Applied Machine Learning (ICS-5110) Coursework Report

Ahnaf Mozib Samin[1]
ahnaf.samin.22@um.edu.mt

Ruan Chaves Rodrigues[1]
ruan.chaves.22@um.edu.mt

Ike Ebubechukwu S.[1]
ebubechukwu.ike.22@um.edu.mt

[1]University of Malta

1 Introduction

The Jena Climate dataset is a time series of weather data recorded at the Weather Station of the Max Planck Institute for Biogeochemistry in Jena, Germany. [13] The dataset includes 14 different measurements, such as air temperature, atmospheric pressure, humidity, and wind direction, that were recorded every 10 minutes over a period of several years. The dataset covers the time period from January 1, 2009 to December 31, 2016.

We plan to use the dataset to predict temperature using the other columns in the dataset that are not directly related to temperature. Therefore, in order to avoid data leakage, we will only work with a subset of the columns in the dataset, as indicated on Table 1

We will implement a random forest, a naive Bayes algorithm and long-short term memory (LSTM) neural network models from scratch for this task and compare them with third-party implementations. This will allow us to determine the effectiveness of our custom implementations and identify any potential improvements.

Dataset column	Description
Date Time	The date and time of the measurement
p (mbar)	Atmospheric pressure (millibar)
T (degC)	**Temperature (degree Celsius)**
Tpot (K)	**Potential temperature (Kelvin)**
Tdew (degC)	**Dew point temperature (degree Celsius)**
rh (%)	Relative humidity (percentage)
VPmax (mbar)	Maximum vapor pressure (millibar)
VPact (mbar)	Actual vapor pressure (millibar)
VPdef (mbar)	Vapor pressure deficit (millibar)
sh (g/kg)	Specific humidity (grams per kilogram)
H2OC (mmol/mol)	Water vapor concentration (millimoles per mole)
rho (g/m**3)	Air density (grams per cubic meter)
wv (m/s)	Wind velocity (meters per second)
max. wv (m/s)	Maximum wind velocity (meters per second)
wd (deg)	Wind direction (degrees)

Table 1: List of columns on the Jena Climate dataset and their descriptions. The boldfaced values indicate the columns that consist of temperature measurements. We keep the **T (degC)** column, while dropping the others.

2 Background

2.1 Selected Techniques

In this section, we will describe the mechanics of the chosen machine learning techniques and discuss some important concepts related to them.

2.1.1 Random Forests

The random forest algorithm is an ensemble machine learning method that uses multiple decision trees to make predictions. Before we can explain the random forest algorithm, it is important to first understand the decision tree algorithm.

Decision Trees To predict temperature measurements, we will use decision trees for regression. Decision trees work by dividing the data into smaller and smaller subsets based on a decision rule. For example, the decision rule might be based on the time of day, such as ”if it is morning, the temperature is likely to be cooler.” [9] Using this rule, the data will be split into two partitions: one for values measured before a certain time, and another for values measured during or after that time. The pivot, or time at which the data is split, is chosen based on a criterion.

The criterion In our decision tree, we choose the best decision rule and pivot at each step based on a criterion, which is a function used to measure the quality of a split. Our chosen criterion is the weighted average of the mean-squared error of the training labels at each partition [5]. We select the decision rule and pivot that minimize this criterion.

The criterion is calculated by the **get_mse** function as follows:

$$\text{MSE}(\mathbf{a}, \mathbf{b}) = \frac{\sum_{\mathbf{i}=\mathbf{1}}^{\mathbf{n}} (\mathbf{a_i} - \mathbf{b_i})^{\mathbf{2}}}{\mathbf{n}}$$

$$\text{partial\_mse}(\mathbf{a}, \mathbf{b}) = \frac{|\mathbf{a}|}{|\mathbf{a}| + |\mathbf{b}|} \times \text{MSE}(\mathbf{a}, \bar{\mathbf{a}}\mathbf{J})$$

$$\text{get\_mse}(\textbf{left}, \textbf{right}) = \text{partial\_mse}(\textbf{left}, \textbf{right}) + \text{partial\_mse}(\textbf{right}, \textbf{left})$$

left and **right** at *get_mse* are the sets of the training labels for the partitions at each step. MSE is the function that computes the mean squared error between a partition x and the all-ones vector J multiplied by the mean of the partition, $\bar{x}$.

