COS30018 – Intelligent System

Option B: Stock Price Prediction

Report v0.5

Name Phong Tran
Student Id:104334842

Class: 1-5
Tutor: Tuan Dung Lai

Table of Contents

Implement the essemble models

```
def parser(x):
 return datetime.strptime('190'+x, '%Y-%m')
X = target.values
size = int(len(X) * 0.8)
train, test = X[0:size], X[size:len(X)]
history = [x for x in train]
```

Prase the time to year and month. Using close value to split the train and test into the ratio of 0.8. Then use history to display the all list of training values.

```
# Source: https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/
# evaluate an ARIMA model using a walk-forward validation
# load dataset
arima_predictions = list()
# walk-forward validation
for t in range(len(test)):
 model = ARIMA(history, order=(0,1,0))
 model_fit = model.fit()
 output = model_fit.forecast()
 yhat = output[0]
 arima_predictions.append(yhat)
 obs = test[t]
 history.append(obs)
 #print('predicted=%f, expected=%f' % (yhat, obs))
# evaluate forecasts
rmse = sqrt(mean_squared_error(test, arima_predictions))

print('Test RMSE: %.3f' % rmse)
# plot forecasts against actual outcomes
plot_predict(test, arima_predictions, title='Predicted vs Actual Values')
```

Using the list to display the time series forecasting based on ARIMA with specifying the p, d, and q parameters. Then append output the training fit model of ARIMA into the ARIMA predictions to compare, and using history to save the data. After that, show the root mean square, which is used for measuring the standard deviation of the errors and show us on evaluation of the accuracy of a prediction model. Finally, show the plot based on actual and predicted values.

```
#Scale the data to (0, 1) range
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_train = scaler.fit_transform(train.reshape(-1, 1))
scaled_test = scaler.transform(test.reshape(-1, 1))

#Reshape data for LSTM [samples, 1 time step, 1 feature]
X_train = scaled_train.reshape(scaled_train.shape[0], 1, 1)
X_test = scaled_test.reshape(scaled_test.shape[0], 1, 1)
y_train = scaled_train
y_test = scaled_test
```

In LSTM, we need to scale the data between 0 and 1 to fit_transform the train for learning the parameters and transformation applied into the new data and transform the test to apply the new data by the learned transformation. Reshaping the data for LSTM to split the data into train, test, with X used to applied the reshaped data with resampled train, 1 time step to look back, and 1 feature to look forward the future, and y used to show the scaled data from both train and test that has been applied before.

```
#Define the LSTM model
model = Sequential()
model.add(LSTM(25, return_sequences=False, input_shape=(1, 1)))  # 1 time step and 1 feature
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1)
```

Build the Simple LSTM with 1 time step and 1 feature to predict 1 day based on 1 past look back.

```
test_predict = model.predict(X_test)
test_predict = scaler.inverse_transform(test_predict)
y_test = scaler.inverse_transform(y_test)
test_rmse = sqrt(mean_squared_error(y_test, test_predict))
print('Test RMSE: %.3f' % test_rmse)
```

After that, we need to inverse back to the original value to show the real data. Calculate the root square of the mean squared error to display root mean squared error.

```
lstm_result = [float(x) for x in test_predict]
```

We need to assign the result of the LSTM at the float number for prediction test

```
# Convert predictions to numpy arrays if they are not already
arima_predictions = np.array(arima_predictions)
sarima_predictions = np.array(sarima_predictions)
lstm_result = np.array(test_predict).flatten()

# Simple Averaging Ensemble
ensemble_predictions = (arima_predictions + sarima_predictions + lstm_result) / 3
```

Converting the numpy array result at arima and sarima, however at the result of lstm, we need to make sure that the value should be on 1 dimensional array because previously it has been used reshape data to 3 and 2 for X and y respectively. Then the ensemble is used to calculate the average of three predictions (armina, sarima, and lstm)

```
# Define the weights based on model performance
# Higher weight to models with better performance (lower RMSE)
arima_weight = 0.4
sarima_weight = 0.2
lstm_weight = 0.4

# Weighted Averaging Ensemble
ensemble_predictions_weighted = (arima_weight * arima_predictions +
                                 sarima_weight * sarima_predictions +
                                 lstm_weight * lstm_result)
```

There is another way to calculate is that we need to calculate the ensemble is to put the weights of three model, based on that we will try to figure out which one should be higher, and lower based on the model's performance. The emsemble weight calculation is based on the sum of every available model on weight multiply with the prediction.

```
# Evaluate the Weighted Ensemble Predictions
ensemble_rmse_weighted = sqrt(mean_squared_error(y_test, ensemble_predictions_weighted))
print(f'Weighted Ensemble Test RMSE: {ensemble_rmse_weighted:.3f}')
```

We need to check to make sure that if the ensemble weighted is low enough. If it higher than the simple ensemble, we need to put the test whether lower lstm or sarima or arima, or even put higher number at these three models.

```
def create_sequences(data, time_step=30):
    X, y = [], []
    for i in range(len(data) - time_step - 1):
        X.append(data[i:(i + time_step), 0])
        y.append(data[i + time_step, 0])
    return np.array(X), np.array(y)

scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data)

n_steps = 30 # the days lookback
X, y = create_sequences(scaled_data, n_steps)

X = X.reshape((X.shape[0], X.shape[1], 1))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=False)
X_train.shape, X_test.shape
```

We need to create the sequences method for checking the past days look back and the split the data into training and testing set of X and y, in order to apply for future models, such as Random Forest, LSTM, GRU, RNN.

```
def evaluate_model(y_test, y_pred):
    y_pred_rescaled = scaler.inverse_transform(y_pred.reshape(-1, 1))
    rmse = sqrt(mean_squared_error(y_test, y_pred_rescaled))
    print(f'RMSE: {rmse}')
    return rmse, y_pred_rescaled
```

This evaluation model method to check the root mean square error. The method operates as we need to inverse_transform the prediction y data to go back the actual data.

RANDOM FOREST

```
# Flatten the data for Random Forest
X_train_rf = X_train.reshape((X_train.shape[0], X_train.shape[1]))
X_test_rf = X_test.reshape((X_test.shape[0], X_test.shape[1]))

# Loop over different hyperparameters manually
n_estimators_list = [100, 200, 300]
max_depth_list = [10, 20, 30]

best_rmse_rf = float('inf')
best_params = {}

for n_estimators in n_estimators_list:
    for max_depth in max_depth_list:
        # Train Random Forest with different hyperparameters
        rf_model = RandomForestRegressor(n_estimators=n_estimators, max_depth=max_depth, random_state=42)
        rf_model.fit(X_train_rf, y_train)
        rf_predictions = rf_model.predict(X_test_rf)
        rf_rmse = sqrt(mean_squared_error(y_test, rf_predictions))
        print(f'n_estimators: {n_estimators}, max_depth: {max_depth}, RMSE: {rf_rmse}')

        # Track the best performing model
        if rf_rmse < best_rmse_rf:
            best_rmse_rf = rf_rmse
            best_params = {'n_estimators': n_estimators, 'max_depth': max_depth}

print(f'Best RF Hyperparameters: {best_params}, Best RMSE: {best_rmse_rf}')
```

