1_MLBasics

March 16, 2025

0.1 CS-E4740 - Federated Learning D (Spring 25)

1 Assignment 1: ML Basics

1.0.1 R. Gafur, A. Jung

`<h2>Deadline: 17.03.2025</h2>`

1.1 Learning Goals

Access weather data from the Finnish Meteorological Institute (FMI).

Utilize Python libraries (scikit-learn, Keras) to train basic machine learning (ML) models.

Implement regularization techniques via data augmentation.

1.2 Backround Material

- Chapter 2 of FLBook (PDF)
- optional: Linear model implementation in scikitlearn, Decision tree implementation in scikitlearn, Convolutional Neural Networks (CNN) implementation in Keras

1.3 Loading the Data

Before performing any data preprocessing or model training, we begin by loading the dataset and conducting an initial inspection.

Steps:

1. **Load the dataset**: We read the weather data from a CSV file using `pandas`.
2. **Check data types**: Displaying column data types helps us understand the structure of the dataset.
3. **Inspect the first data point**: We print all feature values of the first record to get an overview of the dataset content.
4. **Check dataset dimensions**: The dataset shape is printed to determine the number of rows and columns.

[2]:
```
# General libraries
import pandas as pd
import numpy as np
```

```python
# Tensorflow
import tensorflow as tf  # for CNN model (tf.keras)

# Scikit-learn
from sklearn.tree import DecisionTreeRegressor  # for Decision Tree Regressor
from sklearn.linear_model import LinearRegression  # for Linear Regressoion
from sklearn.metrics import mean_squared_error  # to measure mean squared error␣
↪(MSE)
from sklearn.preprocessing import StandardScaler  # to standardize the features

# Load the data
dataset = pd.read_csv('weather_data.csv')

# Check the data types and
print(dataset.dtypes)
print("\n******************************\n")
print("First data point:")
print(dataset.iloc[0])
print("\n******************************\n")
print(f"Dataset Shape: {dataset.shape}")
```

```
Observation station              object
Year                              int64
Month                             int64
Day                               int64
Time [Local time]                object
Average temperature [°C]        float64
Maximum temperature [°C]        float64
Minimum temperature [°C]        float64
Average relative humidity [%]     int64
Wind speed [m/s]                 object
Maximum wind speed [m/s]         object
Average wind direction [°]       object
Maximum gust speed [m/s]         object
Precipitation [mm]              float64
Average air pressure [hPa]      float64
dtype: object

******************************

First data point:
Observation station             Kustavi Isokari
Year                                       2023
Month                                         4
Day                                           1
Time [Local time]                         00:00
Average temperature [°C]                   -0.6
Maximum temperature [°C]                   -0.4
```

```
Minimum temperature [°C]                     -0.8
Average relative humidity [%]                  82
Wind speed [m/s]                                6
Maximum wind speed [m/s]                      6.4
Average wind direction [°]                     15
Maximum gust speed [m/s]                      7.6
Precipitation [mm]                            0.0
Average air pressure [hPa]                 1007.5
Name: 0, dtype: object

******************************

Dataset Shape: (720, 15)
```

1.3.1 Turning raw data into datapoints, characterized by features and label.

The dataset consists of hourly weather measurements, including:
- **Temperature**: Average, Maximum, Minimum ([°C])
- **Humidity**: Average relative humidity ([%])
- **Wind**: Speed, Maximum speed, Average direction ([°]), Maximum gust speed ([m/s])
- **Precipitation**: Total precipitation ([mm])
- **Air Pressure**: Average air pressure ([hPa])

1.3.2 Data Points

A data point corresponds to a specific hour, e.g., *06-April-2023 from 06:00 - 07:00*, which is recorded as `2023-04-06 06:00:00` (starting time) after preprocessing.

- **Features** include all hourly observations from the **previous 5 hours**. Example: For the data point `2023-04-06 06:00:00`, the features correspond to data from *01:00 - 06:00* on the same day.
- **Label** is the **temperature recorded 5 hours ahead**. Example: For the data point `2023-04-06 06:00:00`, the label corresponds to the temperature measured during *11:00 - 12:00*.
- **Dataset Splits**:
  - **Training Set**: Includes data from `2023-04-06 00:00:00` to `2023-04-08 00:00:00`.
  - **Validation Set**: Comprises the remaining hours of 2023.

[3]:
```python
# Copy the main dataset
data = dataset.copy()

# Convert specified columns to numeric (handling errors with 'coerce')
numeric_columns = [
    'Wind speed [m/s]',
    'Maximum wind speed [m/s]',
    'Average wind direction [°]',
```

```
    'Maximum gust speed [m/s]'
]
data[numeric_columns] = data[numeric_columns].apply(pd.to_numeric,␣
 ↪errors='coerce')

# Fill missing values with 0
data.fillna(0, inplace=True)

# Create a 'timestamp' column by combining year, month, day, and local time
data['timestamp'] = pd.to_datetime(
    data['Year'].astype(str) + '-' +
    data['Month'].astype(str) + '-' +
    data['Day'].astype(str) + ' ' +
    data['Time [Local time]']
)

# Identify columns for lagged features (excluding non-relevant ones)
excluded_columns = ['Observation station', 'timestamp', 'Year', 'Month', 'Day',␣
 ↪'Time [Local time]']
columns_to_lag = [col for col in data.columns if col not in excluded_columns]

# Create lagged features for the previous 5 hours
for lag in range(1, 6):
    for col in columns_to_lag:
        data[f"{col} lag_{lag}"] = data[col].shift(lag)

# Define target variable (y) as the average temperature 5 hours ahead
data['y'] = data['Average temperature [°C]'].shift(-5)

# Drop rows with NaN values caused by shifts
data.dropna(inplace=True)

# Print dataset shape
print(f"Data shape: {data.shape}")
```

```
Data shape: (710, 67)
```