This process of splitting the data is repeated until one of the partitions contains the minimum amount of values allowed for a leaf node, or until we reach the maximum tree depth.

Inference Once the training stage is complete, predictions are made for each subset based on the average value of the data points in the final leaf node. This allows the decision tree to make accurate predictions for continuous values, such as temperature, rather than just discrete classes.

Random Forests The random forest algorithm trains multiple decision trees on random subsets of the data. The process of generating these subsets, called bootstrapping or bagging, involves randomly sampling the data with replacement. This procedure has been proven in previous literature to improve the accuracy of a random forest [4].

Predictions are made by aggregating the predictions of each individual tree. In our implementation, we average the predictions of the distinct decision trees in the forest. Since decision trees have low bias and high variance, this aggregation helps to reduce the high variance of individual trees and improve the overall performance of the random forest. [7]

2.1.2 Naive Bayes

Naive Bayes is a probabilistic algorithm that makes use of Bayes' Theorem to predict a class based on a set of features. It makes the naive assumption that the features are independent of each other, which is often not the case in real-world situations. Nevertheless, the algorithm can still be highly effective in many classification tasks.

The basic principle of a Naive Bayes algorithm is to calculate the probability of a class (such as cold, mild, or hot) given a set of predictor variables (such as pressure, humidity, etc.). This probability can be calculated by multiplying the prior probability of the class by the likelihood of the predictor variables given the class, and dividing by the prior probability of the predictor variables.

The likelihood of the predictor variables is calculated by assuming that each predictor variable follows a Gaussian distribution. The mean and variance of this distribution are calculated for each class and each predictor variable, and then used to calculate the likelihood of a given value of the predictor variable given the class.

Once the prior probabilities and likelihoods have been calculated, the algorithm can use them to calculate the posterior probability of each class given a set of predictor variable values. The class with the highest posterior probability is then chosen as the predicted class.

In our implementation, our Naive Bayes classifier uses the Gaussian Distribution formula to calculate the likelihood of a predictor given a class. It also uses the logarithm function to calculate the posterior probability while avoiding underflow.

2.1.3 Long-short Term Memory Network

Long-short Term Memory (LSTM) networks are a variant of recurrent neural networks (RNNs) and are widely used in the field of natural language and speech processing where sequential data is present. A recurrent neural network processes an input (e.g. a token or a number) in a specific time-step concatenated with a state vector from the previous time-step and outputs another state vector for the next time step in a recurrent fasion. The issue with traditional RNN is that it is not capable of coping with vanishing gradient that is the exponential decay of gradient propagated back through the network [2].

Like a standard RNN, an LSTM network works in a supervised learning fashion by getting annotated training sequences and being optimized using a gradient descent algorithm that computes the gradients through back-propagation in order to update the weights of the hidden layers. However, the structure of an LSTM unit is different from standard recurrent units and is composed of a cell state and three gates (an input gate, output gate and forget gate). These gates regulate the flow of information passed to the cell state. The cell states are capable of acting as a memory cell and can preserve information for long sequences.

2.2 Rescaling and normalisation

Both rescaling and normalization are important steps in preparing the data for machine learning. They help to ensure that the data is in a suitable format for the algorithms to work with and can improve the performance of the model. In addition, these techniques are also important for principal component analysis (PCA), a common dimensionality reduction technique that will be explored in the next sections.

Rescaling Rescaling is a preprocessing step that adjusts the range of the data so that all the values fall within a specific interval. In our experiments, we implemented min-max rescaling:

$$\text{scaling}(x) = \frac{x - \min(x)}{\max(x) - \min(x)}$$

The min-max scaling formula takes an array x and scales its values to the range [0, 1]. Min-max scaling is an important preprocessing step for machine learning in general because it ensures that the data is in a suitable range for the algorithms to work with. Many machine learning algorithms, including neural networks and support vector machines, are sensitive to the scale of the input data. If the data is not properly scaled, these algorithms may perform poorly or even fail to converge. [3]

However, min-max scaling may not make a significant difference for algorithms like random forests, which are not sensitive to the scale of the data. This is because random forests use decision trees as their building blocks, and decision trees are not affected by the scale of the data because they are optimized on one feature at a time. As a result, min-max scaling is expected to have a neutral effect on the performance of a random forest model.

