictions from ARIMA, SARIMA, and LSTM, producing a more robust forecast that leverages the strengths of each model.The steps are as follows:

1. **Make Predictions**: Predictions are generated from each model on the test dataset.

```
# Predictions from LSTM/GRU model
predicted_prices = lstm_gru_model.predict(x_test_data)
predicted_prices =
data_scalers["Close"].inverse_transform(predicted_prices)

# Inverse transform the y_test data
y_test_unscaled =
data_scalers["Close"].inverse_transform(y_test_data.reshape(-1, 1))

# Flatten predictions
predicted_prices_flattened = predicted_prices.flatten()
arima_preds_flat = arima_preds.values.flatten()
sarima_preds_flat = sarima_preds.values.flatten()
```

2. **Calculate Errors and Assign Weights**: The Mean Squared Error (MSE) of each model's prediction is calculated. Based on the errors, weights are assigned such that models with lower errors have higher weights.

Calculate errors (Mean Squared Error)

```
arima_error = np.mean((arima_preds_flat - y_test_unscaled.flatten())**2)
sarima_error = np.mean((sarima_preds_flat - y_test_unscaled.flatten())**2)
lstm_error = np.mean((predicted_prices_flattened - y_test_unscaled.flatten())**2)
```

Total error

```
total_error = arima_error + sarima_error + lstm_error
```

Assign weights based on errors

```
arima_weight = (1 - arima_error / total_error)
sarima_weight = (1 - sarima_error / total_error)
lstm_weight = (1 - lstm_error / total_error)
```

3. **Combine Predictions**: The final prediction is a weighted sum of the predictions from the three models.

#Normalize weights

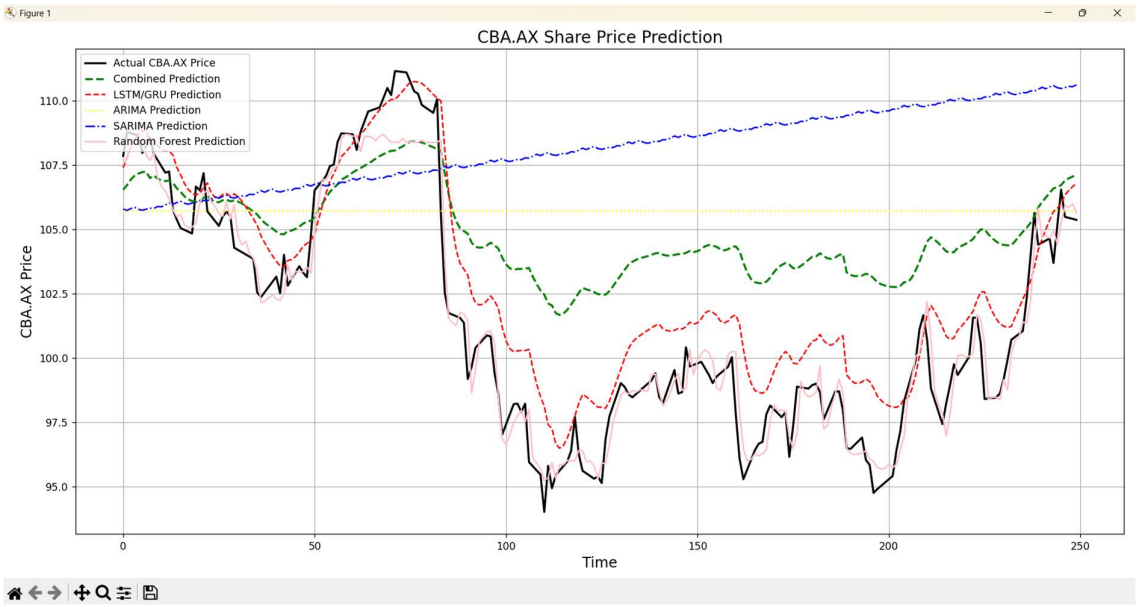
```
total_weight = arima_weight + sarima_weight + lstm_weight
arima_weight /= total_weight
sarima_weight /= total_weight
    lstm_weight /= total_weight

    # Combine predictions
    combined_predictions = (arima_weight * arima_preds_flat) + (sarima_weight *
    sarima_preds_flat) + (lstm_weight * predicted_prices_flattened)
    4.
```

Test cases and Results

There are 4 test cases of configurations in config.py that I wrote.