[4]:

```
# Sanity checks

# Check the data rows number
assert data.shape[0] == 710, f"Unexpected number of rows: {data.shape[0]}."

print('Sanity check passed!')
```

```
Sanity check passed!
```

1.3.3 TASK 1.1: Split the Data into Training and Validation Sets

Task Description:

1. **Split the dataset** into training (`X_train`, `y_train`) and validation (`X_val`, `y_val`) sets based on the specified time range.
2. **Training Set**: Includes data from hours `2023-04-06 00:00:00` to (and including) `2023-04-08 00:00:00`.
3. **Validation Set**: Consists of the remaining data.

Hints:

- Use **lagged feature columns** (i.e., columns containing `"lag"`) to create feature sets.
- Assign **lagged columns** as the feature variables for `X_train` and `X_val`.
- Use the **y column** as the target variable for `y_train` and `y_val`.

Points: 0.5

[5]:
```
train_start = '2023-04-06 00:00:00'
train_end = '2023-04-08 00:00:00'

train_data = data[(data['timestamp'] >= train_start) & (data['timestamp'] <=␣
↪train_end)]
val_data = data[data['timestamp'] > train_end]

# Select feature columns (lagged features only)
feature_columns = [col for col in data.columns if "lag" in col]

# Split the data into training and validation sets
# Split into features (X) and target (y)
X_train = train_data[feature_columns]
y_train = train_data['y']
X_val = val_data[feature_columns]
y_val = val_data['y']

print(f"X_train: {X_train.shape}, y_train: {y_train.shape}, X_val: {X_val.
↪shape}, y_val: {y_val.shape}")
```

```
X_train: (49, 50), y_train: (49,), X_val: (546, 50), y_val: (546,)
```

[6]:
```
# Sanity checks

# Check the data rows number
assert X_train.shape[0] == 49, f"Unexpected number of rows for X_train:␣
↪{X_train.shape[0]}."
assert X_val.shape[0] == 546, f"Unexpected number of rows for X_val: {X_val.
↪shape[0]}."

print('Sanity check passed!')
```

```
Sanity check passed!
```

1.3.4 Preparing Data for Model Training: Standardization & Reproducibility

After splitting the dataset into training and validation sets, the next step is to **prepare the features** for model training. Many machine learning models, especially those based on gradient-based optimization (e.g., linear regression, neural networks), perform better when features are standardized.

Why Standardization? Feature standardization ensures that all input variables: - Have a **mean of 0** and **standard deviation of 1**, preventing features with different scales from disproportionately influencing the model. - Enable **faster and more stable convergence** during training.

Steps:

1. **Standardize the Features**:
   - The `StandardScaler` from `scikit-learn` is used to transform both the training and validation datasets.
   - `fit_transform()` is applied to `X_train` to compute and apply the transformation.
   - `transform()` is applied to `X_val` to use the same scaling parameters as `X_train`, ensuring consistency.
2. **Ensure Reproducibility**:
   - To achieve consistent results across different runs, we set random seeds for both **NumPy** and **TensorFlow**.

[8]:
```
# Import the required library for feature scaling
from sklearn.preprocessing import StandardScaler

# Initialize the StandardScaler
scaler = StandardScaler()

# Fit the scaler to the training data and transform it
# This ensures that the mean is 0 and the standard deviation is 1 for each␣
↪feature in X_train
X_train_scaled = scaler.fit_transform(X_train)

# Apply the same transformation to the validation set
# We use transform() instead of fit_transform() to ensure the same scaling␣
↪parameters from X_train are applied
X_val_scaled = scaler.transform(X_val)

# Set random seed for NumPy to ensure reproducibility in any random operations
np.random.seed(42)

# Set random seed for TensorFlow to ensure reproducibility in model training␣
↪and initialization
tf.random.set_seed(42)
```

1.3.5 TASK 1.2: Linear Regression

Task Overview: Now that we have **standardized** our features to ensure proper scaling, we can proceed with training our first machine learning model. In this task, you will implement and evaluate a **linear regression model** using the standardized training and validation datasets.

Task Instructions:

1. **Train a Linear Regression Model** using `X_train_scaled` as input features.
2. **Evaluate Model Performance** by computing the average squared error of the trained model on both the training (`X_train_scaled`,`y_train`) and validation (`X_val_scaled`,`y_val`) sets.