Mean normalisation Mean normalisation is a technique that scales the data to have a mean of 0 and a standard deviation of 1. This can help to improve the performance of machine learning algorithms by making the data more amenable to processing. In our experiment, we normalized the data around the standard deviation as follows:

$$\text{normalisation}(x) = \frac{x - \bar{x}}{\text{max}(x) - \text{min}(x)}$$

where x is the input array and $\bar{x}$ is the mean of this array.

2.3 Cross validation

K-fold cross validation is a technique that involves dividing the dataset into k subsets, called folds, and training the model k times, each time using a different fold as the validation set and the remaining folds as the training set.

K-fold cross validation is important because it helps to prevent overfitting, which is when a model performs well on the training data but poorly on unseen data [1]. By training the model on different subsets of the data and evaluating its performance on each subset, we can get a better sense of how the model will perform on unseen data. This can help to ensure model robustness and generalization capabilities.

2.4 Dimensionality reduction

Dimensionality reduction can help to remove redundant or irrelevant features from the data and improve the performance of the model. There are many different techniques for dimensionality reduction, each with its own strengths and limitations. In this section, we will focus on two widely used techniques: feature selection with the filter method and PCA.

Feature selection Feature selection involves choosing a subset of the available features to use in the model. The filter method is an approach to feature selection ranks features based on a statistical measure. It reduces the dimensionality of the data by removing highly correlated or redundant features.

In our implementation, we pick the Pearson Correlation Coefficient as our statistical measure and drop one highly correlated feature pair while keeping the other. Our implementation of the Pearson correlation coefficient can be described by the formula:

$$\rho = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

Where x and y are the input arrays and $\bar{x}$ and $\bar{y}$ are the means of the input arrays.

Principal Component Analysis (PCA) Principal Component Analysis (PCA) is a linear transformation technique that is used to identify patterns in data and reduce the dimensionality of the data by projecting it onto a lower-dimensional space.

The core idea behind PCA is to identify the directions in which the data varies the most and to project the data onto a new set of axes that are aligned with these directions. The new axes, called principal components, are orthogonal to each other and are ordered in terms of the amount of variance they capture from the original data.

To perform PCA, we first need to compute the covariance matrix of the data. The covariance matrix is a square matrix that measures the degree to which two variables are linearly related. The covariance matrix can be described as:

$$\Sigma = \begin{bmatrix} \sigma_{1,1} & \sigma_{1,2} & \dots & \sigma_{1,n} & \sigma_{2,1} & \sigma_{2,2} & \dots & \sigma_{2,n} & \vdots & \vdots & \ddots & \vdots & \sigma_{n,1} \\ \sigma_{n,2} & \dots & \sigma_{n,n} \end{bmatrix}$$

where $\sigma_{i,j}$ is the covariance between the i^{th} and j^{th} variables. For instance, if we have two variables, x and y, the covariance matrix would be:

$$\Sigma = \begin{bmatrix} \sigma_{x,x} & \sigma_{x,y} & \sigma_{y,x} & \sigma_{y,y} \end{bmatrix}$$

where $\sigma_{x,x}$ is the variance of x, $\sigma_{y,y}$ is the variance of y, and $\sigma_{x,y}$ is the covariance between x and y.

The principal components of our data will be the eigenvectors of this covariance matrix, and they are ordered in terms of the corresponding eigenvalues, which indicate the amount of variance they capture from the original data.

To select the adequate number of components, we can loop over the cumulative explained variance array $C(R^2)$ and stop at the index where we find a value larger than a certain threshold. This index will be the sufficient number of components to explain the variance in our data.

