Lampiran 3 Model Arima

```
def _plot_arima_metrics(result: pd.DataFrame, city_name: str):

    plt.figure(figsize=(20, 5))

    bar_plot = plt.bar(result.index, result['mse'],
color="#EB4747")

    plt.xticks(result.index, result.apply(lambda i:
f"({int(i['p'])}, {int(i['d'])}, {int(i['q'])})",
axis=1).tolist(), rotation=90)

    plt.title(f"Metrics Evaluation on ARIMA Parameter Tuning
({city_name})")

    plt.ylabel("Error Rate (MSE)")

    plt.xlabel("Order Set")

    for i, bar in enumerate(bar_plot):

        plt.text(bar.get_x() + bar.get_width() / 2,
bar.get_height() + 0.01, f"{round(bar.get_height(), 3)}",
ha='center', va='bottom', rotation=0)

    plt.show()

def _plot_arima_forecasting(model: object, sample:
pd.DataFrame, city_name: str, train: pd.DataFrame, test:
pd.DataFrame):

    test_forecast = model.get_forecast(steps=len(test))

    test_forecast_series = pd.DataFrame({value:
test_forecast.predicted_mean}, index=test.index)

    forecast = model.get_forecast(steps=len(test) + 2)

    forecast_series = pd.DataFrame({value:
forecast.predicted_mean}, index=test.index.union([2024,
2025])).tail(2)

    plt.figure(figsize=(20, 10))

    plt.grid(True, linestyle='--', alpha=0.7)

    plt.bar(train.index, train[value], label="Training Set",
alpha=0.7)

    plt.bar(test.index, test[value], label="Testing Set",
color="orange", alpha=0.7)

    plt.plot(test_forecast_series[value], label="Prediction",
color="#228B22", linewidth=2)

    bar_forecast = plt.bar(forecast_series.index,
forecast_series[value], label="Forecast",
color="cornflowerblue")

    plt.xticks(sample.index.union([2024, 2025]),
sample.index.union([2024, 2025]))
```

```
    for x, y in zip(test_forecast_series.index,
```

```
   results = []

    for p, d, q in itertools.product(interval_p, interval_d,
interval_q):

        try:

            model_arima = ARIMA(train.values, order=(p, d, q))

            model_arima = model_arima.fit()

            test_forecast =
model_arima.get_forecast(steps=len(test))

            test_forecast_series =
pd.DataFrame({data.columns[0]: test_forecast.predicted_mean},
index=test.index)

            mse_test = mean_squared_error(test.values,
test_forecast_series.values)

            results.append({'p': p, 'd': d, 'q': q, 'mse':
mse_test})

        except Exception as e:

            print(f"Error for ARIMA({p}, {d}, {q}): {e}")


    results = pd.DataFrame(results)

    results["ranking"] = results["mse"].rank(method="min",
numeric_only=True).astype(int)

    results =
results.sort_values(by="ranking").reset_index(drop=True)

    results.to_excel(f"experiments/tuning_result_arima -
{city_name.lower()}.xlsx", index=False)

    _plot_arima_metrics(results, city_name)

    p, d, q = results.iloc[0, :3].astype(int)

    model_arima_best = ARIMA(train.values, order=(p, d, q))

    model_arima_best = model_arima_best.fit()

    model_arima_best.save(f"models/arima_forecast -
{city_name.lower()}.pkl")

    _plot_arima_forecasting(model_arima_best, data, city_name,
train, test)

    return model_arima_best
```

Lampiran 4 Model LSTM

```
     def lstm_forecast(data: pd.DataFrame, city_name: str,
n_steps: int = 2, n_features: int = 1, train_ratio: float =
0.7, future_years: List[int] = [2024, 2025]):

   """

   Generates forecasts for the test data and additional future
years using an LSTM model.



   Args:

       data (pd.DataFrame): Input DataFrame containing the time
series data.

       city_name (string, required): City name from sample data
used.

       n_steps (int, optional): Number of time steps to use for
the LSTM input sequence. Default is 2.

       n_features (int, optional): Number of features in the
input data. Default is 1.

       train_ratio (float, optional): Ratio of data to be used
for training. Default is 0.7.

       future_years (list, optional): List of additional future
years to forecast. Default is [2024, 2025].

   Returns:

       pd.DataFrame: DataFrame containing the forecasts for the
test data and additional future years.

   """

   data = data.set_index("tahun")

   df_sample = data.copy()

   # Split data into sequences

   X, y = split_sequences(df_sample.values, n_steps)

   # Split data into train and test

   train_size = int(len(X) * train_ratio)

   X_train, X_test, y_train, y_test = X[:train_size],
X[train_size:], y[:train_size], y[train_size:]

   # Reshape data for LSTM input

   train_shape = (X_train.shape[0], X_train.shape[1],
n_features)
```

```
history = model_lstm.fit(X_train_reshape, y_train, epochs=250,
verbose=0, validation_data=(X_test_reshape, y_test))

   # Visualize loss on training and validation metrics

   plot_loss(history, city_name)

   # Generate prediction for test data

   test_forecasts = generate_forecasts(model_lstm,
X_test_reshape, n_steps, n_features)

   test_forecast_series = pd.DataFrame({'value':
test_forecasts}, index=data.iloc[train_size+2:].index)

   # Generate forecasts for additional future years

   future_forecasts = []

   last_sequence = X_test[-1].reshape(1, n_steps, n_features)

   for _ in range(len(future_years)):

       prediction = model_lstm.predict(last_sequence,
verbose=0)

       future_forecasts.append(prediction[0, 0])

       last_sequence = np.roll(last_sequence, -1, axis=1)

       last_sequence[0, -1, 0] = future_forecasts[-1]

   # Combine result for test data and future years

   forecast_series = pd.concat([test_forecast_series,
pd.DataFrame({'value': future_forecasts}, index=future_years)],
axis=0)

   # Wrap training and testing series

   train_series = df_sample.iloc[:train_size+2]

   test_series  = df_sample.iloc[train_size+2:]

   # Visualize result

   plot_lstm_forecasting(train_series, test_series,
forecast_series, city_name)

   return model_lstm

def split_sequences(sequences, n_step):

   Splits the input sequences into input (X) and output (y)
sequences for LSTM training.