Points: 1.5

[9]:
```
# Initialize and train the Linear Regression model
regressor = LinearRegression()
regressor.fit(X_train_scaled, y_train)

# Predict on the training and validation sets
y_train_pred = regressor.predict(X_train_scaled)
y_val_pred = regressor.predict(X_val_scaled)

# Compute Mean Squared Error (MSE) for training and validation sets
reg_train_error = mean_squared_error(y_train, y_train_pred)
reg_val_error = mean_squared_error(y_val, y_val_pred)

# Display results
print("\n*************** Linear Regression Diagnosis ***************")
print(f"Training Error (MSE): {reg_train_error:.4f}")
print(f"Validation Error (MSE): {reg_val_error:.4f}")
```

```
*************** Linear Regression Diagnosis ***************
Training Error (MSE): 0.0000
Validation Error (MSE): 22.3691
```

[10]:
```
# Sanity Checks: Ensuring Correctness of Model Evaluation

# Check if computed errors are numeric values
assert isinstance(reg_train_error, float), "Error: reg_train_error must be a␣
↪numeric value."
assert isinstance(reg_val_error, float), "Error: reg_val_error must be a␣
↪numeric value."

print("Sanity check passed! The computed errors are valid numerical values.")
```

```
Sanity check passed! The computed errors are valid numerical values.
```

1.3.6 TASK 1.3: Convolutional Neural Network (CNN)

Task Overview: Now that we have trained a **linear regression model**, let's explore a more powerful approach: **a 1D Convolutional Neural Network (CNN)**. CNNs are well-suited for sequential data as they can capture local patterns and temporal dependencies.

Task Instructions:

1. **Implement a 1D CNN Model** using `tf.keras.Sequential` with `X_train_scaled` as input.
2. **Define the CNN architecture** with the following layers:
   - **Input Layer**: Shape `(X_train_scaled.shape[1], 1)`.
   - **Multiple Conv1D Layers** with decreasing filter sizes and `ReLU` activation.
   - **Flatten Layer** to convert feature maps into a vector.
   - **Dense Output Layer** with a single neuron for regression.
3. **Compile the Model** with:
   - **Optimizer**: Adam
   - **Loss Function**: Mean Squared Error (MSE)
4. **Train the Model** on the training set.
5. **Evaluate Model Performance** on both the training and validation sets (similar to Task 1.2).

Hints:

- Use `tf.keras.layers.Conv1D()` to define the convolutional layers with appropriate `filters`, `kernel_size`, and `activation`.

- Use `tf.keras.layers.Flatten()` to reshape the convolutional output before passing it to the dense layer.

- Use `tf.keras.layers.Dense()` for the final output layer.

- Construct the model using `tf.keras.Sequential()`, and compile it with `model.compile()`.

This CNN will serve as a more expressive alternative to linear regression, allowing us to compare their performances.

Points: 2

[11]:
```
# Reshape input data to match Conv1D expected shape (samples, timesteps,␣
↪features)
X_train_cnn = X_train_scaled.reshape(X_train_scaled.shape[0], X_train_scaled.
↪shape[1], 1)
X_val_cnn = X_val_scaled.reshape(X_val_scaled.shape[0], X_val_scaled.shape[1],␣
↪1)

# Define the CNN model
cnn_model = tf.keras.Sequential([
    tf.keras.layers.Conv1D(filters=64, kernel_size=3, activation='relu',␣
↪input_shape=(X_train_cnn.shape[1], 1)),
```

```
    tf.keras.layers.Conv1D(filters=32, kernel_size=3, activation='relu'),
    tf.keras.layers.Conv1D(filters=16, kernel_size=3, activation='relu'),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(1)  # Output layer for regression
])

# Compile the model
cnn_model.compile(optimizer='adam', loss='mse')

# Train the model
cnn_history = cnn_model.fit(X_train_cnn, y_train, epochs=50, batch_size=16,␣
 ↪validation_data=(X_val_cnn, y_val), verbose=1)

# Evaluate the model
cnn_train_error = cnn_model.evaluate(X_train_cnn, y_train, verbose=0)
cnn_val_error = cnn_model.evaluate(X_val_cnn, y_val, verbose=0)

print("\n*************** Convolutional Neural Network (CNN) Diagnosis␣
 ↪***************")
print("Training error:", cnn_train_error)
print("Validation error:", cnn_val_error)
```

```
Epoch 1/50
```

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
4/4               1s 28ms/step - loss:
1.9546 - val_loss: 8.1712
Epoch 2/50
4/4               0s 11ms/step - loss:
0.7802 - val_loss: 20.2136
Epoch 3/50
4/4               0s 12ms/step - loss:
0.2707 - val_loss: 34.0108
Epoch 4/50
4/4               0s 11ms/step - loss:
0.3195 - val_loss: 12.2149
Epoch 5/50
4/4               0s 19ms/step - loss:
0.1175 - val_loss: 7.7711
Epoch 6/50
4/4               0s 11ms/step - loss:
```