We need to flatten the data into 2D before testing with different hyperameters, then let the loop to explore which one is the best before display the best tracking on the performance level, Root mean square error lowest means that the hyperparameter is chosen.

```
best_rf_model = RandomForestRegressor(n_estimators=200, max_depth=10, random_state=42)
best_rf_model.fit(X_train_rf, y_train)

best_rf_predictions = best_rf_model.predict(X_test_rf)
best_rf_rmse , best_rf_predictions_rescaled= evaluate_model(y_test_rescaled, best_rf_predictions)
```

We uses the best model to train, then evaluate to assign the rmse and their rescaled prediction random forest.

```
import keras_tuner as kt

# Define the model-building function for Keras Tuner
def build_lstm_model(hp):
    model = Sequential()
    # Tune the number of units
    model.add(LSTM(units=hp.Int('units', min_value=50, max_value=200, step=50), input_shape=(n_steps, 1)))
    model.add(Dense(1))

    # Tune the learning rate
    model.compile(optimizer=hp.Choice('optimizer', ['adam', 'rmsprop']), loss='mean_squared_error')

    return model

# Initialize the tuner
tuner = kt.RandomSearch(
    build_lstm_model,
    objective='val_loss',
    max_trials=10,
    executions_per_trial=1,
    directory='tuner_results',
    project_name='lstm_tuning'
)

# Run the search
tuner.search(X_train, y_train, epochs=50, validation_data=(X_test, y_test), batch_size=32)

# Get the optimal hyperparameters
best_hp = tuner.get_best_hyperparameters()[0]
print(f'Best Hyperparameters: {best_hp.values}')
```

Building the simple LSTM model with compiling the choice for optimizer whether adam or rmsprop. We used keras tuner for fine tunning the model to make sure the find the best, with initialising the tuner, with 10 times hyperparameters trial and many executions_per_trail for controlling the number of independent model trainings per hyperarameters, then export into the directory. After that, we run the search to find the best before displaying the best hyperparameters.

```
# Best hyperparameters from the tuner
best_units = 200
best_optimizer = 'rmsprop'

best_lstm_model = Sequential()
best_lstm_model.add(LSTM(best_units, input_shape=(n_steps, 1)))
best_lstm_model.add(Dense(1))
best_lstm_model.compile(optimizer=best_optimizer, loss='mean_squared_error')
best_lstm_model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1, validation_data=(X_test, y_test))
best_lstm_predictions = best_lstm_model.predict(X_test)

# Evaluate the final model

best_lstm_rmse, best_lstm_predictions_rescaled = evaluate_model(y_test_rescaled, best_lstm_predictions)
```

Applied the best one for training the model and display the output in progress bar (verbose), before display on the predictions value and assign the method evaluate_model into the best rmse lstm and rescaled lstm predictions.

GRU

```
# Define the model-building function for Keras Tuner
def build_gru_model(hp):
    model = Sequential()
    # Tune the number of units
    model.add(GRU(units=hp.Int('units', min_value=50, max_value=200, step=50), input_shape=(n_steps, 1)))
    model.add(Dense(1))

    # Tune the learning rate or optimizer
    model.compile(optimizer=hp.Choice('optimizer', ['adam', 'rmsprop']), loss='mean_squared_error')

    return model

# Initialize the tuner
tuner_gru = kt.RandomSearch(
    build_gru_model,
    objective='val_loss',
    max_trials=10,
    executions_per_trial=1,
    directory='tuner_results_gru',
    project_name='gru_tuning'
)

# Run the search
tuner_gru.search(X_train, y_train, epochs=50, validation_data=(X_test, y_test), batch_size=32)

# Get the optimal hyperparameters
best_hp_gru = tuner_gru.get_best_hyperparameters()[0]
print(f'Best GRU Hyperparameters: {best_hp_gru.values}')
```

Same as LSTM, we applied to different model such as GRU for searching the hyperarameters before finding the best to show in evaluation.

```
# Best hyperparameters from the tuner
best_units_gru = 150
best_optimizer_gru = 'rmsprop'

best_gru_model = Sequential()
best_gru_model.add(GRU(best_units_gru, input_shape=(n_steps, 1)))
best_gru_model.add(Dense(1))
best_gru_model.compile(optimizer=best_optimizer_gru, loss='mean_squared_error')
best_gru_model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1, validation_data=(X_test, y_test))

best_gru_predictions = best_gru_model.predict(X_test)

# Evaluate the final model
best_gru_rmse, best_gru_predictions_rescaled = evaluate_model(y_test_rescaled, best_gru_predictions)
```

Getting the best hyperparameters of GRU to applied for training the model, before evaluating into the value of best rootmeansquare for GRU and GRU's predictions rescaled.

RNN

```
# Define the model-building function for Keras Tuner
def build_rnn_model(hp):
    model = Sequential()
    # Tune the number of units
    model.add(SimpleRNN(units=hp.Int('units', min_value=50, max_value=200, step=50), input_shape=(n_steps, 1)))
    model.add(Dense(1))

    # Tune the learning rate or optimizer
    model.compile(optimizer=hp.Choice('optimizer', ['adam', 'rmsprop']), loss='mean_squared_error')
    return model

# Initialize the tuner for RNN
tuner_rnn = kt.RandomSearch(
    build_rnn_model,
    objective='val_loss',
    max_trials=10,
    executions_per_trial=1,
    directory='tuner_results_rnn',
    project_name='rnn_tuning'
)

# Run the search
tuner_rnn.search(X_train, y_train, epochs=50, validation_data=(X_test, y_test), batch_size=32)

# Get the optimal hyperparameters
best_hp_rnn = tuner_rnn.get_best_hyperparameters()[0]
print(f'Best RNN Hyperparameters: {best_hp_rnn.values}')
```

Same as GRU and LSTM, we applied the same approach for RNN to find the optimal hyperparameters. Then display the best hypermeters.

```
# Best hyperparameters from the tuner
best_units_rnn = 200
best_optimizer_rnn = 'rmsprop'

# Define the final RNN model with the best hyperparameters
best_rnn_model = Sequential()
best_rnn_model.add(SimpleRNN(best_units_rnn, input_shape=(n_steps, 1)))
best_rnn_model.add(Dense(1))
best_rnn_model.compile(optimizer=best_optimizer_rnn, loss='mean_squared_error')
best_rnn_model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1, validation_data=(X_test, y_test))

best_rnn_predictions = best_rnn_model.predict(X_test)
best_rnn_rmse, best_rnn_predictions_rescaled = evaluate_model(y_test_rescaled, best_rnn_predictions)
```

Applying the best hypermeters to train the model, then retrieve the result of method evaluate_model to apply for best root mean square for rnn and rescaled predictions rnn.

```
def evaluate_model(y_test, y_pred):
    #y_pred_rescaled = scaler.inverse_transform(y_pred.reshape(-1, 1))
    rmse = sqrt(mean_squared_error(y_test, y_pred))
    print(f'RMSE: {rmse}')
    return rmse, y_pred
```