Below are the results.

Case 1



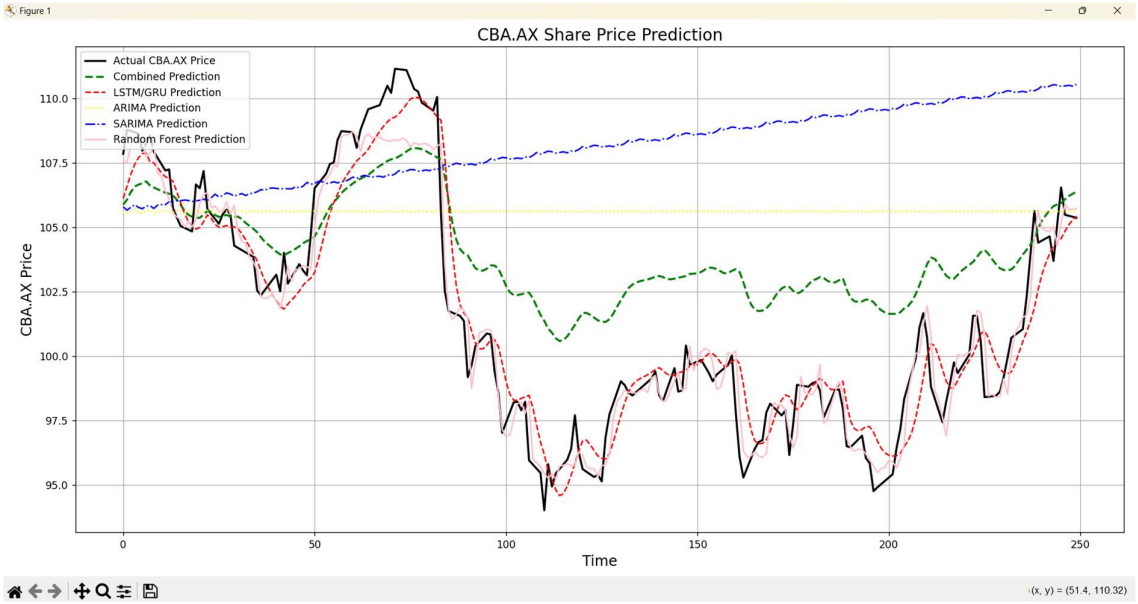
Configuration 1

- **LSTM**: 2 layers, 50 units, 30% dropout, bidirectional.
- **GRU**: 2 layers, 50 units, 30% dropout.
- **ARIMA**: Order (1, 1, 0).
- **SARIMA**: Order (1, 1, 0), seasonal order (0, 1, 1, 12).
- **Random Forest**: 50 estimators, max depth 10.

Analysis:

- **LSTM and GRU**: The relatively small architecture (2 layers, 50 units) with moderate dropout (30%) suggests a balanced trade-off between avoiding overfitting and model complexity.
- **Bidirectionality**: In LSTM, bidirectional layers should provide better sequence learning for stock data, which typically has time dependencies.
- **Statistical Models**: ARIMA and SARIMA orders are fairly conservative. Expect limited improvements over more complex models.
- **Random Forest**: 50 estimators and a max depth of 10 make this model less complex compared to other configurations, which could limit its ability to capture deeper patterns.