The $C(R^2)$ array can be defined as:

$$R^2 = \begin{bmatrix} \frac{\lambda_0}{\sum_{i=1}^{n}\lambda_i} & \dots & \frac{\lambda_n}{\sum_{i=1}^{n}\lambda_i} \end{bmatrix}$$

$$C(R^2) = \begin{bmatrix} \sum_{j=1}^{1} R_0^2 & \dots & \sum_{j=1}^{n} R_n^2 \end{bmatrix}$$

where λ_i is the ith eigenvalue.

The optimal threshold is determined empirically. One common approach is to choose a threshold around 95% [8].

2.5 Quantitative measurements

To quantitatively measure the results of our experiments, we will use the mean squared error as the evaluation metric for the Random Forest and LSTM models, that frame the problem as a regression task.

For the Naive Bayes classifier, we will use accuracy as our metric, as it treats temperature prediction as a classification task between four classes (below zero, cold, mild and hot).

2.5.1 Mean Squared Error (MSE)

The mean squared error is an adequate metric for evaluating the performance of a machine learning model on a regression task because it provides a convenient way to measure the average squared difference between the predicted values produced by the model and the true values from the dataset.

Mathematically, MSE is calculated as the average of the squared differences between the predicted values and the true values:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2$$

where N is the number of examples in the dataset, y_i is the true temperature for the i-th example, and $\hat{y_i}$ is the predicted temperature produced by the model for the same example.

The MSE is a useful metric because it provides an easily interpretable measure of the model's performance, and it is differentiable, which means that it can be used as a loss function for training machine learning models [10].

Additionally, because MSE is an average of the squared errors, it is more sensitive to outliers in the data than other alternative metrics such as MAE (Mean Absolute Error). [6] This is important for temperature prediction as we are particularly interested in models that can capture unusual weather events.

2.5.2 Accuracy

Formally, Classification accuracy has the following definition: It is the ratio of the number of correct predictions to the total number of input samples. In other words, it is the degree to which the measured value is comparable to a standard or true value.

$$\text{Accuracy} = \frac{TN + TP}{TN + FP + TP + FN}$$

where * TP equals True Positives i.e. positive classes that are correctly predicted as positive. FP equals False Positives i.e negative classes that are falsely predicted as positive. TN equals True Negatives i.e. negative classes that are correctly predicted as negative. And FN equals False Negatives i.e positive classes that are falsely predicted as negative. [11] When working in data-driven industries like science, measuring calculation accuracy is essential to ensuring accurate results. Professionals may make sure they are collecting exhaustive and comprehensive data by making use of precise measures.

3 Experiments

In this section, we will describe the steps we took to process the data and the experiments we carried out.

3.1 Preprocessing

To preprocess the dataset, we first need to drop the redundant columns ”Tpot (K)” and ”Tdew (degC)”, as they represent the same physical quantity (temperature) as the ”T (degC)” column. This is important because redundant columns can cause data leakage, which can lead to overfitting and poor performance of machine learning models.

Next, we define a function that converts the date and time strings in the dataset into timestamps, days of the year, and seconds since midnight. This allows us to capture all three factors that can affect the temperature at a given point: the date, the season, and the time of the day.

Finally, we split the data into two parts: one for training and development, and one for testing. We then use a factory to create a nested dictionary structure to store the data from the DataFrame. The training and development data and the testing data are stored in different keys, and the raw data and processed data are stored in different subkeys. This allows us to easily access and manipulate the data for different purposes.

3.2 Implementation: LSTM Network

In this project, we implement an LSTM network with one/two hidden layers from the first principals without using any pre-defined LSTM function. The purpose of this study is to examine how well an LSTM network works with different amounts of sequential time-series data and various input feature types.

We take each feature from our dataset in each time step and concatenate it with both cell state and hidden state. The cell state and the hidden state are initialized with a single zero value. The number of features depends on the normalization technique we are using and thus the number of time-steps also varies. After concatenating the input with the cell state and hidden state, we perform the following operations in each time step and obtain a list of interim states.

$$g_{t+1} = \text{sig}(\text{conc}(s_t, x_{t+1})W + b) \tag{1}$$

$$c_{t+1} = g_t^i \times \tanh(\text{conc}(s_t, x_{t+1})W + b) + g_t^f \times c_t \tag{2}$$

$$s_{t+1} = g_t^o \times \tanh(c_t) \tag{3}$$

Here, g_t^i, g_t^f, g_t^o are the input gate, forget gate, and output gate, respectively. x_t, s_t and c_t refer to the input vector, state vector and cell vector at t time step, accordingly. Weights and biases are represented by W and b, respectively.

