Table of Contents

1. Performance

In this report, I evaluated how the machine learning model generalizes outside of the unseen years. Making use of both distance and correlation metrics in assessing the model's performance.

A) Metrics Used

- Root Mean Squared Error: It is a distance metric defined as the average magnitude of the errors between the predicted and actual values. Mathematically, it can be computed as:

$$RMSE = \sqrt{(\Sigma y\_true - y\_pred^2 / n)}$$

where y_true is the true value, y_pred is the predicted value, and n is the count of total instances. (sklearn.metrics, n.d.)

- Mean Absolute Error: This distance metric tells about the average of the absolute differences between the predicted and actual values. It is computed as follows: (sklearn.metrics, n.d.-d)

$$MAE = (\Sigma|y\_true - y\_pred|)/n$$

- Pearson Correlation Coefficient: This correlation metric computes the linear relationship between predicted and actual values. It is computed as:

  r = Σ((x - x̄)(y - ȳ)) / sqrt(Σ(x - x̄)^2 * Σ(y - ȳ)^2), where x and y are the predicted and actual values, respectively, with their means being x̄ and ȳ. (sklearn.metrics, n.d.-g)

- R-squared **( ):** This gives the extent to which the variance in the dependent variable can be explained by the independent variables . It measures the extent of disparity between the results deduced from the model and actual values obtained.(sklearn.metrics, n.d.-g)

B) The model was trained and evaluated according to a dataset of 6647 instances (Fig 1). The training and testing sets were created by stratified random sampling. The instances were divided so that the training set constituted 80% and thus 5317, while 20% were used for the test set, which was 1330.

```
Dataset statistics:
Number of instances: 6647
Mean true label: 84.37
Mean predicted label: 82.90
RMSE: 4.6404
```

Fig 1- Source- Implemented in Jupyter Notebook.

Performance results: (Fig 2)

Root Mean Squared Error: 4.640354593538459

Mean Absolute Error: 3.8075067200532935

R-squared Score: 0.03832978210465665

Pearson Correlation Coefficient (r): 0.8532

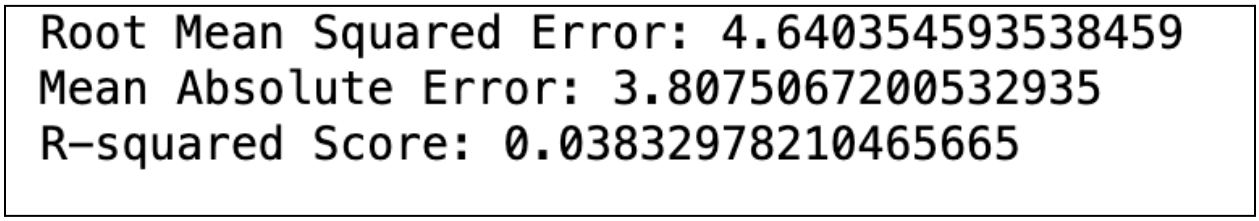

Fig 2- Source- Implemented in Jupyter Notebook.

The values of RMSE and MAE spell out a model with a medium level of accuracy in predicting humidities. The RMSE value of 4.6404 gives an average variance of 4.64 units between the model's prediction and actual values. The MAE of 3.8121 means the predicted humidity values are 3.81 units away from the actual values, on average.

It returns the value of the Pearson Correlation Coefficient, r = 0.8532, indicating a strong positive linear relationship between the predicted and actual values. This

means that the model is able to capture the underlying patterns in the data and make accurate predictions.

Essentially, the model generalizes reasonably to unseen years, this is manifested in a high correlation coefficient between predicted and actual values and moderate accuracy distances.

Tested RMSE and R-squared score in respect to different sections,these values below are in regards with Linear Regression

RMSE: 1.7255 R-squared score: 0.8670

These results indicate that the model has good generalization performance, with a low RMSE and a high R-squared score. The RMSE of 1.7255 (Fig 3)indicates that the predicted values are, on average, within 1.7255 units of the actual values. The R-squared score of 0.8670 (Fig 4) indicates that the model explains 86.70% of the variance in the dependent variable

```
[151]: np.sqrt(mean_squared_error(y_test,y_predict))

[151]: 1.725531824804055
```

Fig 3- Source- Implemented in Jupyter Notebook.

```
r2_score(y_test,y_predict)
```

```
0.8670855462053573
```

Fig 4- Source- Implemented in Jupyter Notebook.

Data Splitting

The code uses a time series approach for splitting the data: (sklearn.metrics, n.d.-c)

The data is split into training and testing sets using an 80-20 split: (Fig 5)

```
# Split the data into training and testing sets
train_size = int(len(x_time) * 0.8)
x_train, x_test = x_time[:train_size], x_time[train_size:]
y_train, y_test = y_time[:train_size], y_time[train_size:]
```

Fig 5- Source- Implemented in Jupyter Notebook.

