Weather Prediction Using Machine Learning

This document outlines key decisions for building and deploying a weather prediction model using machine learning. The main aspects covered are training-test data split, acceptable error rates, the chosen machine learning algorithm, outlier handling, dimensionality reduction, and model versioning. The implementation is done in Python, following best practices, and key steps are visualized for clarity.

1. Training/Test Data Split (80%-20%)

Decision:

The dataset is split into 80% for training and 20% for testing the model. This split ensures that the model has sufficient data to learn patterns while reserving enough for evaluation on unseen data.

Justification:

An 80%-20% split is standard practice in machine learning and helps in both avoiding overfitting and assessing model generalization performance on test data.

from sklearn.model\_selection import train\_test\_split

# Train-test split (80%-20%)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_clean, y\_clean, test\_size=0.2, random\_state=42)

2. Acceptable Error Rates in Prediction

Decision:

The acceptable error rates for the model are measured using:

- Mean Absolute Error (MAE): Expected to be less than 10.

- Mean Squared Error (MSE) and Root Mean Squared Error (RMSE): RMSE should ideally remain below 15.

- R-squared (R²): Closer to 1 indicates a good fit.

Justification:

These metrics help understand both the average error magnitude (MAE) and the variance in errors (MSE, RMSE). R² explains how well the model fits the data.

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score

import numpy as np

# Prediction on test data

y\_pred = rf\_model.predict(X\_test)

# Error metrics

mae = mean\_absolute\_error(y\_test, y\_pred)

mse = mean\_squared\_error(y\_test, y\_pred)

rmse = np.sqrt(mse)

r2 = r2\_score(y\_test, y\_pred)

# Print performance

print(f"MAE: {mae:.2f}, MSE: {mse:.2f}, RMSE: {rmse:.2f}, R²: {r2:.2f}")

3. Machine Learning Algorithm Used

Decision:

We selected the \*\*Random Forest Regressor\*\* for weather prediction due to its strong performance on tabular data and its ability to capture non-linear relationships between features. The model aggregates multiple decision trees and averages their predictions, making it less prone to overfitting.

Justification:

Random Forest is a robust algorithm, especially for regression tasks with complex feature interactions. It also provides interpretability through feature importance scores

from sklearn.ensemble import RandomForestRegressor

# Initialize and train Random Forest model

rf\_model = RandomForestRegressor(random\_state=42, n\_estimators=100)

rf\_model.fit(X\_train, y\_train)

4. Outlier Removal

Decision:

Outliers are detected and removed using the Z-score method. Rows with a Z-score greater than 3 are treated as outliers and excluded from the dataset.

Justification:

Outliers can disproportionately influence the model’s predictions. Removing them enhances model performance and generalization.

from scipy import stats

# Detect and remove outliers

z\_scores = np.abs(stats.zscore(X\_encoded))

X\_clean = X\_encoded[(z\_scores < 3).all(axis=1)]

y\_clean = y.loc[X\_clean.index]

5. Dimensionality Reduction

Decision:

To handle potentially high-dimensional data, \*\*Principal Component Analysis (PCA)\*\* or feature importance scores from Random Forest can be used to reduce the number of features without losing significant information.

Justification:

Dimensionality reduction improves model training efficiency and reduces overfitting. In this case, Random Forest’s feature importance scores can directly identify which features contribute most to the model’s performance.

from sklearn.decomposition import PCA

# Apply PCA for dimensionality reduction

pca = PCA(n\_components=10)

X\_reduced = pca.fit\_transform(X\_clean)

6. Optional: User Interface for Testing

A simple user interface (UI) can be created in Python using the `Tkinter` library to allow users to input weather conditions and predict the temperature in real time.

Code for a Basic UI:

import tkinter as tk

def predict():

temp = rf\_model.predict([user\_inputs]) # Simulated inputs

label\_result.config(text=f'Predicted Temperature: {temp[0]:.2f}')

# Tkinter setup

root = tk.Tk()

root.title('Weather Prediction')

label = tk.Label(root, text='Weather Prediction System')

label.pack()

btn\_predict = tk.Button(root, text='Predict Temperature', command=predict)

btn\_predict.pack()

label\_result = tk.Label(root, text='Predicted Temperature:')

label\_result.pack()

root.mainloop()

7. Testing and Documentation

To ensure code correctness and prevent bugs, we will write unit tests for each function using the ‘unittest’ framework. Documentation for tests will be generated to provide clarity on how they were designed and passed.