Update the method evaluate_model, only root mean square error is kept, because we will try to apply the ensemble prediction, which is not to use transform the shape of the value.

```
best_lstm = np.array(best_lstm_predictions_rescaled).flatten()
best_gru = np.array(best_gru_predictions_rescaled).flatten()
best_rnn = np.array(best_rnn_predictions_rescaled).flatten()
best_rf = np.array(best_rf_predictions_rescaled).flatten()
```

The value of the best in all models need to be flatten into 1 dimensional array because ensemble only allows that value.

```
# Simple Averaging Ensemble (take the mean of the four models)
ensemble_predictions = (best_rnn + best_rf + best_gru + best_lstm) / 4

# Evaluate the Weighted Ensemble Predictions
ensemble_rmse_avg, ensemble_prediction_rescaled = evaluate_model(y_test_rescaled, ensemble_predictions)
```

With four available model, we tried to calculate the average of them to assign the ensemble predictions. Using the method to evaluate the average ensemble of root mean square and assign into the rescaled prediction ensemble for plotting the data.

```
rmse = [best_rf_rmse, best_lstm_rmse, best_gru_rmse, best_rnn_rmse]
rmse
```

```
[5.816587089873092, 4.890835931353441, 4.128703274761103, 4.055255841119359]
```

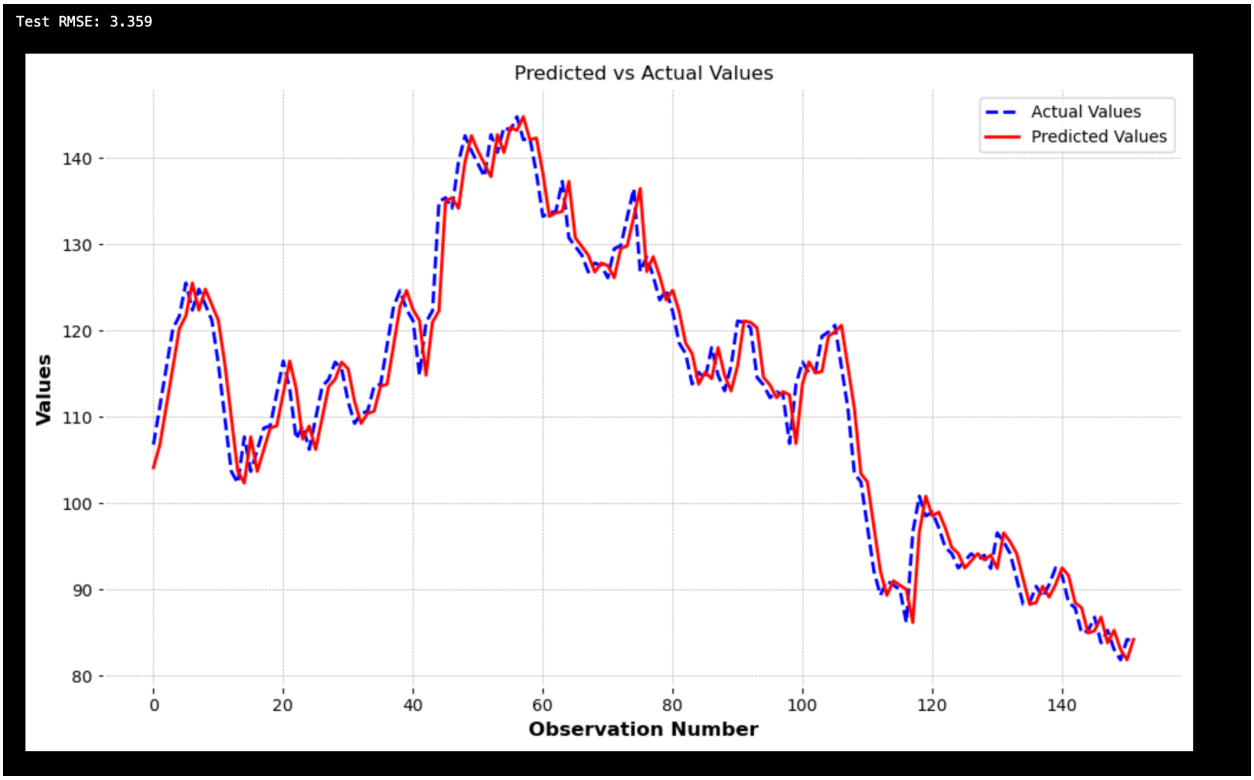
```
rmse = np.array([best_rf_rmse, best_lstm_rmse, best_gru_rmse, best_rnn_rmse])
inverse_rmse = 1 / rmse
weights = inverse_rmse / np.sum(inverse_rmse)
rf_weight, lstm_weight, gru_weight, rnn_weight = weights
```

Assign four best root mean square error into the array, before calculate the weights by the inversing rmse divide with sum of inverse_rmse. After that the weights assign into four respective models as specified before at the rmse variable.

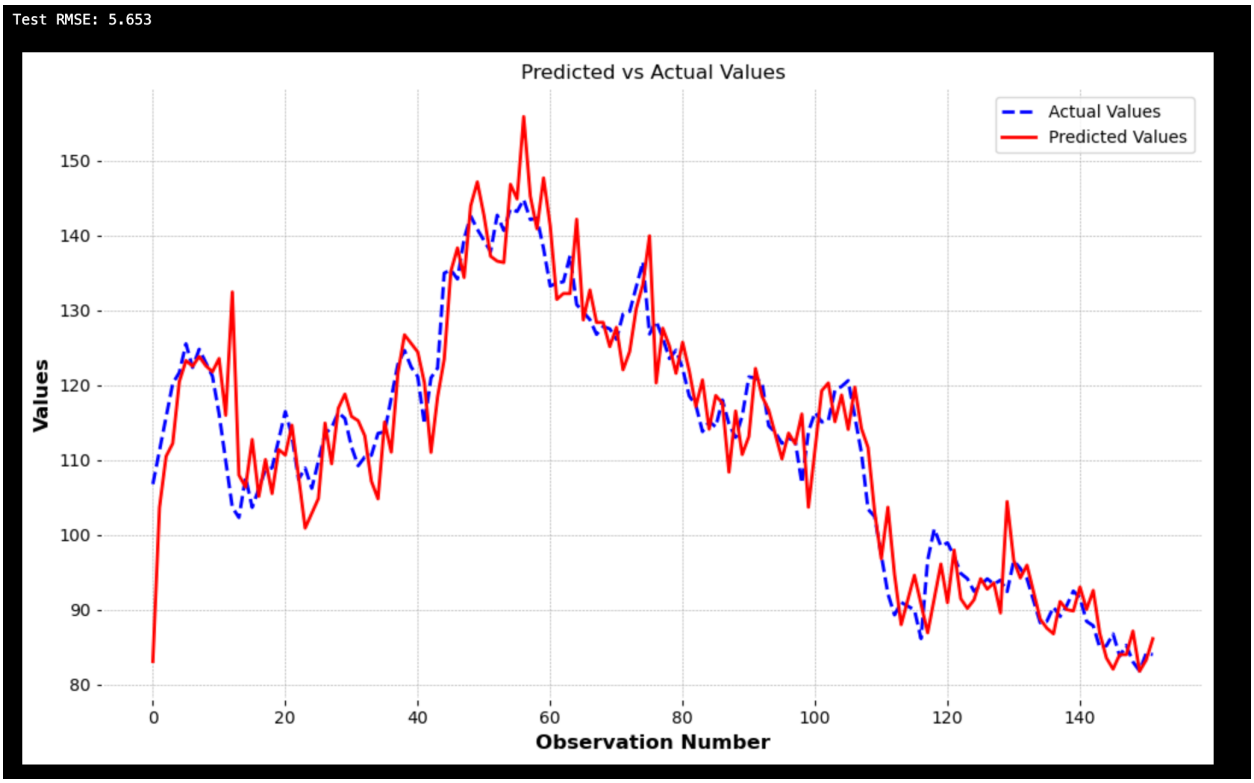
```
# Weighted Averaging Ensemble
ensemble_predictions_weighted = (best_lstm * lstm_weight +
                                 best_gru * gru_weight +
                                 best_rnn * rnn_weight +
                                 best_rf * rf_weight)

# Evaluate the Weighted Ensemble Predictions
ensemble_rmse_weighted, ensemble_prediction_rescaled = evaluate_model(y_test_rescaled, ensemble_predictions_weighted)
```