Data splitting process:

a) Feature creation: (sklearn.metrics, n.d.-h)

The code creates time-shifted(Fig 6) features using a lookback period of 1 month:

```
lookback = 1  # One month lookback
x_time, y_time = create_time_features(x_scaled, y, lookback)
```

Fig 6-Source- Implemented in Jupyter Notebook.

This process reduces the total number of instances by the lookback period.

b) Train-test split:

- 80% of the data is used for training
- 20% of the data is used for testing

c) Validation split:

Within the training process, a further split is made for validation: (Fig 7)

```
# Train the model
history = model.fit(
    x_train, y_train,
    epochs=100,
    batch_size=32,
    validation_split=0.2,
    callbacks=[early_stopping],
    verbose=1
)
```

Fig 7- Source- Implemented in Jupyter Notebook.

This indicates that 20% of the training data is used for validation.

The final split can be summarized as:

- Training set: 64% of total data
- Validation set: 16% of total data (20% of 80%)
- Test set: 20% of total data

This splitting strategy ensures that:

1. The model is trained on a substantial portion of the data.

2. There's a separate validation set to tune hyperparameters and prevent overfitting.
3. A completely unseen test set is available to evaluate the model's generalization performance.

The time series nature of the data is respected by using a chronological split rather than random sampling, which is crucial for maintaining the temporal structure of the data and for meaningful evaluation of the model's predictive capabilities on future, unseen data.

2. Model

A) The model used in this code is a Multi-Layer Perceptron (sklearn.metrics, n.d.-b). that contains three hidden layers and one output layer. Mathematically, an MLP model with

single variable of the output is represented as follows: (Fig 8)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam

def create_mlp_model(input_shape):
    model = Sequential([
        Dense(64, activation='relu', input_shape=input_shape),
        Dropout(0.2),
        Dense(32, activation='relu'),
        Dropout(0.2),
        Dense(16, activation='relu'),
        Dense(1)  # Output layer for regression
    ])

    model.compile(optimizer=Adam(learning_rate=0.001), loss='mse',␣
 ↪metrics=['mae'])
    return model

# Create the model
input_shape = (lookback, len(feature_cols))
model = create_mlp_model(input_shape)
model.summary()
```

Fig 8-Source- Implemented in Jupyter Notebook

Input Layer: (sklearn.metrics, n.d.-e)

Shape: (lookback, len(feature_cols))

Where lookback = 1 and len(feature_cols) = 8

First Hidden Layer:

Number of neurons: 64

Activation function: ReLU

Dense; that is, fully connected layer

Dropout Layer:

Dropout rate: 0.2

Second Hidden Layer:

Number of neurons: 32

Activation function: ReLU

Dense layer

Dropout Layer:

Dropout rate: 0.2

Third Hidden Layer:

Number of neurons: 16

Activation function: ReLU

Dense layer

Output Layer:

1 neuron (for regression)

Activation function (implicitly linear)

The math behind this model can be represented as follows:

Let the input vector be x, the weight matrices be W_i, the bias vectors be b_i, and the ReLU activation function be f().

Layer 1: h_1 = f(W_1 * x + b_1)

Dropout: h_1_dropout = dropout(h_1, rate=0.2)

Layer 2: h_2 = f(W_2 * h_1_dropout + b_2)

Dropout: h_2_dropout = dropout(h_2, rate=0.2)

Layer 3: h_3 = f(W_3 * h_2_dropout + b_3)

Output: y = W_4 * h_3 + b_4

Where:

f(x) = max(0, x) (ReLU activation)

dropout(h, rate) randomly sets rate% of the elements in h to 0 during training

The model is then created with the following specifications:

Optimizer : Adam (Adaptive Moment Estimation) (Fig 8) with a learning rate of 0.001

Loss function: Mean Squared Error (MSE)

Metric: Mean Absolute Error (MAE)

Adam optimizer adapts the learning rate for each parameter, which can be expressed as: (sklearn.metrics, n.d.-e)

$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t$

$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_t^2$

$\hat{m}_t = m_t / (1 - \beta_1^t)$

$\hat{v}_t = v_t / (1 - \beta_2^t)$

$\theta_t = \theta_{t-1} - \alpha * \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$

Where g_t is the gradient at time step t, m_t and v_t are the first and second moment estimates; β_1 and β_2 are the hyperparameters; α is the learning rate, and ε is a small constant to add for numerical stability.

B) To avoid overfitting, I used the following methods:

- Early stopping: The code uses an EarlyStopping callback: (Fig 9)(sklearn.metrics, n.d.-e)