These interim states from the first LSTM layer are then passed as input after concatenating them with the hidden state and the cell state in each time step and the same operations are performed to get the final states. At last, a linear layer takes the final states where each state has the same size of the

number of features and provides a value as output. After the forward pass, MSE is calculated and the error is back-propagated to update the weights using gradient descent algorithm.

In our implementation, the LSTM is trained using Adam optimizer on three subsets of the training data (e.g. 10k, 20 and 30k data). Due to the large size of the dataset and unavailability of computational resources, we are unable to train LSTM with the full training dataset. We hypothesize that with more data, our LSTM network will perform better. The learning rate is set to 1e-4. The models are evaluated on the dev set after each epoch and we only take the checkpoint with the best dev set result for final evaluation on the test set.

Apart from investigating several input features for different sizes of training data, we also compare the performance between one LSTM layer and two LSTM layers. Furthermore, we perform experiments for different hidden sizes in each LSTM layer. However, we find that with increasing hidden sizes, the performance tends to decrease and so, we only keep the hidden size to 1 in all our experiments.

3.3 Implementation: Naive Bayes

In our **NaiveBayesClassifier** class, the fit method is used to train the model by calculating the mean, variance and prior probability of the classes in the training data. The predict method takes in new data and uses these statistics to calculate the likelihood of each class for each new observation, and then classifies the observation based on the class with the highest likelihood.

The likelihood is calculated using a Gaussian density function. The mean and variance for this density function are estimated from the training data by the **calc_mean_var** method, which groups the data by class and then calculates the mean and variance for each group.

The prior probability of a class is calculated by the **calc_prior** method, which groups the data by class and then counts the number of samples in each group. The prior probability for each class is then calculated by dividing the number of samples in each group by the total number of samples.

Once the mean, variance, and prior probability are calculated, the predict method uses them to calculate the likelihood of each class for each new observation. The **calc_posterior** method is used to calculate the likelihood of each class using the Gaussian density function. This method takes in an observation as input and calculates the likelihood of the observation belonging to each class using the mean, variance, and prior probability.

The predict method uses the **calc_posterior** method to calculate the likelihood of each class for each new observation and then classifies the observation based on the class with the highest likelihood, resulting in the final labels produced by the model.

3.4 Implementation: Random Forests

The implementation of Random Forest presented in this report is an ensemble of decision tree models. Each decision tree is trained on a randomly selecting a subset of the original data with replacement.

Input feature	10k samples		20k samples		30k samples	
	1 layer	2 layers	1 layer	2 layers	1 layer	2 layers
raw+norm	11.66	12.10	11.65	12.12	11.81	12.88
raw+norm+pca	12.44	14.97	11.70	14.74	11.90	14.33
fs+norm	9.70	9.80	9.24	9.96	9.31	12.21
fs+norm+pca	33.00	67.12	33.24	41.4	32.66	68.6

Table 2: MSE calculated for LSTM models trained with four input feature types, one and two LSTM layers and three subsets of training data including 10k, 20k and 30k data. MSE is measured on the test set.

The ‘DecisionTree‘ class is initialized with the input data, the target values, and the row indexes of the data to be used in building the tree. The minimum leaf size and maximum depth of the tree are also specified as hyperparameters.

The ‘generate_children‘ method is used to recursively build the tree by dividing the input data into left and right branches at each node, based on the value of a chosen pivot. The pivot is selected by minimizing the criterion. The tree continues to grow until the maximum depth is reached or until there are no further improvements in our criterion by dividing the data.

Once the tree is built, the ‘predict‘ method can be used to make predictions for new data. The method traverses the tree by following the left or right branches depending on the value of the input data at the chosen pivot. When it reaches a leaf node, the predicted value is the mean of the target values in the leaf node.

We train multiple ‘DecisionTree‘ models and return their ensemble as a Random Forest model.