Test Code:

import unittest

class TestWeatherPrediction(unittest.TestCase):

def test\_data\_split(self):

self.assertEqual(X\_train.shape[0], 4000) # Example values

self.assertEqual(X\_test.shape[0], 1000)

def test\_mae(self):

self.assertLess(mae, 15) # Ensuring MAE is below threshold

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

Conclusion:

This design document outlines the end-to-end approach for building a weather prediction system, including data processing, model selection, versioning, and performance evaluation. The model uses Random Forest as the predictive algorithm and evaluates performance using MAE, MSE, RMSE, and R² metrics. The code is written in Python and adheres to best practices, with diagrams and code versioning to ensure reproducibility.

Complete code:

# Import necessary libraries

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score

from scipy import stats

import numpy as np

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

import tkinter as tk

import unittest

# 1. Load the dataset

file\_path = '/data/weather\_prediction\_dataset.csv'

weather\_data = pd.read\_csv(file\_path)

# 2. Data Preprocessing

# Drop the 'Date' column if it's not needed for prediction

weather\_data\_clean = weather\_data.drop(columns=['Date'])

# Separate features (X) and target (y)

X = weather\_data\_clean.drop(columns=['Temperature'])

y = weather\_data\_clean['Temperature']

# One-hot encoding for categorical features: 'City', 'WindDirection', 'Season'

X\_encoded = pd.get\_dummies(X, columns=['City', 'WindDirection', 'Season'], drop\_first=True)

# 3. Outlier Detection and Removal using Z-score method

z\_scores = np.abs(stats.zscore(X\_encoded))

X\_clean = X\_encoded[(z\_scores < 3).all(axis=1)]

y\_clean = y.loc[X\_clean.index]

# 4. Train-Test Split (80%-20%)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_clean, y\_clean, test\_size=0.2, random\_state=42)

# 5. Apply Dimensionality Reduction using PCA (optional)

pca = PCA(n\_components=10)

X\_train\_pca = pca.fit\_transform(X\_train)

X\_test\_pca = pca.transform(X\_test)

# 6. Model Training: Random Forest Regressor

rf\_model = RandomForestRegressor(random\_state=42, n\_estimators=100)

rf\_model.fit(X\_train\_pca, y\_train)

# 7. Model Prediction

y\_pred = rf\_model.predict(X\_test\_pca)

# 8. Calculate Performance Metrics

mae = mean\_absolute\_error(y\_test, y\_pred)

mse = mean\_squared\_error(y\_test, y\_pred)

rmse = np.sqrt(mse)

r2 = r2\_score(y\_test, y\_pred)

# Print performance metrics

print(f"Model Performance Metrics:")

print(f"Mean Absolute Error (MAE): {mae:.2f}")

print(f"Mean Squared Error (MSE): {mse:.2f}")

print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")

print(f"R-Squared (R²): {r2:.2f}")

# Optional: Visualizing Feature Importance

importances = rf\_model.feature\_importances\_

indices = np.argsort(importances)[::-1]

# Plot the feature importance

plt.figure(figsize=(10, 6))

plt.title("Feature Importance")

plt.bar(range(X\_train.shape[1]), importances[indices], align="center")

plt.xticks(range(X\_train.shape[1]), X\_train.columns[indices], rotation=90)

plt.tight\_layout()

plt.show()

# Optional: User Interface for Weather Prediction

def predict():

user\_inputs = np.array([float(entry.get()) for entry in entries])

user\_inputs\_pca = pca.transform([user\_inputs])

temp = rf\_model.predict(user\_inputs\_pca)

label\_result.config(text=f'Predicted Temperature: {temp[0]:.2f}')

# Create a simple UI with Tkinter

root = tk.Tk()

root.title('Weather Prediction System')

# Labels and Entries for user inputs (replace with appropriate number of features)

labels =weather\_data\_clean[:5]

entries = []

for label\_text in labels:

label = tk.Label(root, text=label\_text)

label.pack()

entry = tk.Entry(root)

entry.pack()

entries.append(entry)

btn\_predict = tk.Button(root, text='Predict Temperature', command=predict)

btn\_predict.pack()

label\_result = tk.Label(root, text='Predicted Temperature:')

label\_result.pack()

root.mainloop()

# 9. Unit Testing for the Weather Prediction Model

class TestWeatherPrediction(unittest.TestCase):

def test\_data\_split(self):

# Test if the train-test split resulted in the expected number of samples

self.assertEqual(X\_train.shape[0], 4000) # Replace with actual expected sizes

self.assertEqual(X\_test.shape[0], 1000)

def test\_mae(self):

# Ensure that the Mean Absolute Error is below an acceptable threshold

self.assertLess(mae, 15)

def test\_pca(self):

# Test that PCA reduced the dimensions correctly

self.assertEqual(X\_train\_pca.shape[1], 10)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()