Calculate the weight averaging ensemble same as we did for only ensemble of LSTM, ARIMA, and SARIMA, then evaludate the model to assign ensemble_rmse_weighted and ensemble_prediction_rescaled for plotting the graph.

Summary result of experiment



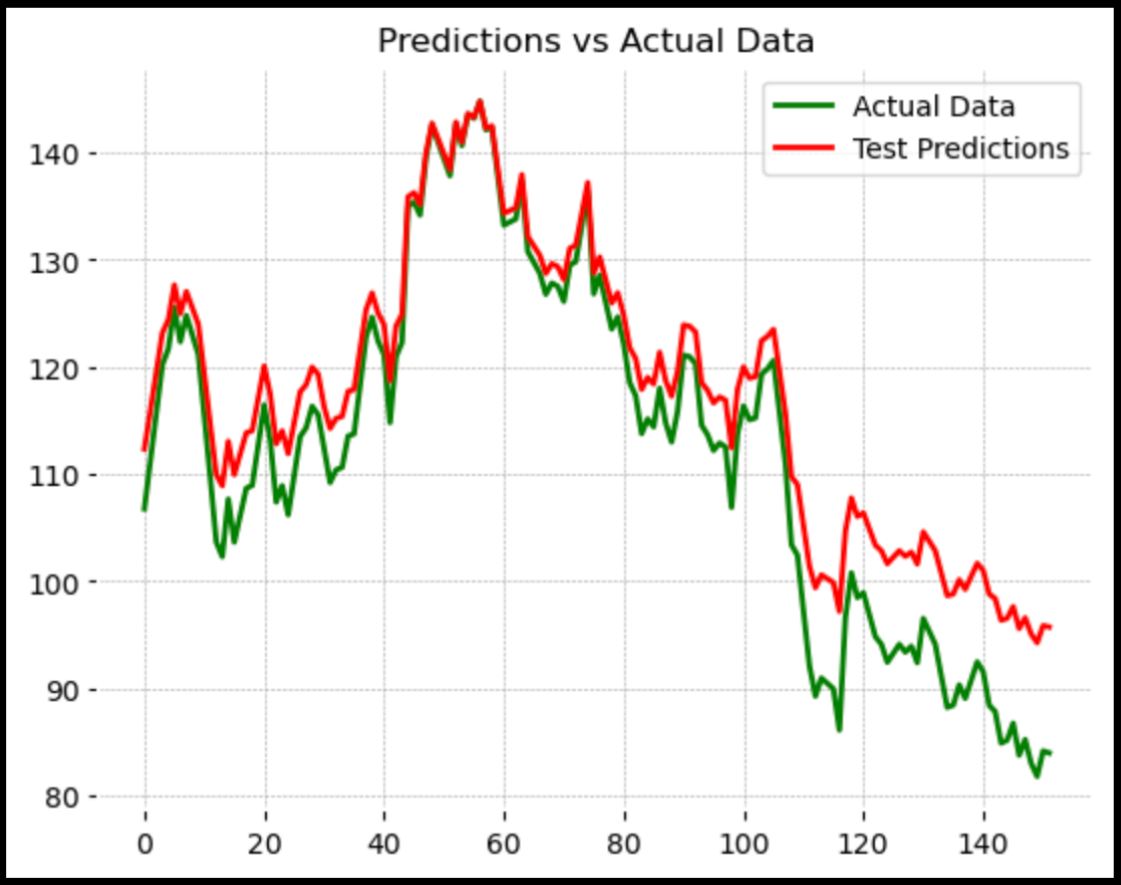
Test RMSE of ARIMA shows the result is 3.359, and the plot between two values are slightly close.



Test RMSE of SARIMA is more than 5, which means the error is higher than the ARIMA, and the plot shows the gap between two values, which means the prediction value lacks reality when observing with actual value.

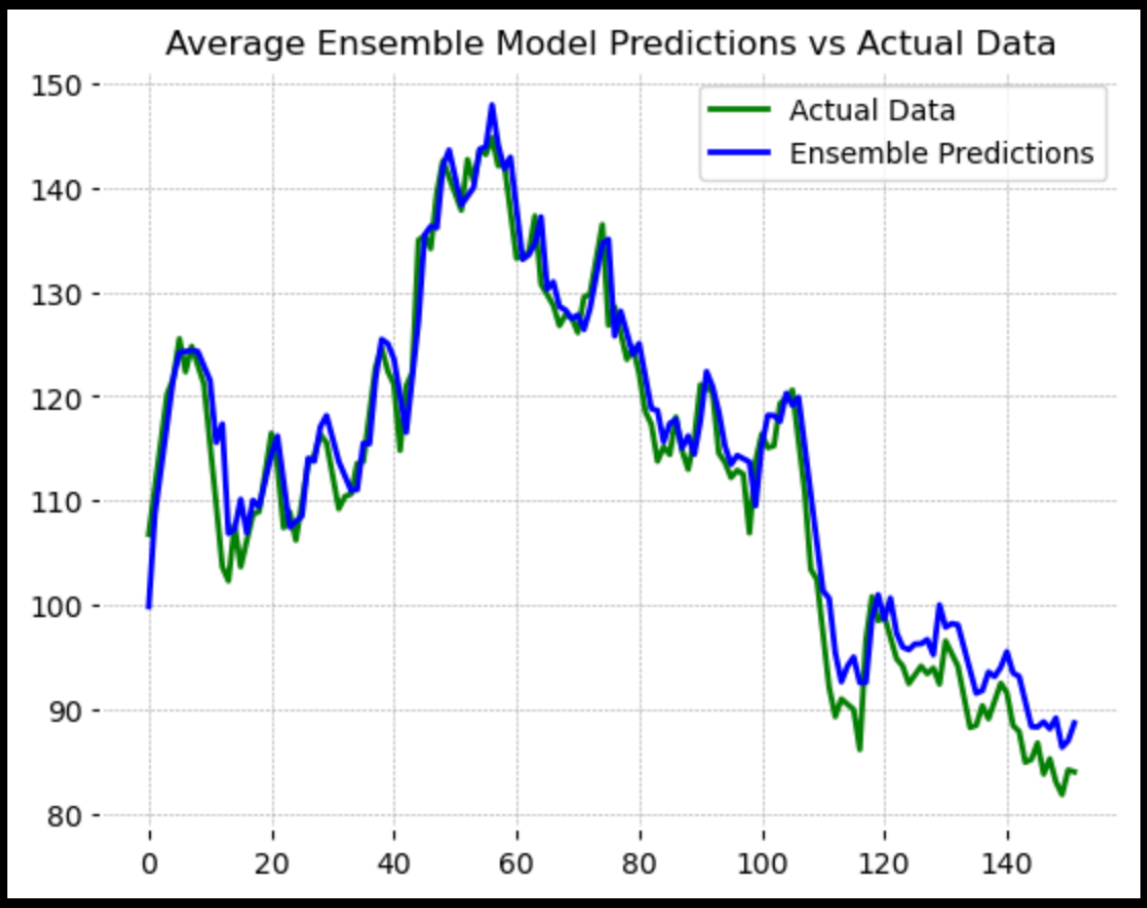
Test RMSE: 5.942

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...