3.5 Experiments: Long-short Term Memory Network

Table 2 presents the MSE results of our experiments with LSTM. Three subsets of the training dataset with 10k, 20k and 30k samples have been chosen to examine the relationship between the performance with regard to dataset sizes. With each of the subsets, LSTMs with different numbers of hidden layers (e.g. one layer and two layers) have been trained. Furthermore, we utilize four types of input features to train the LSTM-based models. It is found that we get the lowest MSE of 9.24 with the feature selection technique and normalized features and one layer of LSTM layer, trained with 20k samples. Overall, the performance of LSTM with the feature selection and normalized features is better than the rest of the features. We also find that applying PCA is not beneficial for LSTMs. We obtain the highest MSE when we feed the combination of feature selection, normalized and PCA-based features to the LSTMs.

In terms of number of layers, we observe that single layer LSTMs perform better in all cases compared to their two-layer-LSTM counterparts. We argue that single layer LSTM with less parameters is suitable for this dataset, since we have relatively less number of features (e.g. we have only at most 15 features per sample).

In our experiments, the sizes of training dataset have little impact on the performance of LSTMs. It contradicts with our initial hypothesis based on deep

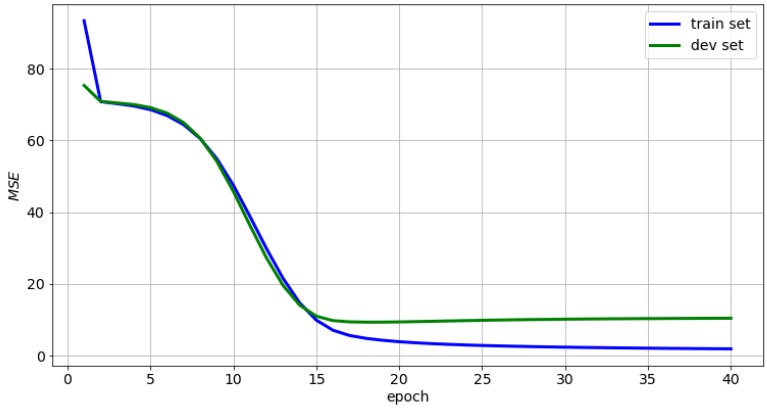

Figure 1: MSE calculated after each epoch on the train and dev sets for the model with one LSTM layer, fs+norm input features and 20k training samples.

learning methods getting higher accuracy with more data. However, further experiments are required with varied sizes of dataset from 10k samples upto the full dataset of 300k samples to investigate more into this direction.

Figure 1 shows the MSE after each epoch on the train and dev sets. The model is trained with a single LSTM layer, feature selection and normized input features and 20k training samples. As epoch increases, the MSE on both train and dev sets decreases initially, however, at approximately epoch 15, the MSE on the dev set start increasing at a very slow rate.

3.5.1 Comparison with third-party libraries: Long-Short Term Memory Network

We choose our best-performing setting from the previous experiments with 20k training samples, single LSTM layer and fs+norm features and train it using the third party LSTM implementation of 'LSTMCell' provided by PyTorch. We get an MSE of 9.29 using the pre-built LSTM implementation which is very similar to the MSE of 9.24, which we obtain from our own implementation.

3.6 Experiments: Random Forest

The performance of random forests was evaluated through two sets of experiments. In the first set, k-fold cross-validation was used to assess the random forest model from sklearn under various scenarios. In the second set, the effects of hyperparameter optimization on the performance of our own random forest algorithm were investigated.

3.6.1 K-fold cross validation

In this experiment, we used k-fold cross-validation to evaluate the performance of a Random Forest model implemented in sklearn. The scenarios we consid-

ered were:

- Using the original training data (**raw**)
- Using the training data after feature selection and normalisation (**fs+norm**)
- Using the training data after feature selection, normalisation and PCA (**fs+norm+pca**)
- Using the training data after normalisation (**raw+norm**)
- Using the training data after normalisation and PCA (**raw+norm+pca**)

We computed metrics such as mean squared error and Pearson's correlation coefficient to evaluate the performance of the model under each scenario. We then used these metrics to create a summary of the model's performance, which is shown on Table 3.