```
0.1627 - val_loss: 8.3098
Epoch 7/50
4/4            0s 12ms/step - loss:
0.1093 - val_loss: 11.4444
Epoch 8/50
4/4            0s 12ms/step - loss:
0.1036 - val_loss: 12.9782
Epoch 9/50
4/4            0s 12ms/step - loss:
0.1136 - val_loss: 10.0023
Epoch 10/50
4/4            0s 11ms/step - loss:
0.0828 - val_loss: 8.0377
Epoch 11/50
4/4            0s 12ms/step - loss:
0.0812 - val_loss: 7.9834
Epoch 12/50
4/4            0s 12ms/step - loss:
0.0721 - val_loss: 8.8459
Epoch 13/50
4/4            0s 12ms/step - loss:
0.0717 - val_loss: 8.8929
Epoch 14/50
4/4            0s 12ms/step - loss:
0.0693 - val_loss: 7.9264
Epoch 15/50
4/4            0s 12ms/step - loss:
0.0598 - val_loss: 7.4091
Epoch 16/50
4/4            0s 12ms/step - loss:
0.0554 - val_loss: 7.5467
Epoch 17/50
4/4            0s 12ms/step - loss:
0.0517 - val_loss: 7.7742
Epoch 18/50
4/4            0s 12ms/step - loss:
0.0504 - val_loss: 7.5276
Epoch 19/50
4/4            0s 12ms/step - loss:
0.0467 - val_loss: 7.1900
Epoch 20/50
4/4            0s 12ms/step - loss:
0.0433 - val_loss: 7.1337
Epoch 21/50
4/4            0s 12ms/step - loss:
0.0406 - val_loss: 7.1680
Epoch 22/50
4/4            0s 12ms/step - loss:
```

```
0.0386 - val_loss: 7.0438
Epoch 23/50
4/4             0s 11ms/step - loss:
0.0359 - val_loss: 6.9016
Epoch 24/50
4/4             0s 12ms/step - loss:
0.0333 - val_loss: 6.8995
Epoch 25/50
4/4             0s 12ms/step - loss:
0.0313 - val_loss: 6.9450
Epoch 26/50
4/4             0s 12ms/step - loss:
0.0296 - val_loss: 6.9394
Epoch 27/50
4/4             0s 12ms/step - loss:
0.0276 - val_loss: 6.9361
Epoch 28/50
4/4             0s 12ms/step - loss:
0.0258 - val_loss: 6.9753
Epoch 29/50
4/4             0s 12ms/step - loss:
0.0241 - val_loss: 7.0047
Epoch 30/50
4/4             0s 12ms/step - loss:
0.0225 - val_loss: 6.9994
Epoch 31/50
4/4             0s 12ms/step - loss:
0.0210 - val_loss: 7.0136
Epoch 32/50
4/4             0s 12ms/step - loss:
0.0196 - val_loss: 7.0507
Epoch 33/50
4/4             0s 21ms/step - loss:
0.0183 - val_loss: 7.0651
Epoch 34/50
4/4             0s 12ms/step - loss:
0.0171 - val_loss: 7.0636
Epoch 35/50
4/4             0s 12ms/step - loss:
0.0159 - val_loss: 7.0860
Epoch 36/50
4/4             0s 12ms/step - loss:
0.0149 - val_loss: 7.1105
Epoch 37/50
4/4             0s 12ms/step - loss:
0.0139 - val_loss: 7.1185
Epoch 38/50
4/4             0s 12ms/step - loss:
```

```
0.0130 - val_loss: 7.1361
Epoch 39/50
4/4              0s 12ms/step - loss:
0.0122 - val_loss: 7.1451
Epoch 40/50
4/4              0s 12ms/step - loss:
0.0114 - val_loss: 7.1447
Epoch 41/50
4/4              0s 12ms/step - loss:
0.0106 - val_loss: 7.1511
Epoch 42/50
4/4              0s 12ms/step - loss:
0.0099 - val_loss: 7.1428
Epoch 43/50
4/4              0s 12ms/step - loss:
0.0093 - val_loss: 7.1265
Epoch 44/50
4/4              0s 12ms/step - loss:
0.0087 - val_loss: 7.1272
Epoch 45/50
4/4              0s 12ms/step - loss:
0.0081 - val_loss: 7.1377
Epoch 46/50
4/4              0s 12ms/step - loss:
0.0076 - val_loss: 7.1437
Epoch 47/50
4/4              0s 15ms/step - loss:
0.0071 - val_loss: 7.1623
Epoch 48/50
4/4              0s 12ms/step - loss:
0.0066 - val_loss: 7.1919
Epoch 49/50
4/4              0s 12ms/step - loss:
0.0062 - val_loss: 7.2208
Epoch 50/50
4/4              0s 12ms/step - loss:
0.0058 - val_loss: 7.2421

*************** Convolutional Neural Network (CNN) Diagnosis ***************
Training error: 0.0049854451790452
Validation error: 7.242134094238281
```

[12]:

```
# Sanity checks

# Check if the variables store numeric values
assert isinstance(cnn_train_error, float), "cnn_train_error must be a numeric␣
↪value."
```