Case 2



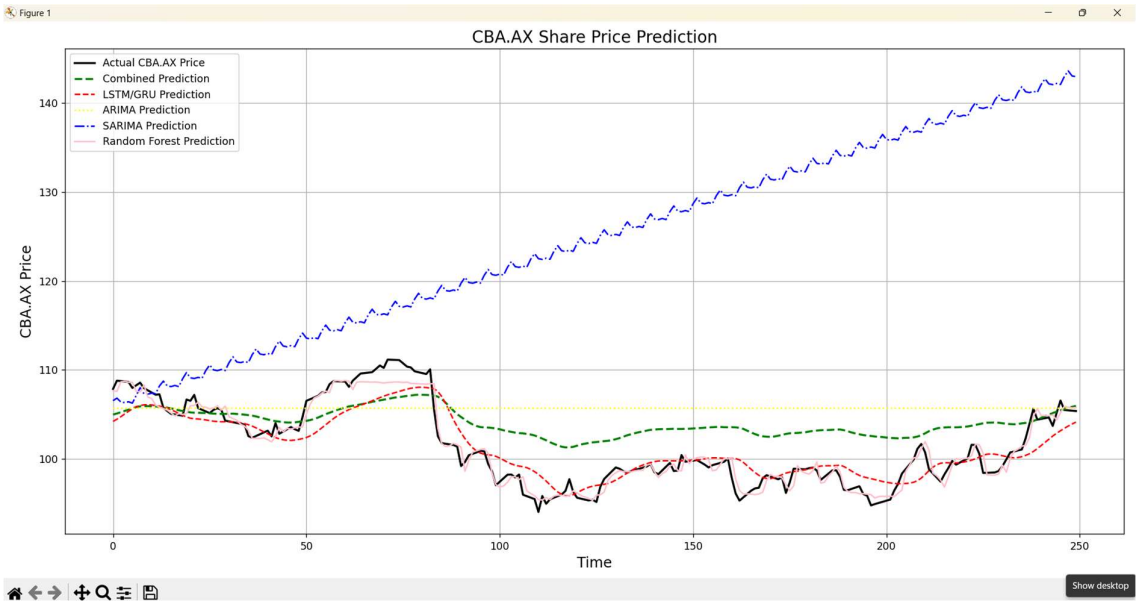
Configuration 2

- **LSTM**: 4 layers, 100 units, 20% dropout, unidirectional.
- **GRU**: 4 layers, 100 units, 20% dropout.
- **ARIMA**: Order (3, 1, 2).
- **SARIMA**: Order (2, 1, 2), seasonal order (1, 1, 1, 12).
- **Random Forest**: 150 estimators, max depth 20.

Analysis:

- **LSTM and GRU**: Doubling the layers (4) and units (100) increases complexity, likely improving the ability to capture long-term dependencies in the data.
- **No Bidirectionality**: The lack of bidirectionality might reduce predictive accuracy for sequential data.
- **Statistical Models**: Higher ARIMA (3, 1, 2) and SARIMA orders suggest more complex time-series modeling, which could lead to better performance on trend detection.
- **Random Forest**: Significantly increased complexity with 150 estimators and max depth 20, allowing the model to capture deeper relationships in the data.