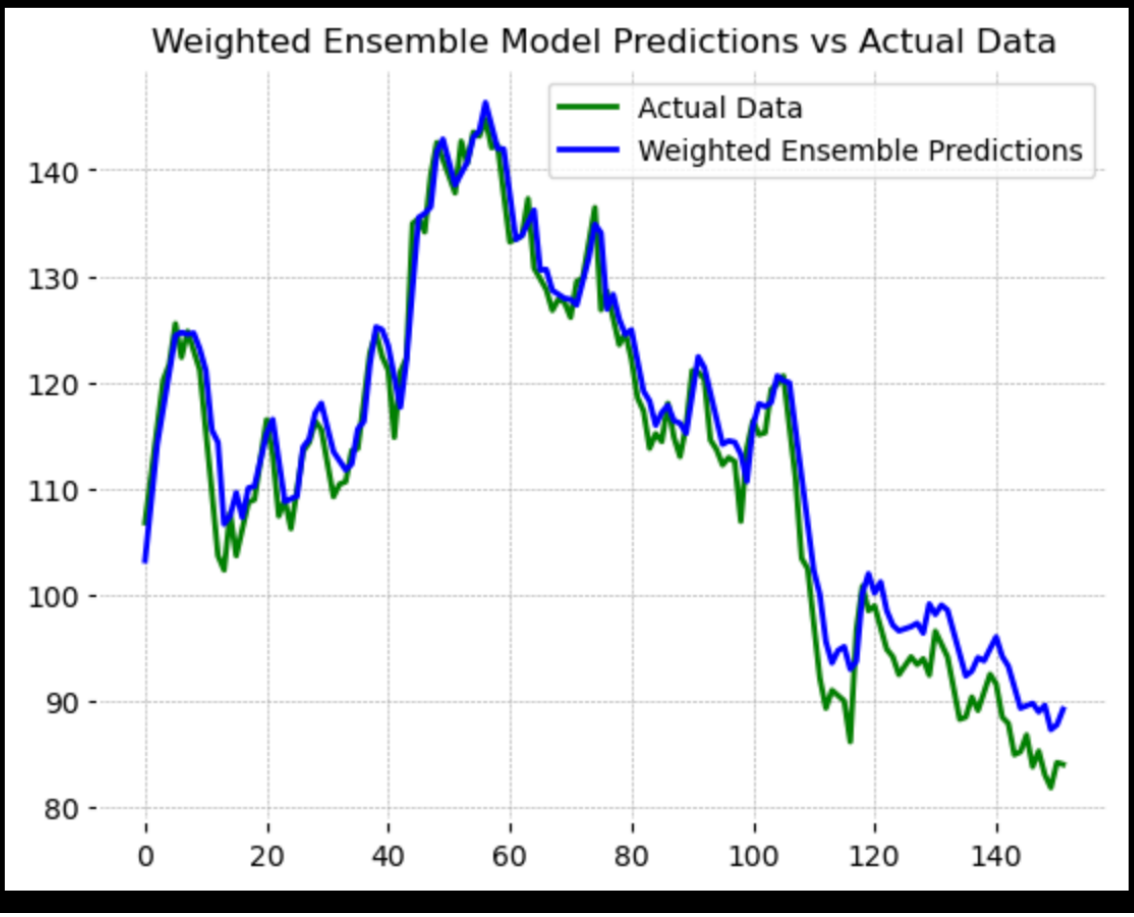
The LSTM root mean square error seems slightly higher than SARIMA, which shows the is a large gap between actual and predicted values, especially when looking at the observation in around 110 till last.

```
Ensemble Test RMSE: 3.507
```



Simple ensemble predictions shows the root mean square is much lower comparing with SARIMA and LSTM, and slightly lower than ARIMA. The gap between actual and predicted values seems not much, except at 110, from bit more than 120 to after 140.

```
Weighted Ensemble Test RMSE: 3.415
```