```
assert isinstance(cnn_val_error, float), "cnn_val_error must be a numeric value.
↪"

print('Sanity check passed!')
```

```
Sanity check passed!
```

1.3.7 TASK 1.4: Decision Tree Regressor

Task Overview: Building on our previous models, we will now implement a **Decision Tree Regressor**, a non-parametric model that can capture nonlinear relationships in the data.

Task Instructions:

1. **Train a Decision Tree Regressor** using `X_train_scaled` as input features with `max_depth=3`.
2. **Compute Mean Squared Error (MSE)** for training (`X_train_scaled`,`y_train`) and validation (`X_val_scaled`,`y_val`) sets.

Points: 1.5

[13]:
```
# Initialize and train the Decision Tree Regressor with max_depth=3
tree_regressor = DecisionTreeRegressor(max_depth=3)
tree_regressor.fit(X_train_scaled, y_train)

# Predict on the training and validation sets
y_train_pred_tree = tree_regressor.predict(X_train_scaled)
y_val_pred_tree = tree_regressor.predict(X_val_scaled)

# Compute Mean Squared Error (MSE) for training and validation sets
tree_train_error = mean_squared_error(y_train, y_train_pred_tree)
tree_val_error = mean_squared_error(y_val, y_val_pred_tree)

print("\n*************** Decision Tree Regressor Diagnosis ***************")
print("Training error:", tree_train_error)
print("Validation error:", tree_val_error)
```

```
*************** Decision Tree Regressor Diagnosis ***************
Training error: 0.023813573048266926
Validation error: 12.172040647031434
```

[14]:
```
# Sanity checks

# Check if the variables store numeric values
assert isinstance(tree_train_error, float), "tree_train_error must be a numeric␣
↪value."
assert isinstance(tree_val_error, float), "tree_val_error must be a numeric␣
↪value."
```

```
print('Sanity check passed!')
```

```
Sanity check passed!
```

1.4 Regularization

So far, we have trained **three basic ML models**, a linear model, a CNN and a decision tree. While these models can effectively learn patterns from the training data, they are **prone to overfitting**, especially when the dataset is small. To improve generalization, we can use **regularization techniques** such as adding the **ridge penalty**.

1.4.1 Implementing the Ridge Penalty with Data Augmentation

To implement **ridge regularization**, we will: 1. **Generate an augmented training set** using independent and identically distributed (i.i.d.) **Gaussian random vectors**. 2. **Set the labels** for this augmented data to **0**. 3. **Modify our existing models** (Linear Regression, CNN, and Decision Tree) to incorporate this augmented dataset.

This approach introduces **a penalty term**, encouraging models to keep their predictions closer to zero for these artificial data points, thus improving stability and reducing overfitting.

[15]:
```
import numpy as np
import pandas as pd
import tensorflow as tf

def generate_augmented_dataset(X_train, y_train, augmentation_ratio=0.5,␣
↪random_seed=42):
    """
    Generates an augmented training set by adding synthetic data points drawn␣
↪from a Gaussian distribution.

    Parameters:
    - X_train (numpy array or DataFrame): Original training feature set.
    - y_train (numpy array or Series): Original training labels.
    - augmentation_ratio (float): Ratio of synthetic samples to real training␣
↪samples (default: 0.5).
    - random_seed (int): Random seed for reproducibility.

    Returns:
    - X_train_aug (numpy array): Augmented feature set.
    - y_train_aug (numpy array): Augmented labels (including original and␣
↪synthetic samples).
    """

    # Set random seed for full reproducibility
    np.random.seed(random_seed)
    tf.random.set_seed(random_seed)  # Ensure reproducibility in TensorFlow if␣
↪used later
```

```python
    # Determine the number of augmented samples
    num_augmented_samples = int(augmentation_ratio * X_train.shape[0])

    # Compute mean and standard deviation of each feature in X_train
    feature_means = np.mean(X_train, axis=0)
    feature_stds = np.std(X_train, axis=0) + 1e-8  # Small epsilon to avoid␣
↪zero variance issues

    # Generate synthetic feature vectors from a Gaussian distribution
    X_augmented = np.random.normal(loc=feature_means, scale=feature_stds,␣
↪size=(num_augmented_samples, X_train.shape[1]))

    # Set the labels for the augmented data points to 0
    y_augmented = np.zeros(num_augmented_samples)

    # Combine the original and augmented datasets
    X_train_aug = np.vstack((X_train, X_augmented))
    y_train_aug = np.hstack((y_train, y_augmented))

    return X_train_aug, y_train_aug

# Example usage:
# Assuming X_train_scaled and y_train are already defined
X_train_aug, y_train_aug = generate_augmented_dataset(X_train_scaled, y_train)

print("Original training set shape:", X_train_scaled.shape)
print("Augmented training set shape:", X_train_aug.shape)
```