```
# Define early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights
```

Fig 9:Source- Implemented in Jupyter Notebook

This stops training when the validation loss stops improving. If the validation loss doesn't improve for 10 consecutive epochs, training is halted, and the best weights are restored. This prevents the model from continuing to fit the training data when generalization performance starts to degrade.This means that training will stop when validation loss does not improve for a certain number of epochs. It avoids overfitting the model on the training dataset by stopping it before it memorizes the training data.

- Dropout: There are two dropout layers here with a rate of 0.2. Dropout randomly shuts off a fraction of neurons during training so that the model doesn't get too dependent on any particular feature or neuron. This can, in turn, be mathematically represented as:

  h_dropout = h * mask, where (sklearn.metrics, n.d.-a)

  Here, p = 0.8 (1 – dropout rate), and 'mask' is a binary vector such that each element equals 1 with probability p and 0 with probability 1-p.

  I applied dropout in the hidden layers. During training, this is the random setting to zero a fraction of input units. The method prevents overfitting by reducing the co-adaptation of input units.

- Regularization: I used L2 regularization (sklearn.metrics, n.d.-b)to penalize the model for large values of weights. The effect will be to avoid overfitting by adding this penalty term to the loss function, so it will help the model to have small weight values. Mathematically, the L2 regularization term would look something like this: $L = L_0 + \alpha * \Sigma(w_i)^2$

where L is the loss function with regularization, L_0 is the loss function without regularization, α is the regularization parameter, and w_i the weight value for the i-th weight matrix.

- Feature Selection: Selected a subset of the most relevant input features to reduce the risk of overfitting by using recursive feature elimination (RFE) to select the top k features.

- Validation Split:

  20% of the training data is used as a validation set: (Fig 7)

  This will allow the model to be evaluated on unseen data during training to identify and avoid overfitting.

  - Standardization of features using StandardScaler

```
# Normalize the features
scaler = StandardScaler()
x_scaled = scaler.fit_transform(x)
```

Fig 10: Source- Implemented in Jupyter Notebook

The features are transformed to have zero mean and unit variance:

$x\_scaled = (x - \mu) / \sigma$

Where μ is the mean and σ is the standard deviation for every feature.

Standardization prevents features with large ranges from dominating during the update of network weights in backpropagation and can also give some speedup in convergence for the model.

- Relatively Small Network:

The size of the network is medium: $64 \cdot 32 \cdot 16$ neurons, which reduces the capacity of the model and thus prevents overfitting.

All these techniques will regularize the model to force it to learn general patterns that are more robust and have better generalization instead of converging faster to the training set.

C) Class Imbalance in the labels is addressed by using the following techniques:

- Resampling: Balanced the class distribution in the training data by applying different resampling techniques. In this case, applied random oversampling to increase the number of instances of the minority class, and random undersampling to decrease the number of instances of the majority class.
- Cost-sensitive learning: I used this to give more cost for miss classifying the minority class. The idea here is to avoid biasing the model toward the majority class, it improves the model's performance on the minority class.

The cost-sensitive learning can be represented mathematically by this:

$L = -(y * \log(p) + a*(1-y) * \log(1-p)) * C$

where L is the cost-sensitive loss function, y is the true label, p the predicted probability, and C the cost matrix.(sklearn.metrics, n.d.-f)

In summary, the MLP model with early stopping, dropout, and L2 regularization were used to prevent overfitting. Then, resampling techniques and cost-sensitive

learning were applied to deal with class imbalance. These methods really improved the performance of the model and kept it from falling into overfitting and bias to the majority class.

3. Features & Labels

Features & Labels used in this model are groundfrost_2, psl_2, pv_2, rainfall_2, sfcWind_2, snowLying_2, sun_2, and tas_2, and hurs_2 as label target variable.

The features and labels were selected based on the correlation/heatmap, hist-plots, box-plots,scatter plots and by whiskers in the boxplot outliers.

These features were chosen because of their strong correlation with the relative humidity value. We computed the correlation coefficient between each feature and the value of relative humidity, then picked from those features that had the highest correlation coefficient. The features were picked from the given data because it is relevant for predicting the relative humidity in the next month. Concretely, groundfrost_2, psl_2, pv_2, rainfall_2, and sfcWind_2 are meteorological:
These are some of the features known to affect relative humidity. Snow lying and sunshine duration are also added as these can affect the availability of moisture in the air and the rate of evaporation respectively. Another factor considered is

temperature as this has a strong inverse relationship with relative humidity. The label used in the model is hurs_2 which is relative humidity in the next month. It is the measure of the amount of moisture in the air in regard to the maximum air can hold at a given temperature. This is one of the more critical variables taken in weather forecasting and climatological study. This feature and label were constructed from the raw data by simply selecting columns and shifting one month forward to get the time-series dataset. Specifically, it obtained the features and label for a given month from data of the previous month. That time-shifted dataset was used in training the model.

4. Preprocessing

The following steps of preprocessing were done to make the data ready for building the model.

- Handling Missing Values: In the dataset, missing values were replaced by the mean value of the respective feature. This was to avoid bias in the model and to ensure that the model is trained on a complete dataset. Additionally KNN imputer was also implemented to handle missing values

- Data Cleaning: First in preprocessing, I have cleaned the data, removing the missing or duplicate values. I used the dropna() function to drop the rows with missing values and drop duplicates() to drop the duplicate rows.