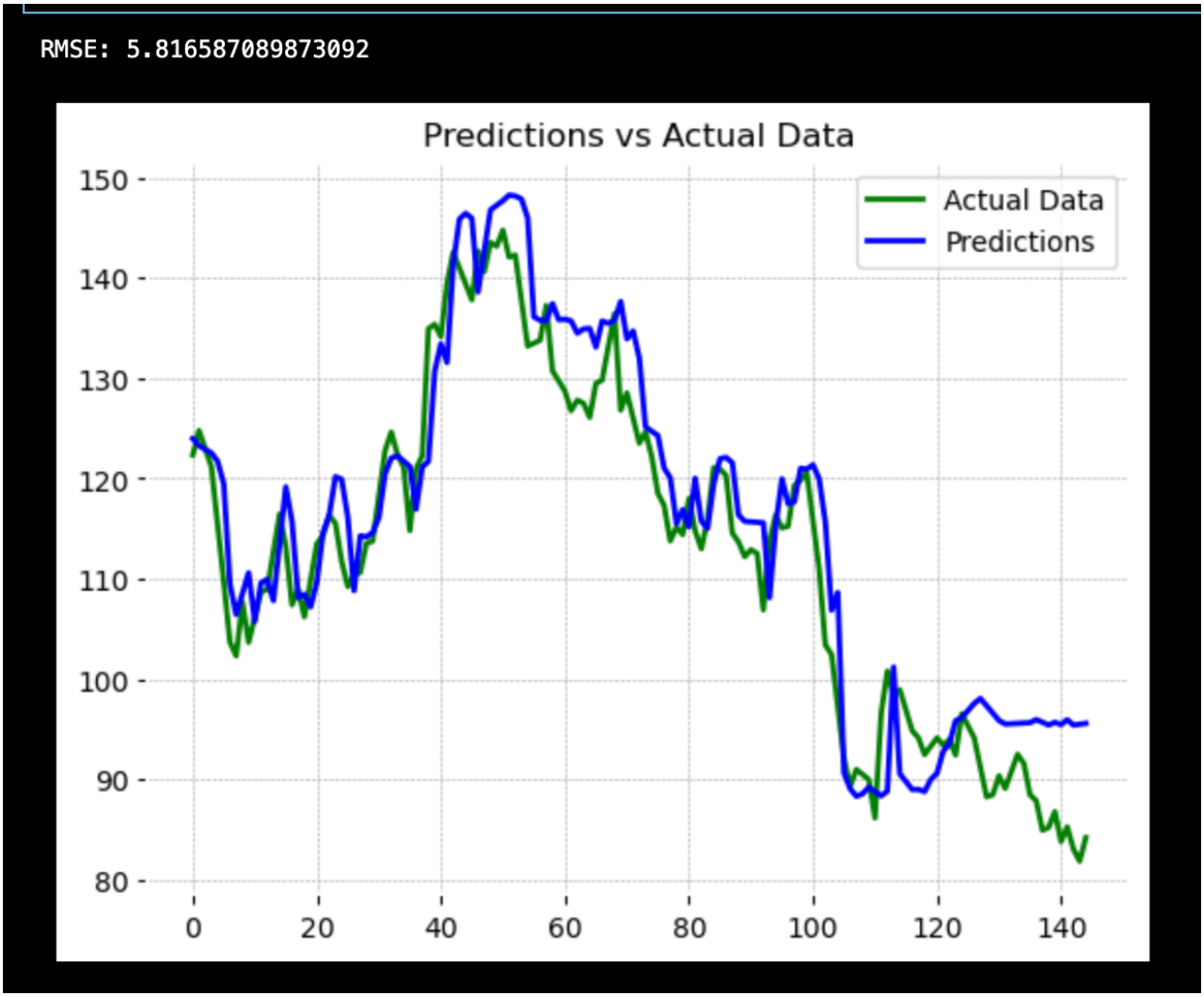


When experimenting the optimal weights for the ensemble, the root mean square error seems dropped slightly, although there is not any difference much from the weighted ensemble model compared with simple ensemble model.

```
n_estimators: 100, max_depth: 10, RMSE: 0.05619805485885643
n_estimators: 100, max_depth: 20, RMSE: 0.05625890511633933
n_estimators: 100, max_depth: 30, RMSE: 0.05625890511633933
n_estimators: 200, max_depth: 10, RMSE: 0.05552801482519782
n_estimators: 200, max_depth: 20, RMSE: 0.05564414670643148
n_estimators: 200, max_depth: 30, RMSE: 0.05564414670643148
n_estimators: 300, max_depth: 10, RMSE: 0.05553500992736693
n_estimators: 300, max_depth: 20, RMSE: 0.05577380299246915
n_estimators: 300, max_depth: 30, RMSE: 0.05577380299246915
Best RF Hyperparameters: {'n_estimators': 200, 'max_depth': 10}, Best RMSE: 0.05552801482519782
```

Hyperarameters for Random Forest shows at different estimators and depth, with the root mean square error value. The root mean square error from different hyperaramers search

not showing much gap between them. And the best hyperparameters that is the highest number of estimators and the lowest depth.



Applied the optimal hyperparameters to train the model, the root mean square error seems really high, alongside the plot the prediction value seems not show quite reality.

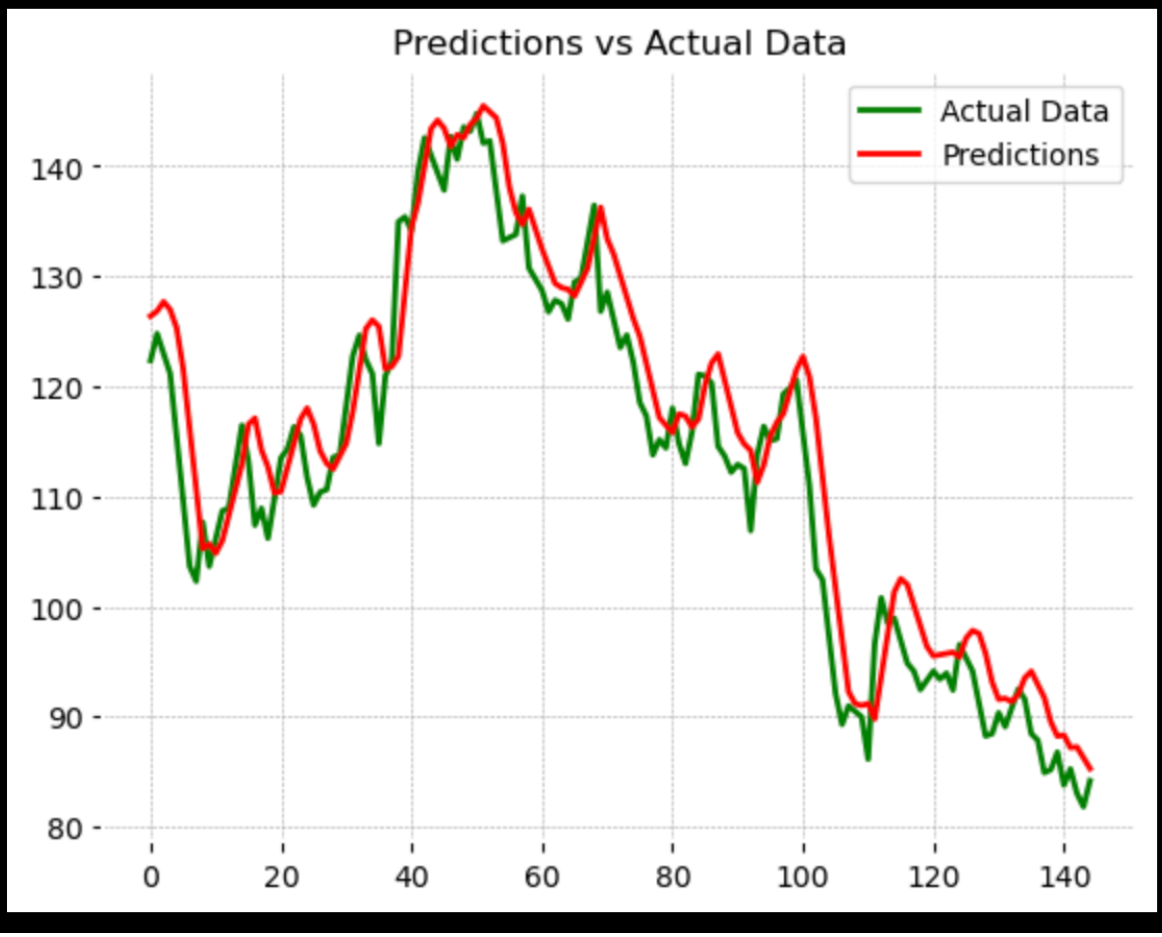
```
Trial 8 Complete [00h 00m 20s]
val_loss: 0.002538818633183837

Best val_loss So Far: 0.0015758618246763945
Total elapsed time: 00h 03m 13s
Best Hyperparameters: {'units': 200, 'optimizer': 'rmsprop'}
```

LSTM hyperparameters search taken only 20 seconds trail and the total elapsed time is 3 minutes 13 seconds to find with the optimal provider. The trails is calculate with 4 units multiply with 2 optimizers equals to 8 trails.

```
RMSE: 4.890835931353441
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
<matplotlib.legend.Legend at 0x2642bf3a910>
```



Applied for the optimal hyperparameters to train the model, which shows the root mean square is lower than the RandomForest.