Looking at the results, we can see that rescaling and normalisation had little impact on the model's performance. This is because decision trees, which are used in random forests, only consider one feature at a time when making a decision, allowing them to perform well even when the features have different scales and distributions.

The use of PCA had a negative effect on the model's performance as measured by mean squared error. Feature selection was able to slightly minimize the negative impact of performing PCA on our data.

3.6.2 Hyperparameter optimization

In this experiment, we investigate the effects of hyperparameter optimization on the performance of the random forest model that we have implemented from scratch. We focus on three distinct hyperparameters: sample size, maximum tree depth and number of estimators. For each hyperparameter, we run experiments with different values and compare the results in terms of mean squared error.

Sample size Our results indicate that increasing sample size above 100 is not beneficial for shallow decision trees (with maximum depth less or equal to two). In fact, the mean squared error starts to increase for sample sizes larger than 100 records.

We run experiments with the following sample sizes: 10, 50, 100, 500, 1000, 5000, 10000. The results are shown in Table 4.

Maximum tree depth We also explored how increasing the maximum depth of our random forest trees impacts the model's performance. To conduct this experiment, we first defined a range of maximum tree depths to explore, including 2, 4, 8, and 16. We then trained multiple random forest models, each with a different sample size (100 and 1000), and evaluated their performance using the MSE metric.

The results are shown on Table 5, which indicates the MSE for each combination of maximum tree depth and sample size. The results of this experiment indicate that allowing trees to go deeper results in a significant improvement in

	mse			pearson		
	mean	std	median	mean	std	median
dataset						
fs	8.388	1.419	8.039	0.930	0.015	0.929
fs+norm	8.388	1.419	8.039	0.930	0.015	0.929
fs+norm+pca	13.203	2.922	11.652	0.894	0.021	0.897
raw	8.388	1.419	8.039	0.930	0.015	0.929
raw+norm	8.388	1.419	8.039	0.930	0.015	0.929
raw+norm+pca	14.410	2.012	13.620	0.883	0.022	0.885

Table 3: the mean, standard deviation, and median of two performance metrics for a Random Forest model evaluated on the Jena Weather Dataset under different scenarios. The performance metrics are the mean squared error (MSE) and the Pearson Correlation Coefficient (pearson).

size	mse	pearson
10	43.629	0.813
50	11.202	0.935
100	5.173	0.961
500	6.165	0.954
1000	6.629	0.949
5000	7.478	0.942
10000	7.775	0.940

Table 4: Results of hyperparameter optimization for sample size on the random forest model at a fixed maximum tree depth of 2. The mean squared error (mse) and pearson correlation coefficient (pearson) are shown for different sample sizes. Increasing sample size above 100 does not improve performance for shallow decision trees.

the model's mean squared error (MSE), with an improvement of two orders of magnitude. This suggests that increasing the maximum depth of our trees is an effective way to improve the performance of our model.

Number of estimators we ran two tests with different numbers of estimators, with all other hyperparameters held constant at the best values found so far.

For the first test, we used 10 estimators. This resulted in a mean squared error (MSE) of 0.070864. For the second test, we used 100 estimators. This resulted in a slightly worse MSE of 0.0798.

These results suggest that increasing the number of estimators does not automatically improve the performance of the random forest. The optimal number of estimators may therefore depend on the data characteristics and the specific performance metrics that are most important for a given application.

3.6.3 Comparison with third-party libraries: Random Forest

The results of our experiment indicate that our implementation of a random forest model performs well on the Jena Weather Dataset. We evaluated the performance of our model using the mean squared error (MSE) metric, and found that it achieved an MSE of 0.084.

To compare the performance of our model with that of other implementations, we also evaluated the performance of the random forest model implemented in the scikit-learn (sklearn) library.

Both models were evaluated under the best hyperparameters found on our model. We found that the sklearn model achieved an MSE of 0.0200, which is superior to the performance of our model.

Despite this, our model performed well at shallow tree depths, outperforming the sklearn model in this regard. However, as the depth of the trees increased, the performance of our model began to lag behind third-party libraries, likely due to the complex optimizations implemented in the sklearn model. This is well represented on Table 6.