```
Original training set shape: (49, 50)
Augmented training set shape: (73, 50)
```

1.4.2 TASK 1.5: Regularized Model Training with Augmented Data

In this task, you will **regularize the training** of all three models from previous tasks—**Linear Regression, CNN, and Decision Tree Regressor**—by replacing the original training set with the **augmented training set** generated above. The augmented dataset contains **synthetic feature vectors drawn from a Gaussian distribution**, with their labels set to **0**. For parametric models, using this augmented dataset to train ML a model is equivalent to adding a ridge penalty term (see Sec. 6.6. of MLBook (PDF)).

Task Instructions:

1. **Train the following models using the augmented training set (`X_train_aug`, `y_train_aug`):**
   - **Linear Regression**
   - **Convolutional Neural Network (CNN)**
   - **Decision Tree Regressor**

2. **Compute Mean Squared Error (MSE)** of these regularized models on the original training (`X_train_scaled,y_train`) and validation sets (`X_val_scaled,y_val`).

Points: 1.5

[18]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, Flatten, Dense

# Train a Linear Regression model using the augmented dataset
custom_reg = LinearRegression()
custom_reg.fit(X_train_aug, y_train_aug)

# Train a CNN using the augmented dataset
# Reshape X_train_aug for Conv1D
X_train_cnn_aug = X_train_aug.reshape(X_train_aug.shape[0], X_train_aug.
 ↪shape[1], 1)

# Define the CNN model
custom_cnn = Sequential([
    Conv1D(filters=64, kernel_size=3, activation='relu',␣
 ↪input_shape=(X_train_cnn_aug.shape[1], 1)),
    Conv1D(filters=32, kernel_size=3, activation='relu'),
    Conv1D(filters=16, kernel_size=3, activation='relu'),
    Flatten(),
    Dense(16, activation='relu'),
    Dense(1)  # Output layer for regression
])

# Compile and train the CNN model
custom_cnn.compile(optimizer='adam', loss='mse')
custom_cnn.fit(X_train_cnn_aug, y_train_aug, epochs=50, batch_size=16,␣
 ↪verbose=1)

# Train a Decision Tree Regressor using the augmented dataset
custom_tree = DecisionTreeRegressor(max_depth=3)
custom_tree.fit(X_train_aug, y_train_aug)


custom_reg_train_error = mean_squared_error(y_train, custom_reg.
 ↪predict(X_train_scaled))
custom_reg_val_error = mean_squared_error(y_val, custom_reg.
 ↪predict(X_val_scaled))

print("\n*************** Diagnosis of Regularized Linear Model ***************")
print("Training error:", custom_reg_train_error)
print("Validation error:", custom_reg_val_error)
```

```
custom_cnn_train_error = custom_cnn.evaluate(X_train_scaled[..., np.newaxis],␣
↪y_train, verbose=0)
custom_cnn_val_error = custom_cnn.evaluate(X_val_scaled[..., np.newaxis],␣
↪y_val, verbose=0)

print("\n*************** Diagnosis of Regularized CNN ***************")
print("Training error:", custom_cnn_train_error)
print("Validation error:", custom_cnn_val_error)

custom_tree_train_error = mean_squared_error(y_train, custom_tree.
↪predict(X_train_scaled))
custom_tree_val_error = mean_squared_error(y_val, custom_tree.
↪predict(X_val_scaled))

print("\n*************** Diagnosis of Regularized Decision Tree␣
↪***************")
print("Training error:", custom_tree_train_error)
print("Validation error:", custom_tree_val_error)
```

```
Epoch 1/50
```

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
5/5               1s 3ms/step - loss:
1.6438
Epoch 2/50
5/5               0s 3ms/step - loss:
0.7619
Epoch 3/50
5/5               0s 3ms/step - loss:
0.5498
Epoch 4/50
5/5               0s 3ms/step - loss:
0.4007
Epoch 5/50
5/5               0s 3ms/step - loss:
0.3484
Epoch 6/50
5/5               0s 3ms/step - loss:
0.2927
Epoch 7/50
5/5               0s 2ms/step - loss:
0.2172
```

```
Epoch 8/50
5/5            0s 2ms/step - loss:
0.1588
Epoch 9/50
5/5            0s 10ms/step - loss:
0.1161
Epoch 10/50
5/5            0s 3ms/step - loss:
0.0822
Epoch 11/50
5/5            0s 3ms/step - loss:
0.0641
Epoch 12/50
5/5            0s 3ms/step - loss:
0.0503
Epoch 13/50
5/5            0s 3ms/step - loss:
0.0389
Epoch 14/50
5/5            0s 3ms/step - loss:
0.0309
Epoch 15/50
5/5            0s 2ms/step - loss:
0.0249
Epoch 16/50
5/5            0s 2ms/step - loss:
0.0202
Epoch 17/50
5/5            0s 3ms/step - loss:
0.0164
Epoch 18/50
5/5            0s 3ms/step - loss:
0.0138
Epoch 19/50
5/5            0s 3ms/step - loss:
0.0120
Epoch 20/50
5/5            0s 3ms/step - loss:
0.0102
Epoch 21/50
5/5            0s 3ms/step - loss:
0.0090
Epoch 22/50
5/5            0s 3ms/step - loss:
0.0080
Epoch 23/50
5/5            0s 3ms/step - loss:
0.0071
```