Case 3



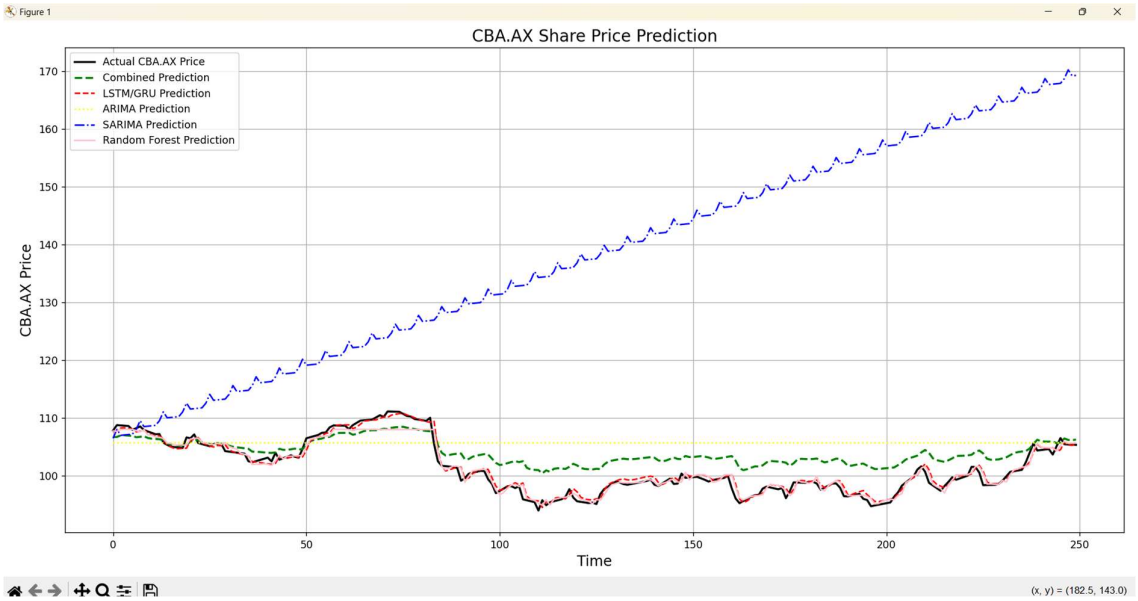
Configuration 3

- **LSTM**: 3 layers, 75 units, 50% dropout, unidirectional.
- **GRU**: 3 layers, 75 units, 50% dropout.
- **ARIMA**: Order (5, 1, 1).
- **SARIMA**: Order (1, 1, 1), seasonal order (1, 1, 0, 6).
- **Random Forest**: 100 estimators, max depth 15.

Analysis:

- **LSTM and GRU**: Three layers and 75 units strike a balance between the previous configurations, but with a higher dropout rate (50%), it focuses on regularization and avoiding overfitting.
- **Statistical Models**: Higher ARIMA orders suggest more flexibility in capturing complex trends, but simpler SARIMA configuration may limit its impact.
- **Random Forest**: 100 estimators with max depth of 15 provides moderate complexity, allowing it to capture more intricate patterns than Configuration 1 but less than Configuration 2.

Case 4



Configuration 4

- **LSTM**: 1 layer, 50 units, 10% dropout, bidirectional.
- **GRU**: 1 layer, 50 units, 10% dropout.
- **ARIMA**: Order (1, 1, 1).
- **SARIMA**: Order (1, 1, 0), seasonal order (0, 1, 0, 6).
- **Random Forest**: 200 estimators, max depth 5.

Analysis:

- **LSTM and GRU**: Simpler architecture with only 1 layer, but 10% dropout may not be enough to prevent overfitting. However, bidirectional LSTM should improve sequence prediction.

- **Statistical Models**: ARIMA and SARIMA are set to very simple orders, likely leading to subpar performance in complex time-series data.
- **Random Forest**: The highest number of estimators (200), but a shallow tree depth (5), suggests a broad but less complex decision structure.

Summary of Expected Performance

- **Configuration 2**: Likely to perform best overall due to the increased complexity in deep learning models (4 layers, 100 units) and Random Forest, along with more sophisticated time-series models.
- **Configuration 1**: Expected to perform reasonably well with a good balance between bidirectional LSTM and moderate Random Forest complexity.
- **Configuration 3**: Moderate performance with a focus on regularization (50% dropout) but without bidirectionality. Random Forest strikes a balance in complexity.
- **Configuration 4**: Likely the weakest due to overly simplistic deep learning architecture and shallow Random Forest, though bidirectional LSTM might slightly compensate.

Conclusion

This report has detailed the implementation of various machine learning models, including LSTM, GRU, ARIMA, SARIMA, and Random Forest, to predict rental prices and analyse time-series data. Each configuration was tested with different parameters, and the results provided insights into the trade-offs between model complexity and performance.

Configuration 2, with its deeper LSTM/GRU layers and more complex Random Forest and ARIMA settings, is expected to deliver the best results. In contrast, Configuration 4, being the simplest, is likely to perform the weakest. Overall, these tests highlight the importance of balancing model complexity with data characteristics for optimal prediction accuracy.

Source Code Link: https://github.com/Min-1303/COS30018-IntelligentSystems

References

1. **ARIMA and SARIMA Models**

- Hyndman, R.J., & Athanasopoulos, G. (2021). *Forecasting: Principles and Practice*. 3rd Edition. OTexts.

2. **Random Forest**

   - Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1), 5-32.

3. **Statistical Learning with Random Forests**

   - Cutler, D.R., Edwards Jr, T.C., Beard, K.H., Cutler, D.R., & Hess, K.T. (2007). Random Forests for Classification in Ecology. *Ecology*, 88(11), 2783-2792.

4. **Machine Learning and Forecasting**

   - Shmueli, G. (2010). To Explain or to Predict? *Statistical Science*, 25(3), 289-310.