   Args:

       sequences (np.array): Input sequences.

       n_step (int): Number of time steps for the input
```

```
Returns:

       tuple: Tuple containing the input (X) and output (y)
sequences.

   X, y = [], []

   length = len(sequences)

   for i in range(length):

       end = i + n_step

       if end > length - 1:

           break

       seq_x, seq_y = sequences[i:end, 0], sequences[end, 0]

       X.append(seq_x)

       y.append(seq_y)

   X = np.array(X)

   y = np.array(y)

   return X, y

def create_model(n_steps, n_features):

   Creates and returns an LSTM model.

   Returns:

       model: LSTM model.

   model = Sequential()

   model.add(LSTM(units=100, activation="relu",
input_shape=(n_steps, n_features)))

   model.add(Dense(1))

   model.compile(optimizer="adam", loss="mse")

   return model

def generate_forecasts(model, X_test, n_steps, n_features)

   Generates forecasts for the test data using the trained LSTM
model.

   Args:

       model (keras.models.Sequential): Trained LSTM model.

       X_test (np.array): Test input sequences.

       n_steps (int): Number of time steps for the input
sequence.
```

```
last_sequence = X_test[0].reshape(1, n_steps, n_features)

   for i in range(len(X_test)):

       prediction = model.predict(last_sequence, verbose=0)

       forecasts.append(prediction[0, 0])

       last_sequence = np.roll(last_sequence, -1, axis=1)

       last_sequence[0, -1, 0] = forecasts[-1]

   return forecasts

def plot_lstm_forecasting(train, test, forecast, city_name):

   Visualizes the training, testing, and forecasted data using
a bar plot and line plot.

   Args:

       train (pd.Series): Training data.

       test (pd.Series): Testing data.

       forecast (pd.Series): Forecasted data.

   plt.figure(figsize=(20, 10))

   plt.grid(True, linestyle='--', alpha=0.7)

   indexes = list(train.index) + list(forecast.index)

   plt.bar(train.index, train['tingkat_pengangguran_terbuka'],
label="Training Set", alpha=0.7)

   plt.bar(test.index, test['tingkat_pengangguran_terbuka'],
label="Testing Set", color="orange", alpha=0.7)

   plt.plot(test.index, forecast['value'].iloc[:len(test)],
label="Prediction", color="#228B22", linewidth=2)

   bar_forecast = plt.bar(forecast.index[len(test):],
forecast['value'].iloc[len(test):], label="Forecast",
color="cornflowerblue")

   plt.xticks(indexes, indexes)

   plt.title(f"Unemployment Rate Forecasting ({city_name})",
fontsize=16)

   plt.xlabel("Year", fontsize=14)

   plt.ylabel("Unemployment Rate (%)", fontsize=14)

   plt.legend(fontsize=12, loc='upper right')

  for x, y in zip(test.index,
forecast['value'].iloc[:len(test)]):

       plt.annotate(f"{round(y, 2)}%", xy=(x, y), xytext=(x, y
+ 0.2), fontsize=10, ha='center', va='bottom')

   for i, bar in enumerate(bar_forecast):
```

```
def plot_loss(history, city_name):

    Visualizes the training and validation loss curves along
with annotations for the highest and lowest losses.

    Args:

        history (keras.callbacks.History): History object
containing the training and validation losses.

    Returns:

        None (displays the plot)

    plt.figure(figsize=(20, 5))

    plt.plot(history.history['loss'], label="Training Loss",
color="blue")

    plt.plot(history.history['val_loss'], label="Validation
Loss", color="red")

    plt.title(f"Training & Validation Loss ({city_name})",
fontsize=16)

    plt.xlabel("Epoch", fontsize=14)

    plt.ylabel("Loss", fontsize=14)

    plt.legend(fontsize=12)

    max_train_loss_idx = np.argmax(history.history['loss'])

    min_train_loss_idx = np.argmin(history.history['loss'])

    max_train_loss =
history.history['loss'][max_train_loss_idx]

    min_train_loss =
history.history['loss'][min_train_loss_idx]

    max_val_loss_idx = np.argmax(history.history['val_loss'])

    min_val_loss_idx = np.argmin(history.history['val_loss'])

    max_val_loss =
history.history['val_loss'][max_val_loss_idx]

    min_val_loss =
history.history['val_loss'][min_val_loss_idx]

    plt.text(max_train_loss_idx, max_train_loss, f"Max Train:
{max_train_loss:.4f}", color="blue", ha="left", va="top",
fontsize=12)

    plt.text(min_train_loss_idx, min_train_loss, f"Min Train:
{min_train_loss:.4f}", color="blue", ha="right", va="top",
fontsize=12)

    plt.text(max_val_loss_idx, max_val_loss, f"Max Val:
```