```
Epoch 24/50
5/5             0s 3ms/step - loss:
0.0063
Epoch 25/50
5/5             0s 3ms/step - loss:
0.0056
Epoch 26/50
5/5             0s 3ms/step - loss:
0.0051
Epoch 27/50
5/5             0s 3ms/step - loss:
0.0045
Epoch 28/50
5/5             0s 3ms/step - loss:
0.0041
Epoch 29/50
5/5             0s 3ms/step - loss:
0.0037
Epoch 30/50
5/5             0s 3ms/step - loss:
0.0033
Epoch 31/50
5/5             0s 3ms/step - loss:
0.0029
Epoch 32/50
5/5             0s 3ms/step - loss:
0.0026
Epoch 33/50
5/5             0s 3ms/step - loss:
0.0023
Epoch 34/50
5/5             0s 3ms/step - loss:
0.0020
Epoch 35/50
5/5             0s 3ms/step - loss:
0.0018
Epoch 36/50
5/5             0s 3ms/step - loss:
0.0016
Epoch 37/50
5/5             0s 10ms/step - loss:
0.0014
Epoch 38/50
5/5             0s 3ms/step - loss:
0.0012
Epoch 39/50
5/5             0s 3ms/step - loss:
0.0011
```

```
Epoch 40/50
5/5            0s 3ms/step - loss:
9.8457e-04
Epoch 41/50
5/5            0s 3ms/step - loss:
8.7058e-04
Epoch 42/50
5/5            0s 3ms/step - loss:
7.7667e-04
Epoch 43/50
5/5            0s 3ms/step - loss:
6.9546e-04
Epoch 44/50
5/5            0s 3ms/step - loss:
6.1022e-04
Epoch 45/50
5/5            0s 3ms/step - loss:
5.4899e-04
Epoch 46/50
5/5            0s 3ms/step - loss:
4.8869e-04
Epoch 47/50
5/5            0s 3ms/step - loss:
4.3266e-04
Epoch 48/50
5/5            0s 3ms/step - loss:
3.8289e-04
Epoch 49/50
5/5            0s 3ms/step - loss:
3.4270e-04
Epoch 50/50
5/5            0s 3ms/step - loss:
3.0429e-04

*************** Diagnosis of Regularized Linear Model ***************
Training error: 0.10068223140329917
Validation error: 17.034471617313823

*************** Diagnosis of Regularized CNN ***************
Training error: 0.00034364761086180806
Validation error: 7.044750690460205

*************** Diagnosis of Regularized Decision Tree ***************
Training error: 0.1841595571737582
Validation error: 12.795367895428631
```

[19]:
```
# Sanity checks

# Check if the variables store numeric values
assert isinstance(reg_train_error, float), "custom_reg_train_error must be a␣
↪numeric value."
assert isinstance(reg_val_error, float), "custom_reg_val_error must be a␣
↪numeric value."

# Check if the variables store numeric values
assert isinstance(cnn_train_error, float), "cnn_train_error must be a numeric␣
↪value."
assert isinstance(cnn_val_error, float), "cnn_val_error must be a numeric value.
↪"

# Check if the variables store numeric values
assert isinstance(tree_train_error, float), "tree_train_error must be a numeric␣
↪value."
assert isinstance(tree_val_error, float), "tree_val_error must be a numeric␣
↪value."