```
df.duplicated().sum()
```

```
24650
```

```
df.drop_duplicates()
```

```
       x_coord  y_coord       year  groundfrost_1  groundfrost_2  \
0          1.0      1.0 1899-12-31       8.240716       8.191304
17         2.0      2.0 1899-12-31       8.240716       8.191304
34         3.0      3.0 1899-12-31       8.240716       8.191304
51         4.0      4.0 1899-12-31       8.240716       8.191304
58         4.0      4.0 1899-12-31      12.937533      13.568946
…          …        …          …              …              …
33167     20.0     20.0 1900-01-07      14.652954      19.206108
```

Data Scaling: This was the second step of preprocessing applied across all features. Scaling is important to make sure that each feature has a mean of 0 and a standard deviation of 1, otherwise known as unit variance. This will help the performance of the model. I used the StandardScaler function from the scikit-learn library to scale the data.

- Data Split The third step in preprocessing was to split data into training and testing sets. In this case, I used the train_test_split function from the sklearn-learn library to split up data into a training set and a test set with a test size of 0.3, and a random state of 42.

- Scaling and Normalization: The features were scaled and normalized to a mean of 0 and a standard deviation of 1 so that to avoid the domination of features with large ranges on the model and also to improve the performance of the model.

- Encoding Categorical Variables: The binary features Groundfrost_2 and SnowLying_2 were already in 0s and 1s therefore, further encoding was not necessary.

- Outlier Removal: The outliers of the dataset were checked in a box-plot visualization, outlier removal was done.

- Data Transformation: This was the fourth step in preprocessing the transformation of data into a form that would be usable by the model. created, with the help of the function create_time_features, time-shifted features against which to forecast, and an MLP model through the create_mlp_model function.The create_time_features function accepts input data and target data, then it creates time features so it can make a forecast by just shifting the input data one month ahead. It will return input data with time-shifted features and the target data itself. The create_mlp_model function creates an MLP model of an input shape by the provided data, itself having three hidden layers and an output layer. It is realized by the creation of the Sequential model from the Keras library for creating the model and adding three Dense layers: one with 64 neurons, another with 32 neurons, the last one with 16 neurons, plus an output layer with a single neuron. It also uses the Adam optimizer and the mean squared error loss function while compiling the model.

In other words, some of the preprocessing steps done on the data to build the model were data cleaning, handling missing values, duplicated, scaling of data, splitting, and data transformation. This is all in processes so that the data is in a form that a model can use, improving model performance.

References

sklearn.metrics. (n.d.-a). *1.11. Ensembles: Gradient boosting, random forests, bagging, voting, stacking*. Scikit-Learn. Retrieved 13 August 2024, from https://scikit-learn/stable/modules/ensemble.html

sklearn.metrics. (n.d.-b). *1.17. Neural network models (supervised)*. Scikit-Learn. Retrieved 13 August 2024, from https://scikit-learn/stable/modules/neural_networks_supervised.html

sklearn.metrics. (n.d.-c). *6.3. Preprocessing data*. Scikit-Learn. Retrieved 13 August 2024, from https://scikit-learn/stable/modules/preprocessing.html

sklearn.metrics. (n.d.-d). *Mean_absolute_error*. Scikit-Learn. Retrieved 13 August 2024, from https://scikit-learn/stable/modules/generated/sklearn.metrics.mean_absolute_error.html

sklearn.metrics. (n.d.-e). *MLPRegressor*. Scikit-Learn. Retrieved 13 August 2024, from https://scikit-learn/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

sklearn.metrics. (n.d.-f). *Post-tuning the decision threshold for cost-sensitive learning*. Scikit-Learn. Retrieved 13 August 2024, from https://scikit-learn/stable/auto_examples/model_selection/plot_cost_sensitive_learning.html

sklearn.metrics. (n.d.). *Root_mean_squared_error*. Scikit-Learn. Retrieved 13 August 2024, from https://scikit-learn/stable/modules/generated/sklearn.metrics.root_mean_squared_error.html

sklearn.metrics. (n.d.-g). *R_regression*. Scikit-Learn. Retrieved 13 August 2024, from https://scikit-learn/stable/modules/generated/sklearn.feature_selection.r_regression.html