3.7 Experiments: Naive Bayes

3.7.1 Comparison with third-party libraries

In a third-party implementation of Naive Bayes using the scikit-learn library, feature selection and normalization greatly improved the results. As shown in the Table 7, the accuracy of the model increased significantly when these techniques were applied.

However, for our own implementation, the best results were achieved using the original data without any feature selection or normalization, with an accuracy of 0.8951.

mse	pearson	max_depth	sample_size
7.080	0.950	2	100
6.731	0.948	2	1000
2.496	0.985	4	100
0.241	0.998	4	1000
1.830	0.988	8	100
0.075	**0.999**	**8**	**1000**
1.923	0.987	16	100
0.071	**0.999**	**16**	**1000**

Table 5: the mean squared error (MSE) and Pearson correlation coefficient of our random forest model with different maximum tree depths and sample sizes. Boldface indicates the two best options. The results indicate that increasing the maximum depth of our trees significantly improves the performance of the model.

Model	Setup	MSE
sklearn	shallow trees	8.156
	deeper trees (optimized)	0.020
our model	shallow trees	6.540
	deeper trees (optimized)	0.084

Table 6: A comparison of the performance of our implementation of a random forest model with that of the random forest model in the scikit-learn library on the Jena Weather Dataset. Evaluation using the mean squared error (MSE) metric showed that the sklearn model achieved superior performance, but our model performed well at shallow tree depths. As tree depth increased, the performance of our model began to lag behind due to the complex optimizations in the sklearn model.

Method	Mean	Std	Median
Feature selection	0.580	0.052	0.597
Feature selection + normalization	0.918	0.012	0.922
Feature selection + normalization + PCA	0.844	0.038	0.847
Original data	0.580	0.052	0.597
Normalization	0.905	0.012	0.910
Normalization + PCA	0.843	0.032	0.843

Table 7: Mean, standard deviation and median accuracy of third-party implementation of Naive Bayes (sklearn) under different combinations of methods for feature selection, normalization and PCA. Results are given after k-fold cross validation.

4 Conclusions

We conducted weather prediction experiments using the Jena Weather Dataset and three distinct models: Random Forest, Long-Short Term Memory networks, and a Naive Bayes classifier. The Naive Bayes classifier performed well, achieving a maximum accuracy of 89%, while treating the problem as a classification task. For Random Forests and LSTMs, we treated the problem as a regression task and achieved an MSE of 0.084 and 9.24, respectively. Interestingly, our results suggest that classical machine learning algorithms may be more suitable for this task than neural networks. Our findings are in line with recent research on tabular data, where tree-based models performed better than deep neural networks [12].

References

[1] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, Montreal, Canada, vol. 14, 1995, pp. 1137–1145.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[3] J. Sola and J. Sevilla, "Importance of input data normalization for the application of neural networks to complex industrial problems," *IEEE Transactions on nuclear science*, vol. 44, no. 3, pp. 1464–1468, 1997.

[4] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[5] L. Rokach and O. Maimon, "Decision trees," in Jan. 2005, vol. 6, pp. 165–192. DOI: 10.1007/0-387-25465-X_9.

[6] T. Chai and R. R. Draxler, "Root mean square error (rmse) or mean absolute error (mae)?–arguments against avoiding rmse in the literature," *Geoscientific model development*, vol. 7, no. 3, pp. 1247–1250, 2014.

[7] L. Zhang, Y. Ren, and P. N. Suganthan, "Towards generating random forests via extremely randomized trees," in *2014 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2014, pp. 2645–2652.

[8] J. VanderPlas, *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc.", 2016.

[9] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and regression trees*. Routledge, 2017.

[10] A. E. Hassanien and D. A. Oliva, *Advances in soft computing and machine learning in image processing*. Springer, 2017, vol. 730.

[11] H. NB, "Confusion matrix, accuracy, precision, recall, f1 score," Dec. 2019.

[12] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data?" In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

[13] *Timeseries forecasting for weather prediction. keras documentation.* https://keras.io/examples/timeseries/timeseries_weather_forecasting/, Accessed: 2012-12-12.