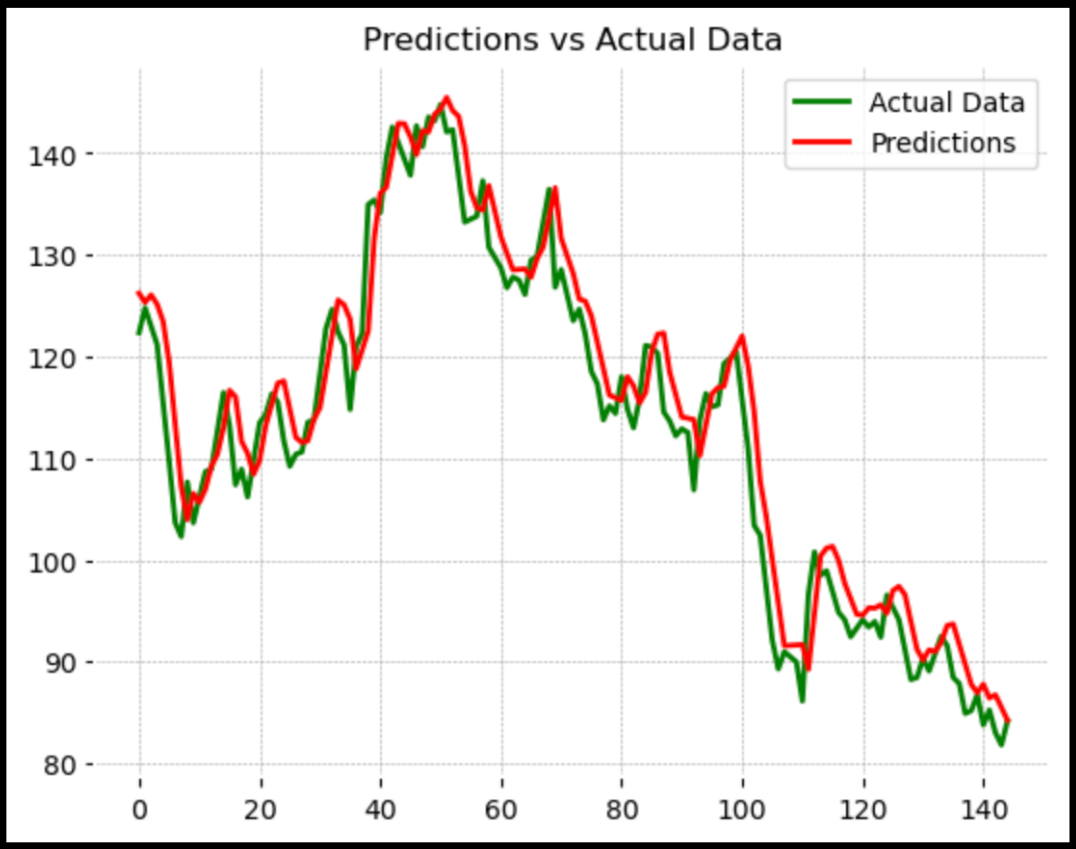
```
Trial 8 Complete [00h 00m 24s]
val_loss: 0.0012688779970631003

Best val_loss So Far: 0.0012316388310864568
Total elapsed time: 00h 02m 41s
Best GRU Hyperparameters: {'units': 150, 'optimizer': 'rmsprop'}
```

Hyperarameter search for GRU to find the optimal one within 24 seconds on trail and elapsed takes 02 minutes 41 seconds to find the optimal GRU hyperarameters.

```
RMSE: 4.128703274761103
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
<matplotlib.legend.Legend at 0x2641dcc8090>
```



Applied the optimal hyperparameters GRU for training, and it shows lower root mean square value error with LSTM and Random Forest.

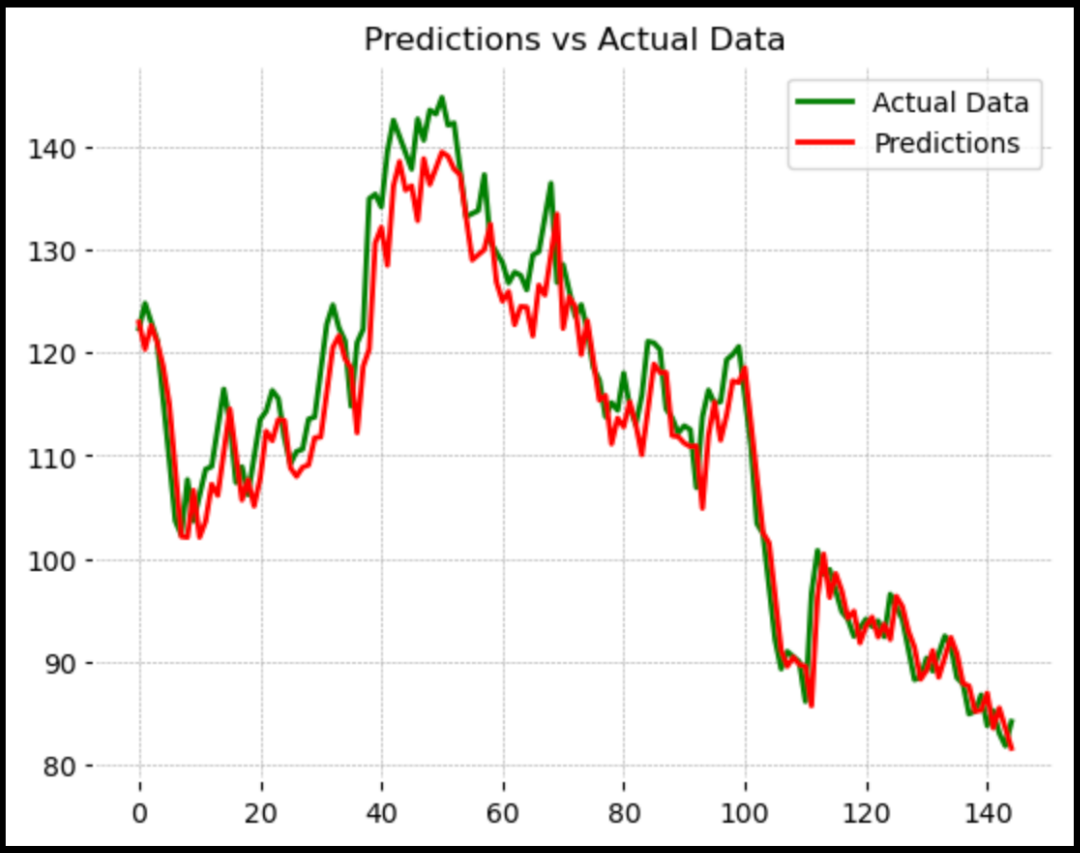
```
Trial 8 Complete [00h 00m 10s]
val_loss: 0.001200784114189446

Best val_loss So Far: 0.001112928963266313
Total elapsed time: 00h 01m 31s
Best RNN Hyperparameters: {'units': 200, 'optimizer': 'rmsprop'}
```

Hyperarameter search for RNN within 10 seconds on trails, and check the val_loss in 1 minutes 31 seconds to find the optimal hyperarameters.

```
RMSE: 4.055255841119359
```

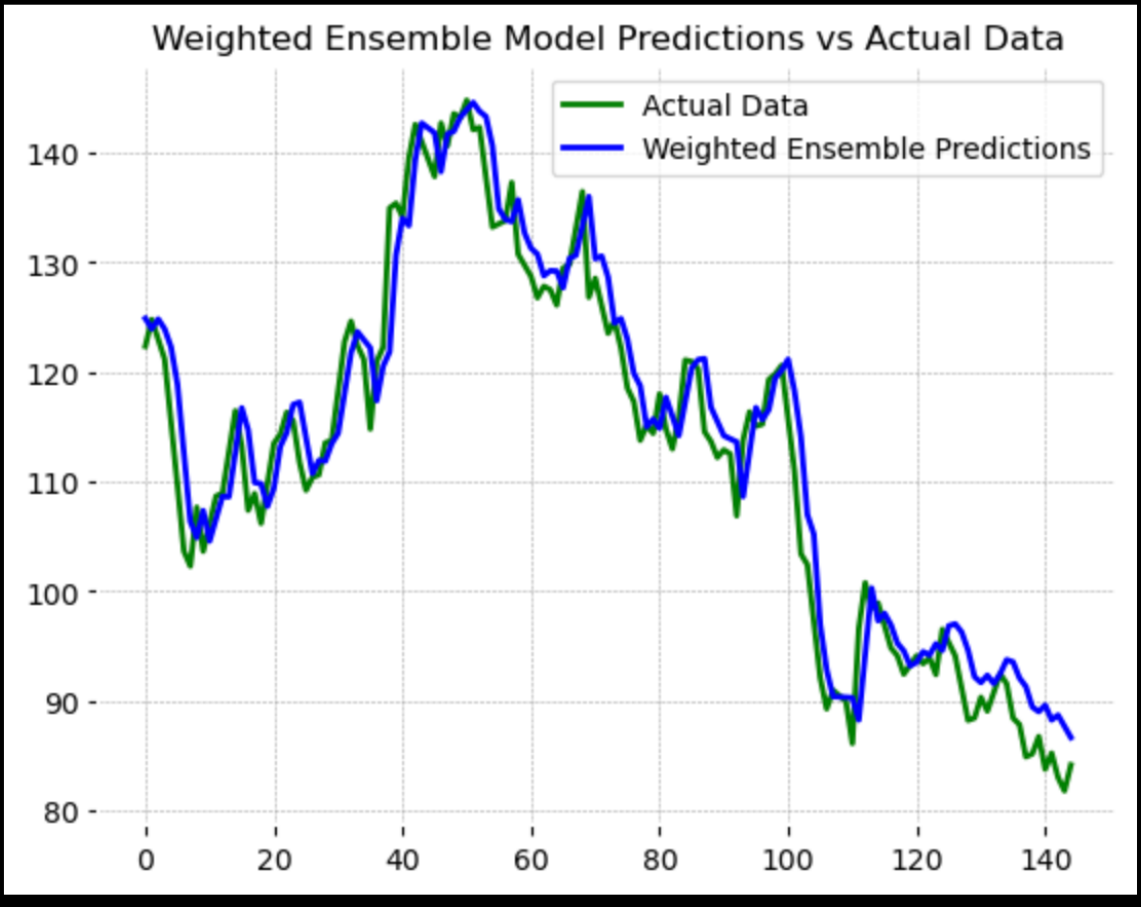
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
<matplotlib.legend.Legend at 0x264248f8090>
```



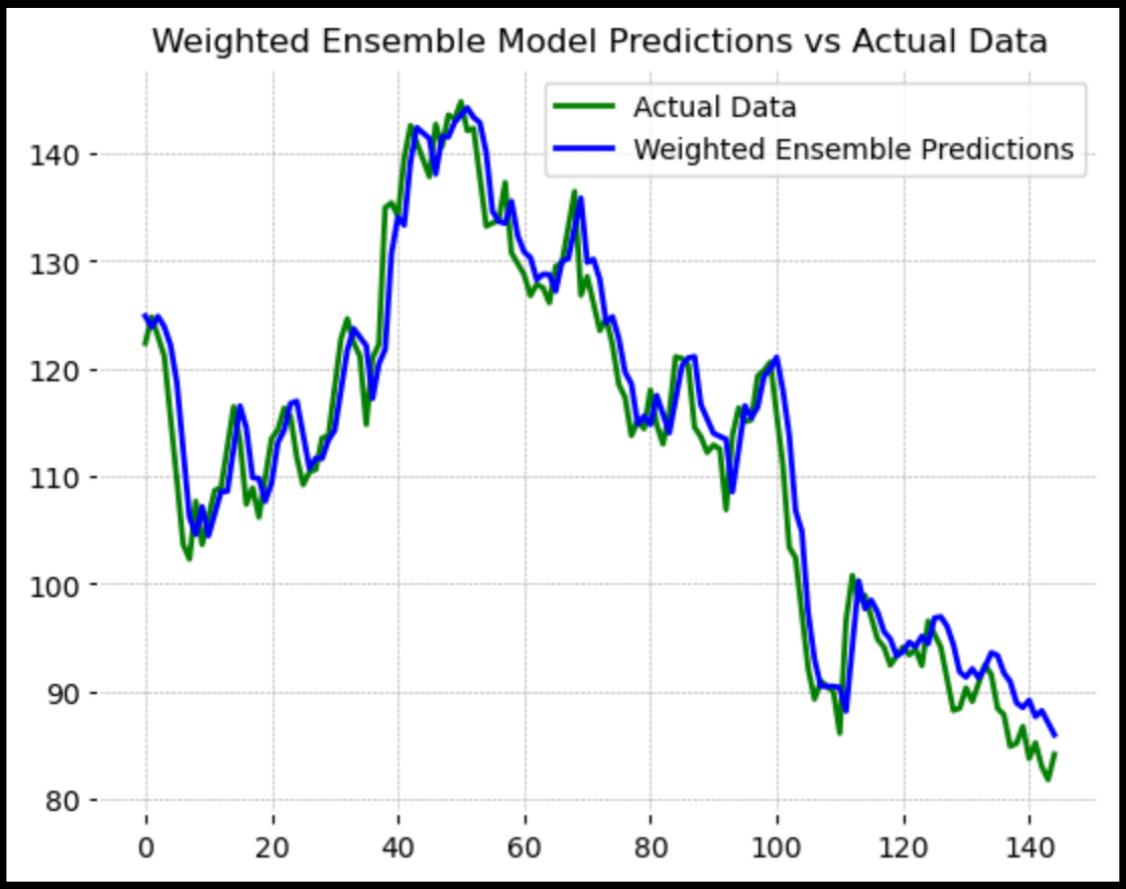
Applied the best hyperarameters to train the model, the root mean square error shows that the RNN is lower than LSTM, GRU, and RandomForest.

```
RMSE: 3.9712084559060092
```



Applied the simple average ensemble to display the significant drop in root mean square error and the plot seems nearly tight in the values of actual and predictions.

```
RMSE: 3.8700937865101803
```



Calculate the weighted averaging ensemble by inverse weight, and it shows that the root mean square error is slightly lower than the hyperparameters of simple ensemble.