print('Sanity check passed!')
```

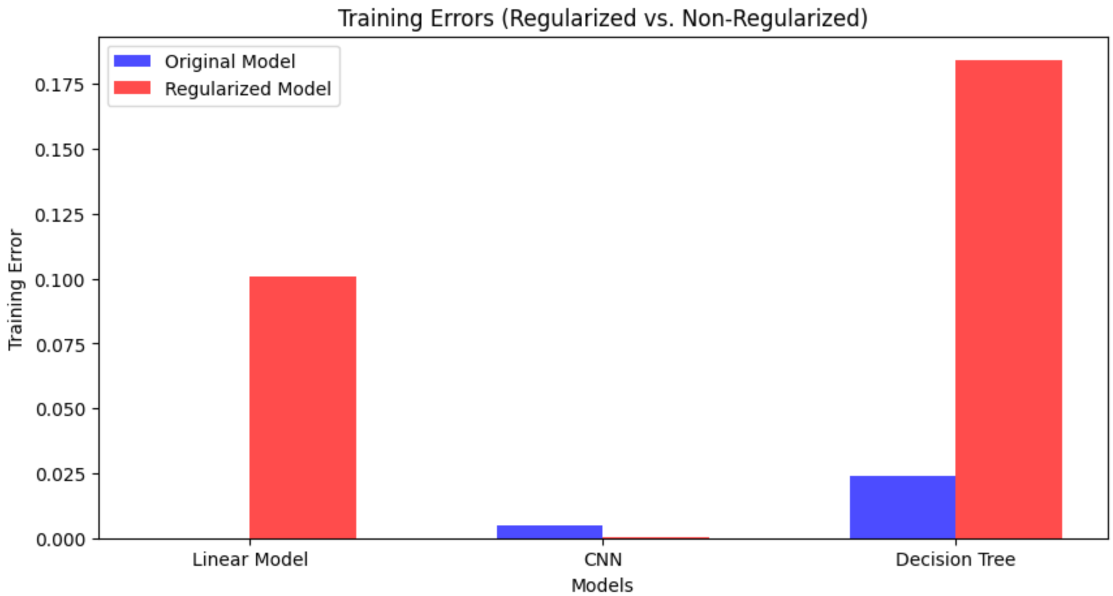
```
Sanity check passed!
```

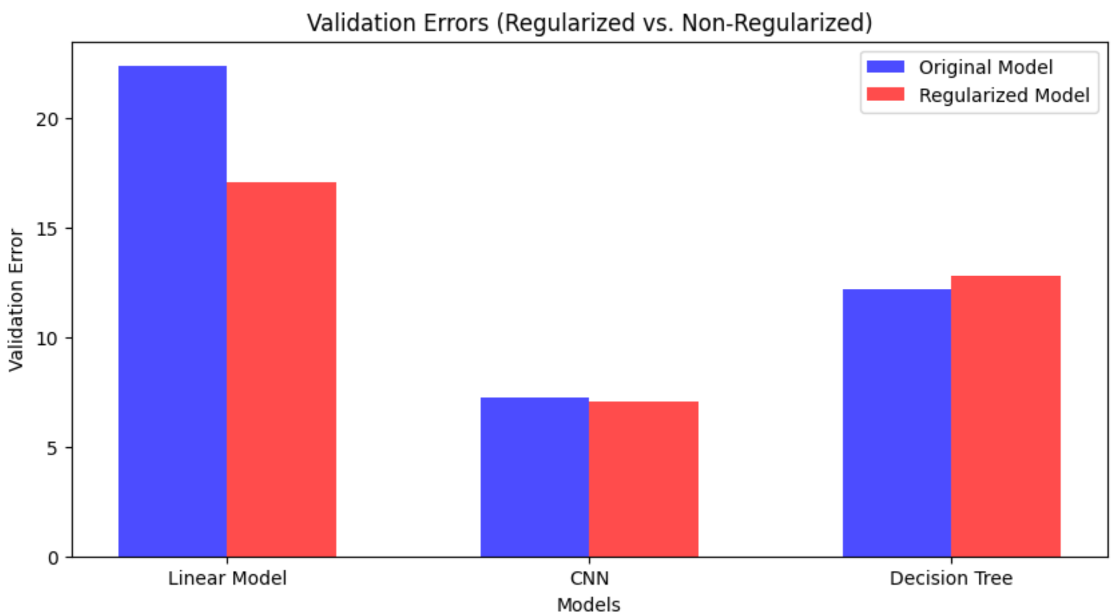
1.5 Results

[20]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Define model names
models = ["Linear Model", "CNN", "Decision Tree"]

# Training errors
train_errors = [reg_train_error, cnn_train_error, tree_train_error]
custom_train_errors = [custom_reg_train_error, custom_cnn_train_error,␣
↪custom_tree_train_error]

# Validation errors
val_errors = [reg_val_error, cnn_val_error, tree_val_error]
custom_val_errors = [custom_reg_val_error, custom_cnn_val_error,␣
↪custom_tree_val_error]

# Create a DataFrame for better visualization
error_data = pd.DataFrame({
    "Model": models,
    "Training Error (Original)": train_errors,
```

```
    "Training Error (Regularized)": custom_train_errors,
    "Validation Error (Original)": val_errors,
    "Validation Error (Regularized)": custom_val_errors
})

# Print the table
print(error_data.to_string(index=False))

# Define x-axis positions for grouped bar charts
x = np.arange(len(models))  # Model positions
width = 0.3  # Bar width for better visualization

# Plot Training Errors
plt.figure(figsize=(10, 5))
plt.bar(x - width/2, train_errors, width, label='Original Model', color='blue',␣
 ↪alpha=0.7)
plt.bar(x + width/2, custom_train_errors, width, label='Regularized Model',␣
 ↪color='red', alpha=0.7)
plt.xlabel("Models")
plt.ylabel("Training Error")
plt.title("Training Errors (Regularized vs. Non-Regularized)")
plt.xticks(ticks=x, labels=models)
plt.legend()
plt.show()

# Plot Validation Errors
plt.figure(figsize=(10, 5))
plt.bar(x - width/2, val_errors, width, label='Original Model', color='blue',␣
 ↪alpha=0.7)
plt.bar(x + width/2, custom_val_errors, width, label='Regularized Model',␣
 ↪color='red', alpha=0.7)
plt.xlabel("Models")
plt.ylabel("Validation Error")
plt.title("Validation Errors (Regularized vs. Non-Regularized)")
plt.xticks(ticks=x, labels=models)
plt.legend()
plt.show()
```

```
        Model  Training Error (Original)  Training Error (Regularized)
Validation Error (Original)  Validation Error (Regularized)
 Linear Model               4.561860e-30                      0.100682
22.369070                       17.034472
          CNN               4.985445e-03                      0.000344
7.242134                        7.044751
Decision Tree               2.381357e-02                      0.184160
12.172041                       12.795368
```


Training Errors (Regularized vs. Non-Regularized)
Original Model
Regularized Model
Training Error
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
Linear Model
CNN
Decision Tree
Models


Validation Errors (Regularized vs. Non-Regularized)
Original Model
Regularized Model
Validation Error
20
15
10
5
0
Linear Model
CNN
Decision Tree
Models

[ ]: