Power Prediction Of combined cycle power plant (CCPP) Using Machine Learning Algorithms

Proposed architecture of the prediction of CCPP power using ML algorithm

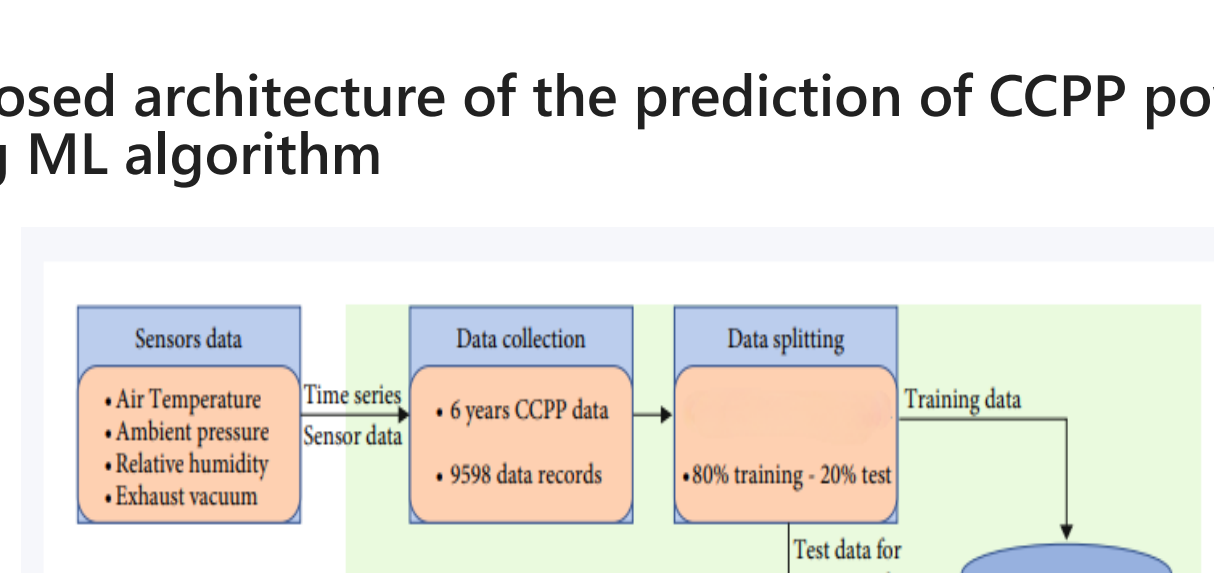


Figure 1: Proposed architecture of the prediction of CCPP power using ML algorithm.

1.Introduction

Power prediction is important not only for the smooth and economic operation of a combined cycle power plant (CCPP) but also to avoid technical issues such as power outages. In this work, we propose to utilize machine learning algorithms to predict the hourly-based electrical power generated by a CCPP. For this, the generated power is considered a function of four fundamental parameters which are relative humidity, atmospheric pressure, ambient temperature, and exhaust vacuum. The measurements of these parameters and their yielded output power are used to train and test the machine learning models. The dataset for the proposed research is gathered over a period of six years and taken from a standard and publicly available machine learning repository.

We predicting the net hourly electrical energy output (PE) of the plant using dataset from UCI Machine Learning Repository.

The accurate prediction of power generated by a plant helps in reducing various related issues such as power outages, economic, and technical difficulties. In particular, an inaccurate prediction results in the rise of per unit cost of electric power due to the high fuel consumption. Hence, in this work, we aim at achieving a precise prediction of electric power of a base load CCPP on full load conditions thus ensuring decreased cost of per unit of electric power.

calculated using complex mathematical models.These models involve a vast range of assumptions and parameters to reflect the actual uncertainty of the system. However, these mathematical models are time consuming and are based on a deterministic approach.

On the other hand, supervised machine learning (ML) algorithms incorporate probabilistic approaches for power prediction instead of mathematical modeling. With the availability of data, the prediction done with the ML approach is far more convenient, scalable, and flexible thus preventing to model the whole the system.

Our proposed ML algorithms assess historical data of a power plant, operating under a variety of environmental conditions, in order to provide optimized power forecasts in less time. However, being probabilistic in nature, the predictions made by ML algorithms involve errors. Due to this reason, we propose to evaluate several algorithms for the task of power prediction of a CCPP. Furthermore, we also search for the parameters of these algorithms where they give the least error on the current dataset.

The generated electric power of a CCPP unexpectedly fluctuates throughout the whole year due to several parameters, such as ambient temperature, atmospheric pressure, humidity, and exhaust vacuum. Consequently, these parameters directly and indirectly influence the output power of a CCPP. Therefore, power production can be improved and fuel consumption can be reduced by optimally controlling these parameters. The primary focus of this work is to analyze the influence of ambient parameters on output power prediction rather than controlling the parameters. For this purpose, these environmental parameters are used to predict the electric power through various machine learning algorithms.

We used Various probabilistic approaches for CCPP power prediction of electric power of CCPP operated on full load using DecisionTreeRegressor (DT), linear regression (LR), artificial neural network (ANN), RandomForestRegressor (RF), LGBMRegressor (LGBR), XGBRegressor (XGBR), CatBoostRegressor (CATBR), and SVM (SVR) are used to improve the power prediction. The individual and cumulative effects of each parameter are evaluated on output power prediction using these machine learning algorithms. These algorithms are compared to find the best results using optomization framework as GridSearchCV. The best performance among these machine learning models is analyzed by choosing the least RMSE and AE values which is our metrics.

2.Data Set Information

The dataset contains 47844 data points collected from a Combined Cycle Power Plant over 6 years (2006-2011) with 5 feature, when the power plant was set to work with full load. Features consist of hourly average ambient variables Ambient Temperature (AT), Ambient Pressure (AP), Relative Humidity (RH) and Exhaust Vacuum (V) to predict the net hourly electrical energy output (PE) of the plant.

A combined cycle power plant (CCPP) is composed of gas turbines (GT), steam turbines (ST) and heat recovery steam generators. In a CCPP, the electricity is generated by gas and steam turbines, which are combined in one cycle, and is transferred from one turbine to another. While the Vacuum is colected from and has effect on the Steam Turbine, the other three of the ambient variables effect the GT performance.

2.1 Attribute Information

Features consist of hourly average ambient variables :

- Ambient Temperature (AT) in the range `1.81°C and 37.11°C`
- Ambient Pressure (AP) in the range `992.89-1033.30 milibar`
- Relative Humidity (RH) in the range `25.56% to 100.16%`
- Exhaust Vacuum (V) in teh range `25.36-81.56 cm Hg`
- Net hourly electrical energy output (PE) `420.26-495.76 MW`

The averages are taken from various sensors located around the plant that record the ambient variables every second. The variables are given without normalization

In [1]:
```python
DATA_PH = "Combined Cycle Power Plant Data Set.csv"
```

In [3]:
```python
# import libraries
import pandas as pd
```

Input and output attributes of CCPP, where the temperature is measured in celcius, vaccum is measure in cm-Hg, pressure is measured in millibars, humidity is in percentage, and the unit of predicted power is megawatts

In [4]:
```python
df = pd.read_csv(DATA_PH)
df.head(10)
```

Out[4]:

	AT	V	AP	RH	PE
0	14.96	41.76	1024.07	73.17	463.26
1	25.18	62.96	1020.04	59.08	444.37
2	5.11	39.4	1012.16	92.14	488.56
3	20.86	57.32	1010.24	76.64	446.48
4	10.82	37.5	1009.23	96.62	473.9
5	26.27	59.44	1012.23	58.77	443.67
6	15.89	43.96	1014.02	75.24	467.35
7	9.48	44.71	1019.12	66.43	478.42
8	14.64	45	1021.78	41.25	475.98
9	11.74	43.56	1015.14	70.72	477.5

- The dataset contains 47844 data points collected from a Combined Cycle Power Plant over 6 years (2006-2011) with 5 feature where PE is our target feature

In [10]:
```python
df.shape
```

Out[10]:
```
(47844, 5)
```

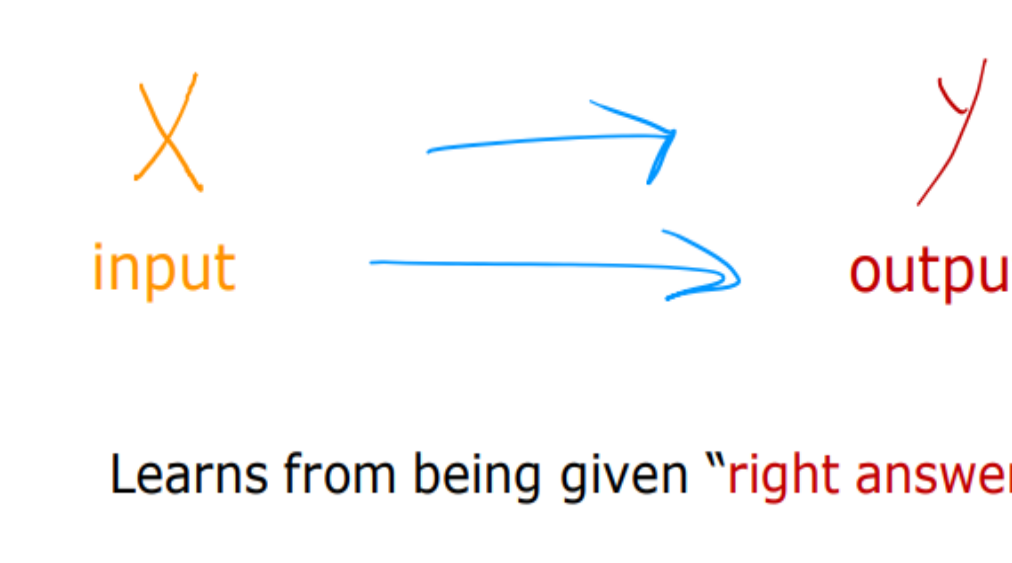
3. Supervised Machine Learning

Supervised machine learning is a powerful approach that enables computers to learn patterns and make predictions or decisions based on labeled training data. In supervised learning, a machine learning model is trained using input-output pairs, where the input is the data and the output is the desired prediction or label. The goal is to learn a function that can generalize from the training data to accurately predict the output for unseen inputs.

Supervised machine learning is split into two category:

- Regression problems (dealing with numeric target feature)
- Classification problems (dealing with categorical target feature)

- Supervised ML where we have our label data



3.1 Regression Supervised ML

Regression is a popular branch of supervised machine learning that focuses on predicting continuous numerical values based on input data. In regression, the goal is to build a model that can learn the underlying relationship between input features and the target variable, enabling accurate predictions on new, unseen data.

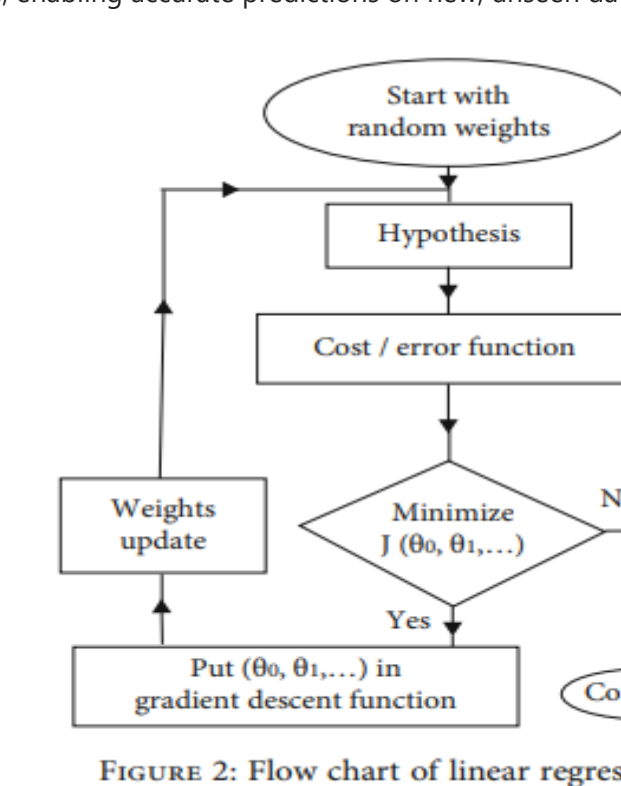


Figure 2: Flow chart of linear regression.

- Matrix X containing our examples

Similar to the table above, examples are stored in a NumPy matrix `X_train`. Each row of the matrix represents one example. When you have m training examples (m is three in our example), and there are n features (5 in our example), $\mathbf{X}$ is a matrix with dimensions (m, n) (m rows, n columns).

$$\mathbf{X} = \begin{pmatrix} x^{(0)}_0 & x^{(0)}_1 & \cdots & x^{(0)}_{n-1} \\ x^{(1)}_0 & x^{(1)}_1 & \cdots & x^{(1)}_{n-1} \\ \cdots \\ x^{(m-1)}_0 & x^{(m-1)}_1 & \cdots & x^{(m-1)}_{n-1} \end{pmatrix}$$

notation:

- $\mathbf{x}^{(i)}$ is vector containing example i. $\mathbf{x}^{(i)} = (x^{(i)}_0, x^{(i)}_1, \cdots,x^{(i)}_{n-1})$
- $x^{(i)}_j$ is element j in example i. The superscript in parenthesis indicates the example number while the subscript represents an element.

Parameter vector w, b

- $\mathbf{w}$ is a vector with n elements.
  - Each element contains the parameter associated with one feature.
  - in our dataset, n is 5.
  - notionally, we draw this as a column vector

$$\mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \cdots \\ w_{n-1} \end{pmatrix}$$

- b is a scalar parameter.

the model is trained using a labeled dataset where the input features are paired with the corresponding target values. The model learns the relationship between the input features and the target variable by optimizing its parameters or coefficients to minimize the prediction errors.

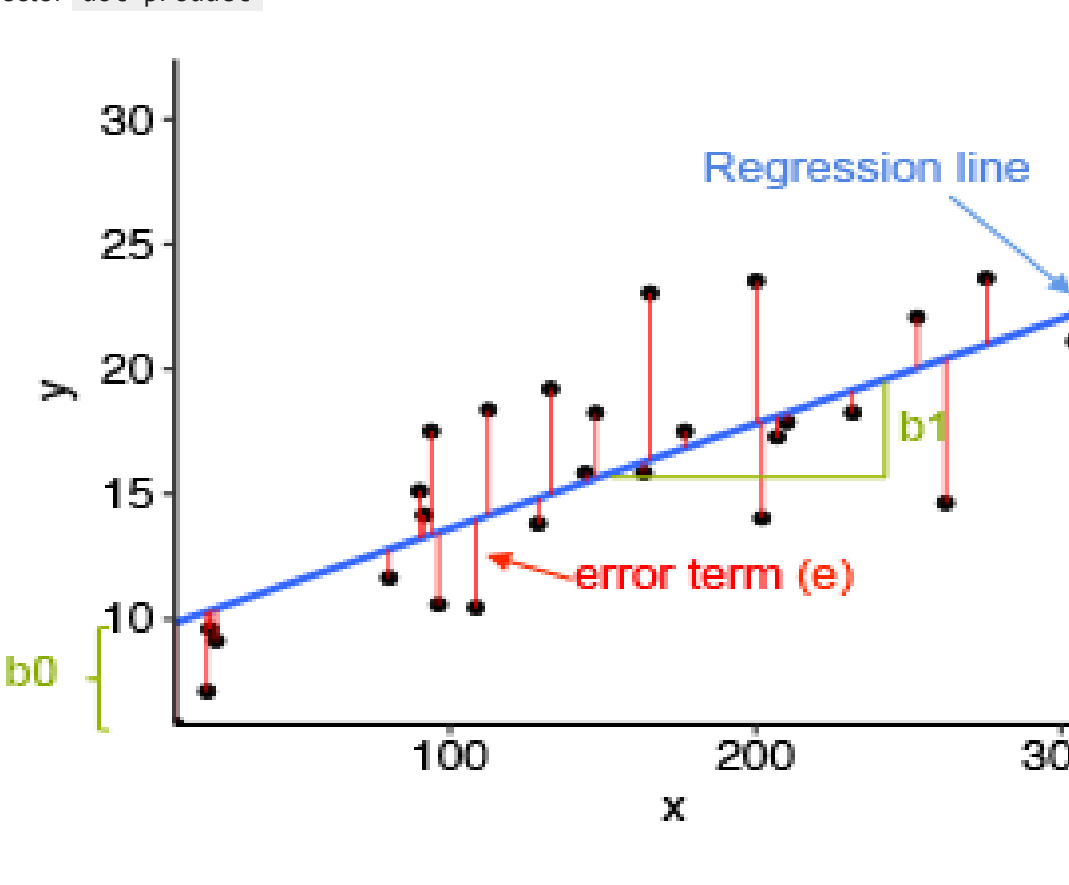
The model's prediction with multiple variables is given by the linear model:

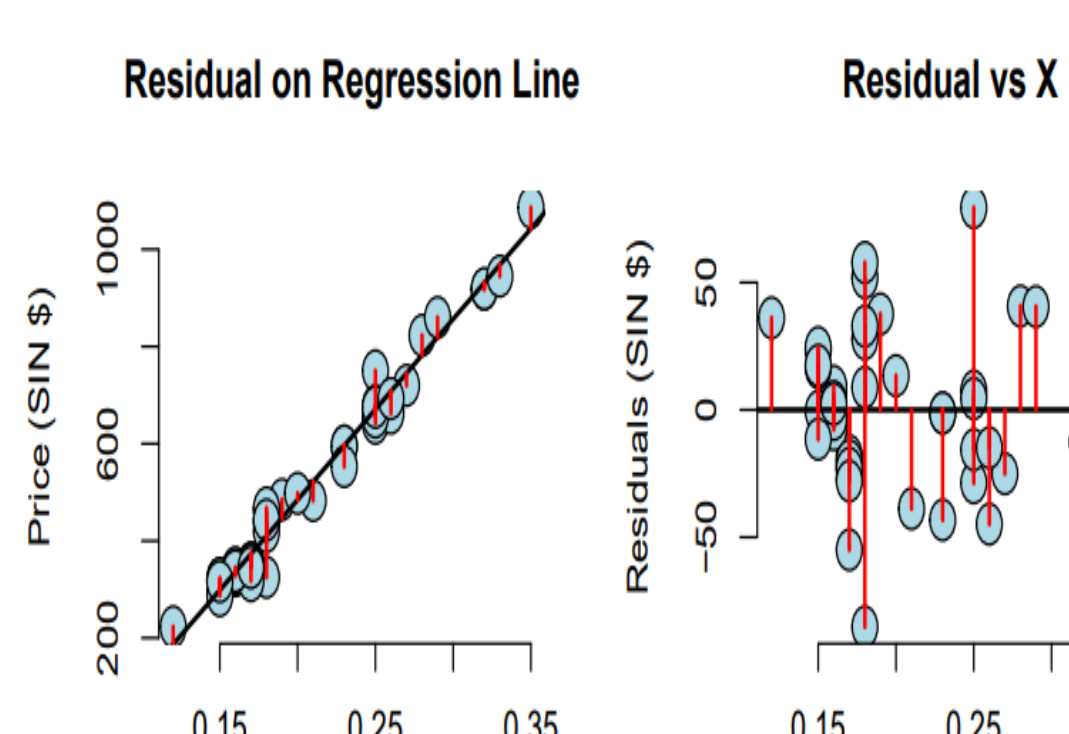
$$f_{\mathbf{w},b}(\mathbf{x}) = w_0x_0 + w_1x_1 +... + w_{n-1}x_{n-1} + b \tag{1}$$

or in vector notation:

$$f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b \tag{2}$$

where $\cdot$ is a vector `dot product`





- Residual represent our error value which is the difference between predicting one $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ and our true value $y^{(i)}$
- we represent this is cost func below $J(\mathbf{w},b)$

The equation for the cost function with multiple variables $J(\mathbf{w},b)$ is:

$$J(\mathbf{w},b) = \frac{1}{2m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{3}$$

where:

$$f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x}^{(i)} + b \tag{4}$$

In contrast to previous labs, $\mathbf{w}$ and $\mathbf{x}^{(i)}$ are vectors rather than scalars supporting multiple features.

- GOAL is to MINIMIZE the cost func $J(\mathbf{w},b)$, we do that by using Gradient Descent Algorithm.

3.1.1 Gradient Descent With Multiple Variables

It's an optimization algorithm commonly used in machine learning to minimize the cost or error function of a model with multiple input variables.

The goal of gradient descent is to iteratively update the model's parameters in a way that reduces the cost function, ultimately finding the optimal parameter values that yield the best predictions.

Gradient descent for multiple variables:

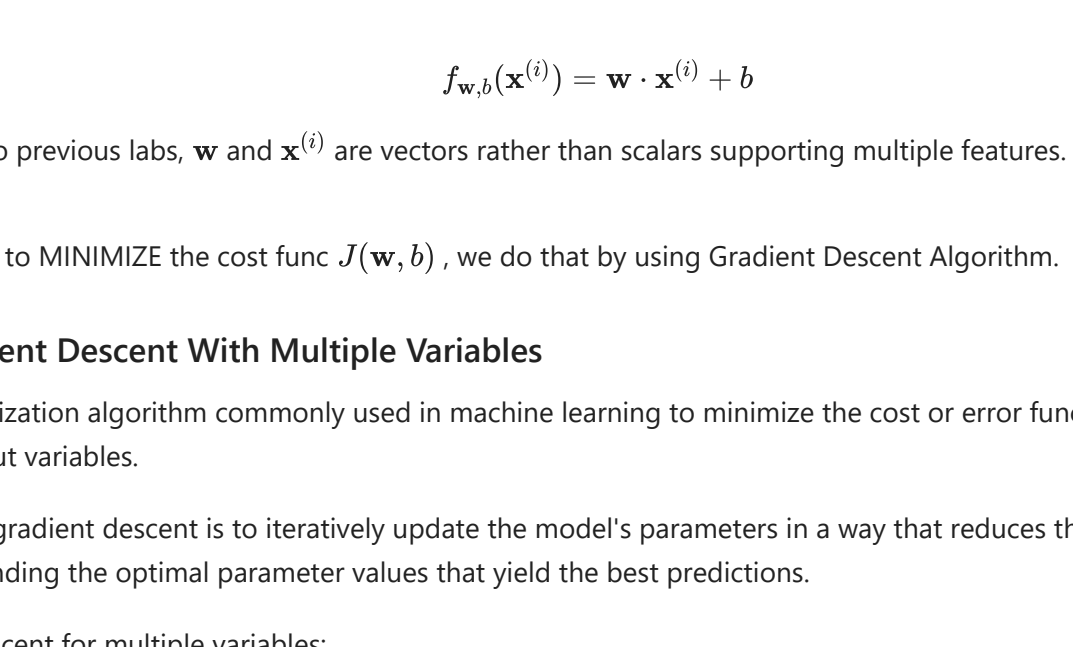
$$\begin{align*} \text{repeat}&\text{ until convergence:} \; \lbrace \newline\; & w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \tag{5} \; & \text{for j = 0..n-1}\newline &b\ \ = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \newline \rbrace \end{align*}$$

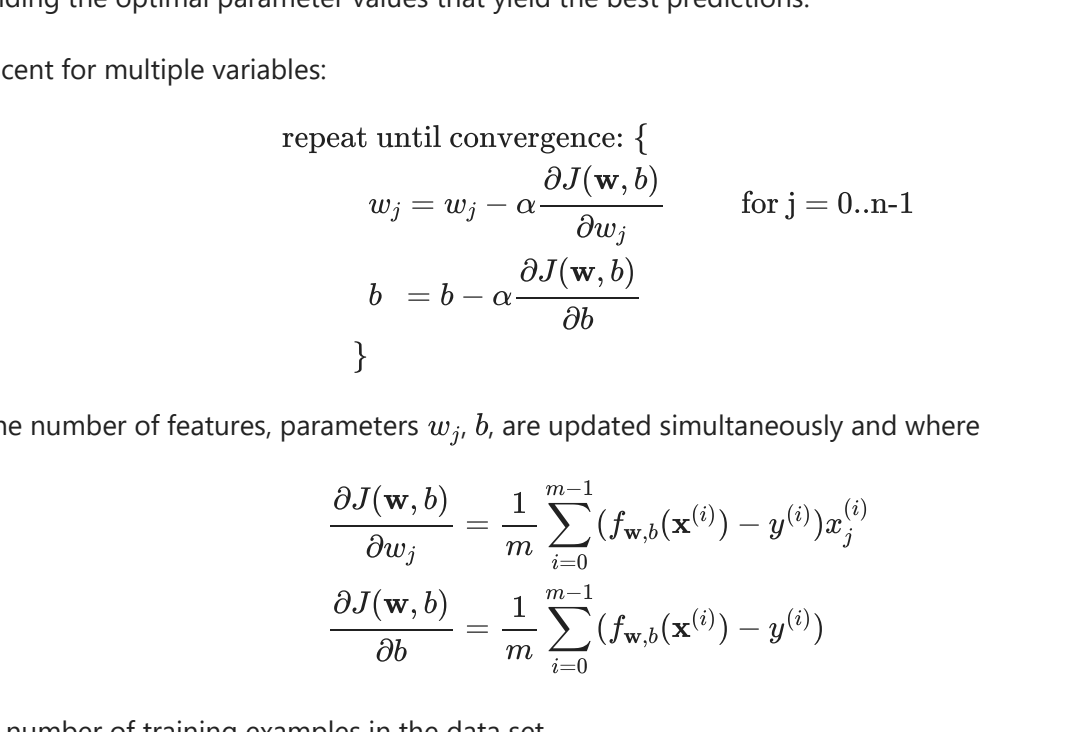
where, n is the number of features, parameters w_j, b, are updated simultaneously and where

$$\begin{align} \frac{\partial J(\mathbf{w},b)}{\partial w_j} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_{j}^{(i)} \tag{6} \\ \frac{\partial J(\mathbf{w},b)}{\partial b} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \tag{7} \end{align}$$

- m is the number of training examples in the data set

- $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ is the model's prediction, while $y^{(i)}$ is the target value





4. Data Analysis

4.1 Analysis

In [5]:
```python
# import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [6]:
```python
# convert data to numeric so we can work on it
def convert_to_numeric(df):
    """
    errors = 'coerce':
        Convert non-numeric values to NaN.
    """
    for col in df.columns:
        df[col] = pd.to_numeric(df[col], errors='coerce')
    return df

df = convert_to_numeric(df)

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 47844 entries, 0 to 47843
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   AT      47840 non-null  float64
 1   V       47840 non-null  float64
 2   AP      47840 non-null  float64
 3   RH      47840 non-null  float64
 4   PE      47840 non-null  float64
dtypes: float64(5)
memory usage: 1.8 MB
```

function below to calculate μ_i (mean for each feature in `X`) and `var` = σ_i^2 (variance for each feature in `X`).

You can estimate the parameters, (μ_i, σ_i^2), of the i-th feature by using the following equations.

- To estimate the mean, you will use:

$$\mu_i = \frac{1}{m}\sum_{j=1}^m x_i^{(j)}$$

- to the variance you will use:

$$\sigma_i^2 = \frac{1}{m}\sum_{j=1}^m (x_i^{(j)} - \mu_i)^2$$

The standard deviation (σ_i) is a measure of how spread out the values in a dataset are around the mean (μ_i). It quantifies the variability or dispersion of the data points. A smaller standard deviation indicates that the data points are close to the mean, while a larger standard deviation suggests greater variability or dispersion.

In [7]:
```python
# Feature characteristics of CCPP dataset with data records.
df.describe().T
```

Out[7]:

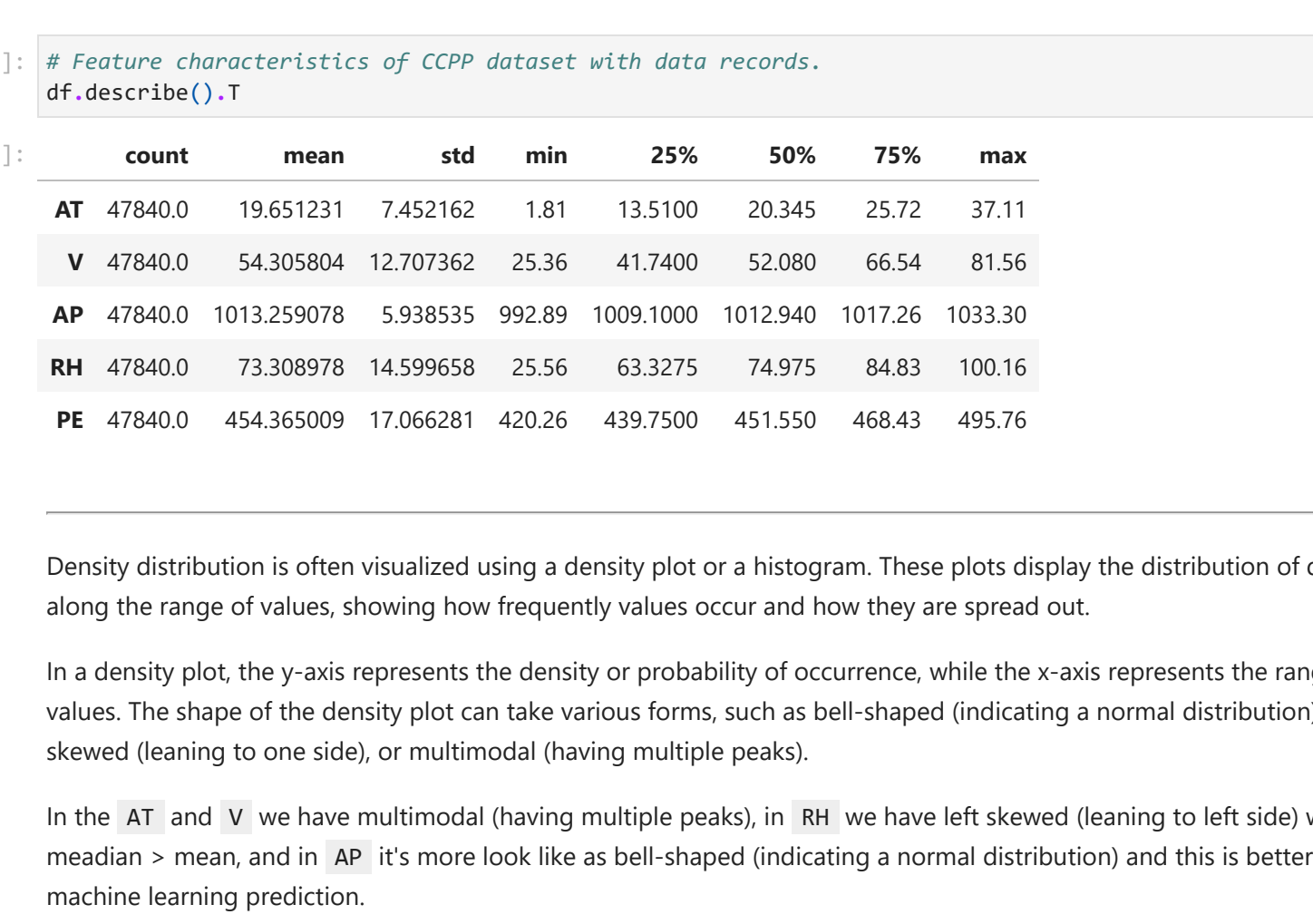
	count	mean	std	min	25%	50%	75%	max
AT	47840.0	19.651231	7.452162	1.81	13.5100	20.345	25.72	37.11
V	47840.0	54.305804	12.707362	25.36	41.7400	52.080	66.54	81.56
AP	47840.0	1013.259078	5.938535	992.89	1009.1000	1012.940	1017.26	1033.30
RH	47840.0	73.308978	14.599658	25.56	63.3275	74.975	84.83	100.16
PE	47840.0	454.365009	17.066281	420.26	439.7500	451.550	468.43	495.76

Density distribution is often visualized using a density plot or a histogram. These plots display the distribution of data along the range of values, showing how frequently values occur and how they are spread out.

In a density plot, the y-axis represents the density or probability of occurrence, while the x-axis represents the range of values. The shape of the density plot can take various forms, such as bell-shaped (indicating a normal distribution), skewed (leaning to one side), or multimodal (having multiple peaks).

In the `AT` and `V` we have multimodal (having multiple peaks), in `RH` we have left skewed (leaning to left side) where meadian > mean, and in `AP` it's more look like as bell-shaped (indicating a normal distribution) and this is better for machine learning prediction.

In [12]:
```python
import warnings
# Filter and suppress warnings
warnings.filterwarnings("ignore")

# distribution of the variables
plt.style.use('fivethirtyeight')
fig, axes = plt.subplots(2, 2, figsize=(16, 14))

for axes, col in zip(axes.flatten(), df[:-1]):
    sns.distplot(df[col], ax=axes, kde=True,
                 hist_kws={"alpha": 0.6},color='b').set(title=f"{col} Distribution")

plt.tight_layout()
sns.set(font_scale=1.5)
sns.despine()
plt.show()
```



A correlation matrix is a table that displays the correlation coefficients between multiple variables in a dataset. Correlation coefficients quantify the strength and direction of the linear relationship between pairs of variables. The matrix provides a comprehensive overview of how variables are related to each other, allowing for insights into patterns, dependencies, and potential associations within the data.

A correlation matrix typically takes the form of a square matrix, where each row and column represents a variable, and the intersection of a row and column represents the correlation coefficient between the corresponding variables. The diagonal elements of the matrix represent the correlation of a variable with itself, which is always 1. The remaining elements above or below the diagonal represent the correlation coefficients between pairs of different variables.

Correlation coefficients range from -1 to 1, where:

- A correlation coefficient of 1 indicates a perfect positive linear relationship, meaning that as one variable increases, the other variable increases proportionally.

- A correlation coefficient of -1 indicates a perfect negative linear relationship, meaning that as one variable increases, the other variable decreases proportionally.

- A correlation coefficient of 0 indicates no linear relationship, suggesting that the variables are independent of each other.

the variance you will use:

$$var = \sigma_i^2 = \frac{1}{m}\sum_{j=1}^m (x_i^{(j)} - \mu_i)^2$$

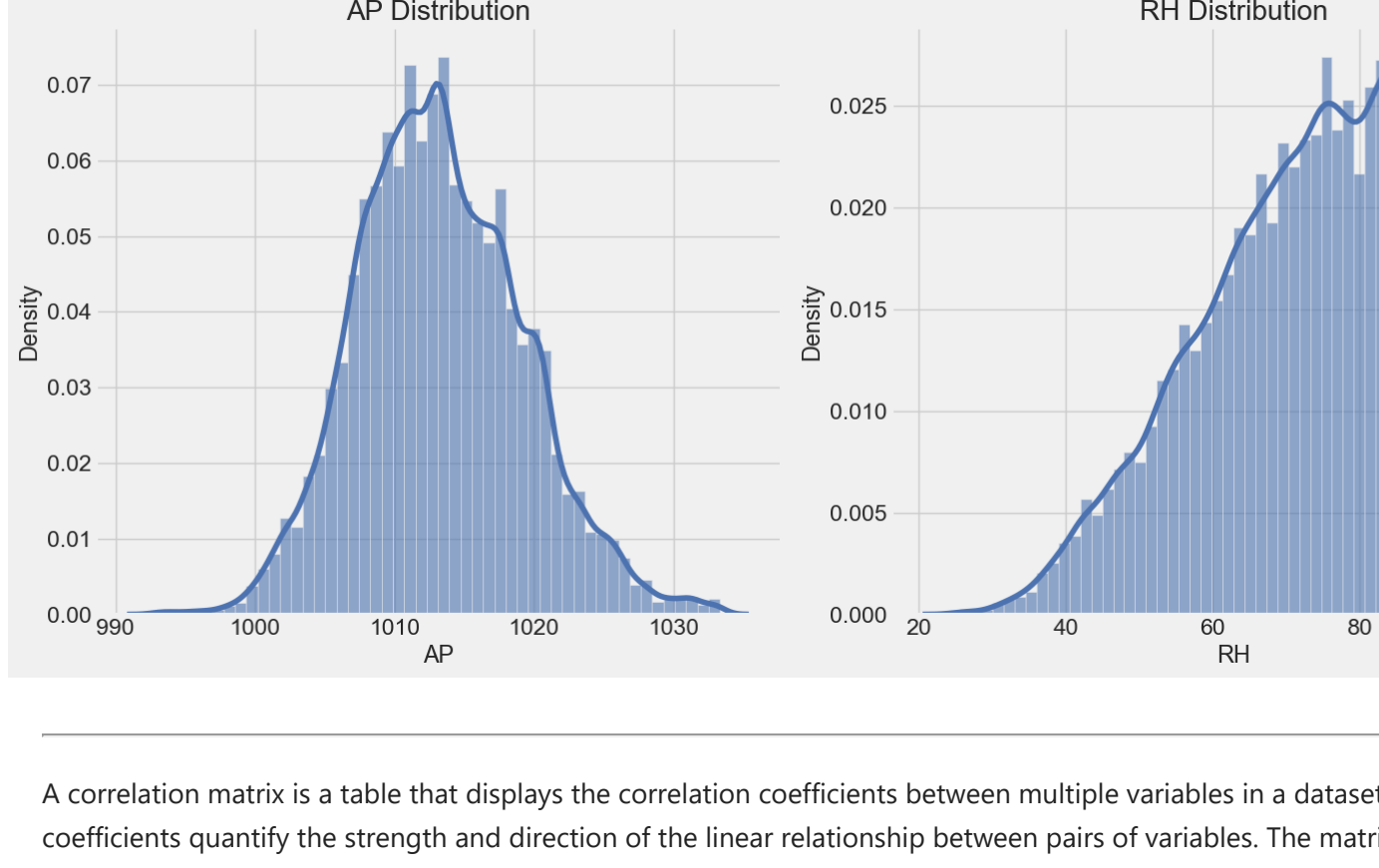
The Correlation Matrix:

$$cor(x, y) = (\frac{cov(x,y)}{\sqrt(var(x) var(y)})$$

the covariance $cov(x, y)$ represents the strength and direction of the linear relationship between pairs of variables.

While correlation measures the strength and direction of a linear relationship, covariance measures the extent to which two variables change together. A positive covariance indicates that the variables tend to increase or decrease together, while a negative covariance suggests that as one variable increases, the other tends to decrease.

In [14]:
```python
# correlation matrix
plt.style.use('fivethirtyeight')
plt.figure(figsize=(8, 5))
sns.set(font_scale=0.9)

sns.heatmap(df.corr(), annot=True, cmap='coolwarm', vmin=-1, vmax=1,annot_kws={"size": 12},
            cbar_kws={"fraction": 0.046})

plt.title("Correlation Matrix", fontsize=15)
sns.despine()
plt.show()
```



	AT	V	AP	RH	PE
AT	1	0.84	-0.51	-0.54	-0.95
V	0.84	1	-0.41	-0.31	-0.87
AP	-0.51	-0.41	1	0.1	0.52
RH	-0.54	-0.31	0.1	1	0.39
PE	-0.95	-0.87	0.52	0.39	1

- We can see from above correlation matrix that the `v` and `AT` features are highly correlated to the target variable but in negative correlation with `PE`.

- We can see also that there is a strong positive correlation between `V` and `AT`

As we can see from the plots above, there is a `linear relationship` between `PE` (target variable) and the other features. This is a good sign as it means that we can use **linear regression** (multivariate linear regression) to model the relationship between the features and the target variable.

- the equation is:

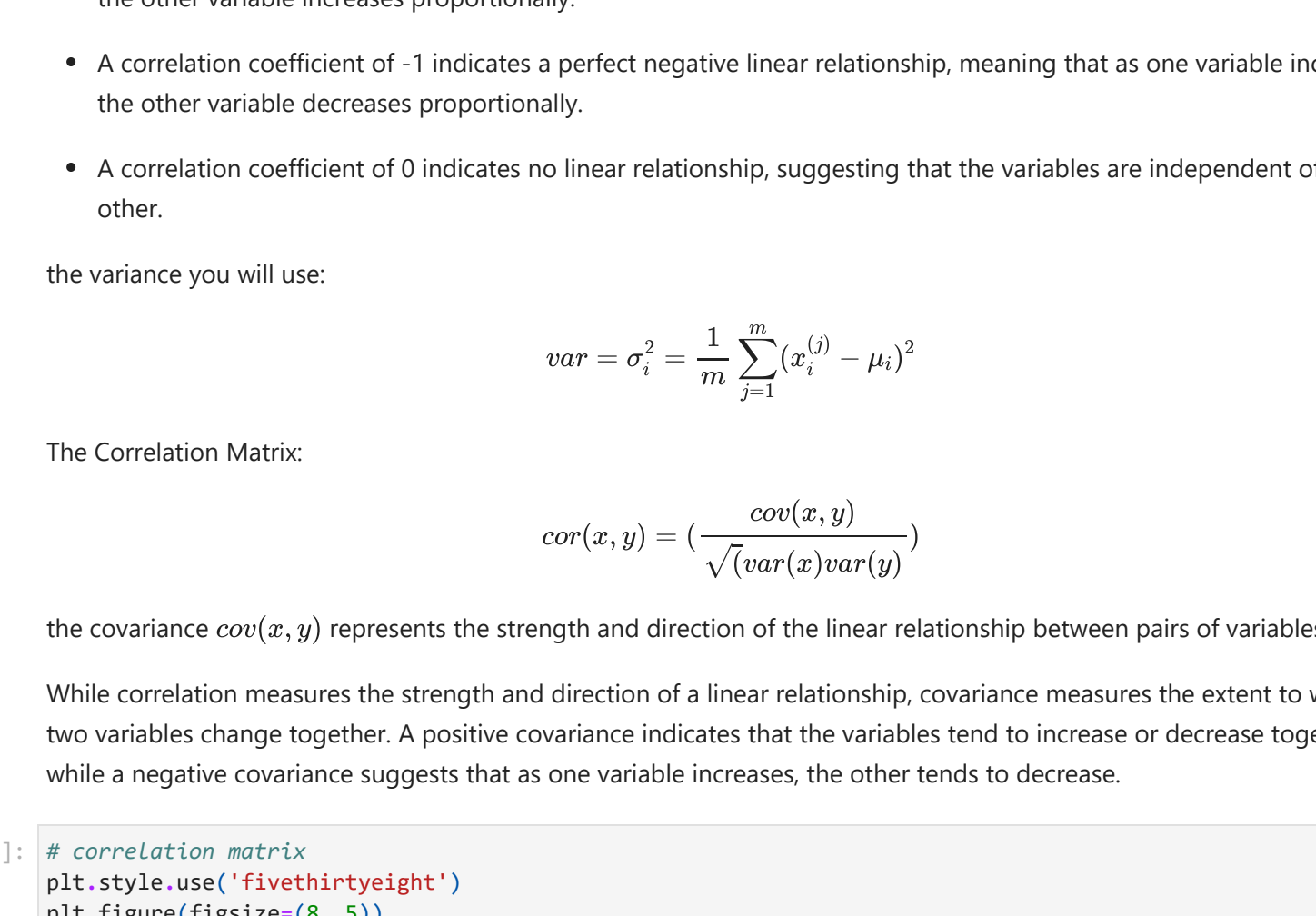
$$\hat{y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4$$

$$\hat{y} = \sum_{i=1}^{4} \beta_i X_i + \beta_0$$

- where X_1, X_2, X_3, X_4 are the features and $\hat{y}$ is the target predicted variable,

NOTE : β_0 is the intercept and β_1 is the coefficient of the feature.

In [15]:
```python
# check the linearization of the data
plt.style.use('fivethirtyeight')
plt.figure(figsize=(18, 15))

for i, col in enumerate(df.columns[:-1]):
    plt.subplot(2, 2, i+1)
    sns.regplot(x="PE", y=col, data=df, scatter_kws={"color": "b"},
                line_kws={"color": "r"}).set(title=f"{col} vs PE")

sns.despine()
plt.show()
```



4.2 Cleaning & Preprocessing Data For ML Models

4.2.1 Removing NULL values

We have here in our data set 4 null values just only, so it's good

In [17]:
```python
df.isnull().sum()
```

Out[17]:
```
AT    4
V     4
AP    4
RH    4
PE    4
dtype: int64
```

In [18]:
```python
df1 = df.dropna()
```

In [19]:
```python
df1.isnull().sum()
```

Out[19]:
```
AT    0
V     0
AP    0
RH    0
PE    0
dtype: int64
```

4.2.2 Removing Outliers

Outliers in a dataset refer to observations or data points that significantly deviate from the overall pattern or distribution of the data.

These outliers can have an impact on regression algorithms, which are used to model relationships between variables and make predictions.

Outliers can cause several problems in regression analysis:

- Distorted regression line: Outliers, especially those far from the majority of data points, can have a substantial influence on the estimated regression line. They can pull the line towards them, causing it to deviate from the true underlying relationship between the variables.

- Biased coefficient estimates: Outliers can lead to biased coefficient estimates in regression models. The presence of outliers can distort the estimation of regression coefficients, making them less representative of the true relationship between the variables.

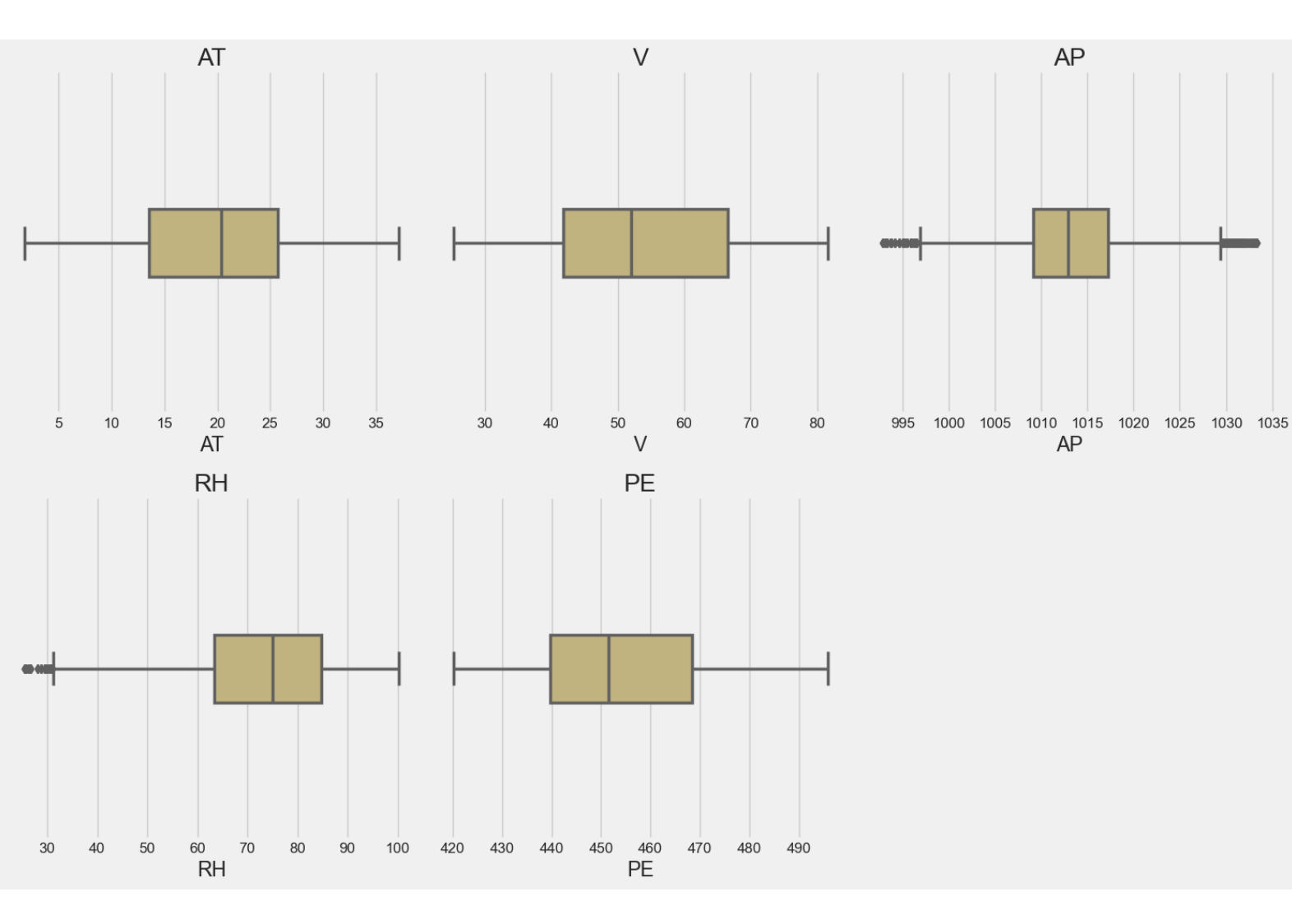
- Increased residual errors: Outliers can result in large differences between the predicted and actual values, leading to increased residual errors. Residual errors are the differences between the observed and predicted values of the dependent variable. Outliers can have a disproportionate impact on the residuals, reducing the accuracy of the regression model.

- Violation of assumptions: Regression algorithms rely on certain assumptions, such as linearity, normality, and constant variance of errors. Outliers can violate these assumptions, leading to inaccurate model assumptions and potentially invalidating the results of the regression analysis.

- Reduced predictive performance: Outliers can adversely affect the predictive performance of regression models. When outliers are present in the training data, the model might learn to overly accommodate these extreme values, leading to poor generalization and inaccurate predictions on new, unseen data.

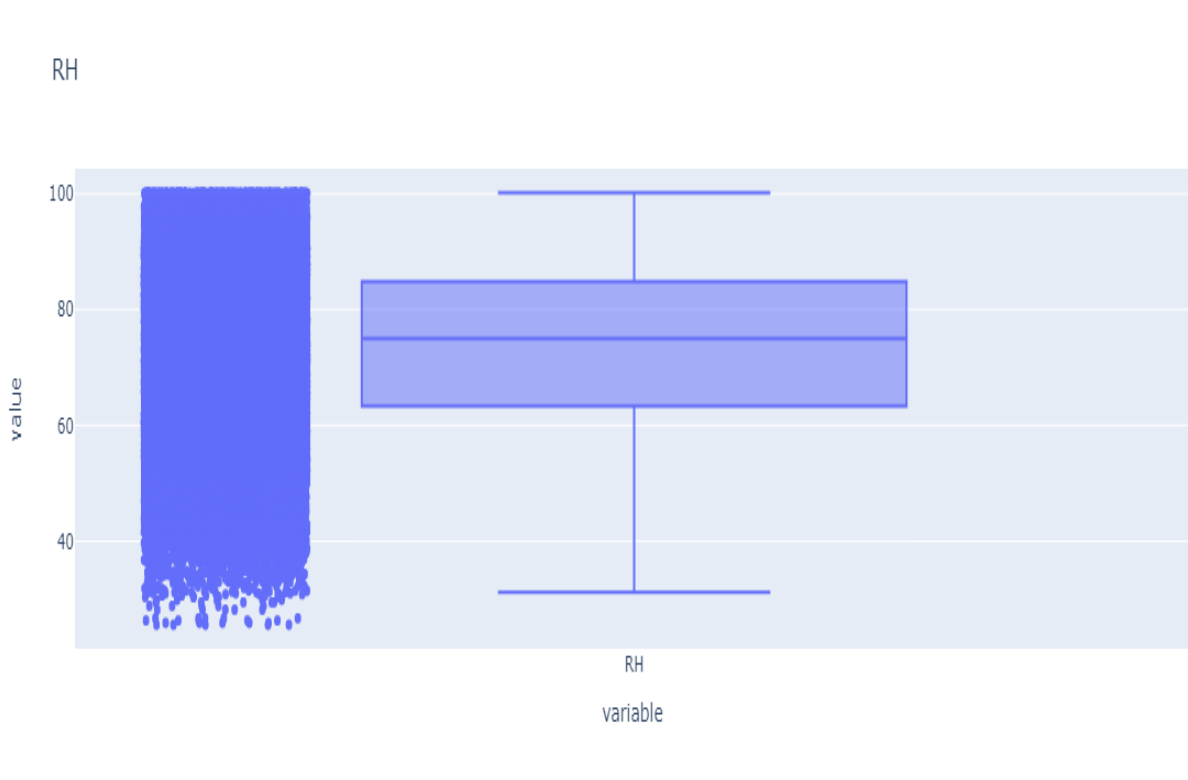
As we see that there is an outlier in the `RH` and `AP` feature in our data set. we need to remove it.

In [30]:
```python
plt.style.use('fivethirtyeight')
plt.figure(figsize=(15, 10))

for i, col in enumerate(df1.columns):
    plt.subplot(2, 3, i+1)
    sns.boxplot(df1[col],color='y',width=0.2,dodge=True,linewidth=2.5).set_title(col)

plt.tight_layout()
sns.set(font_scale=1.1)
sns.despine()
plt.show()
```

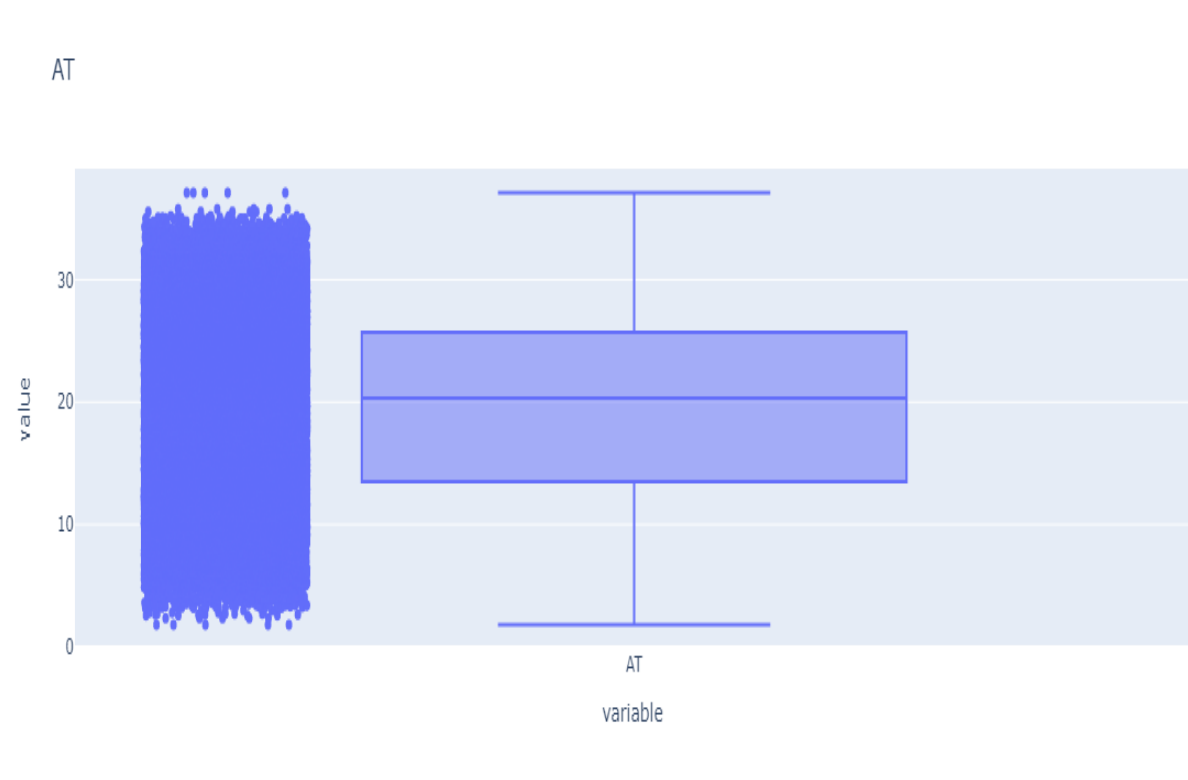


- See the outliers in data Distribution

- RH Distribution



- AP Distribution



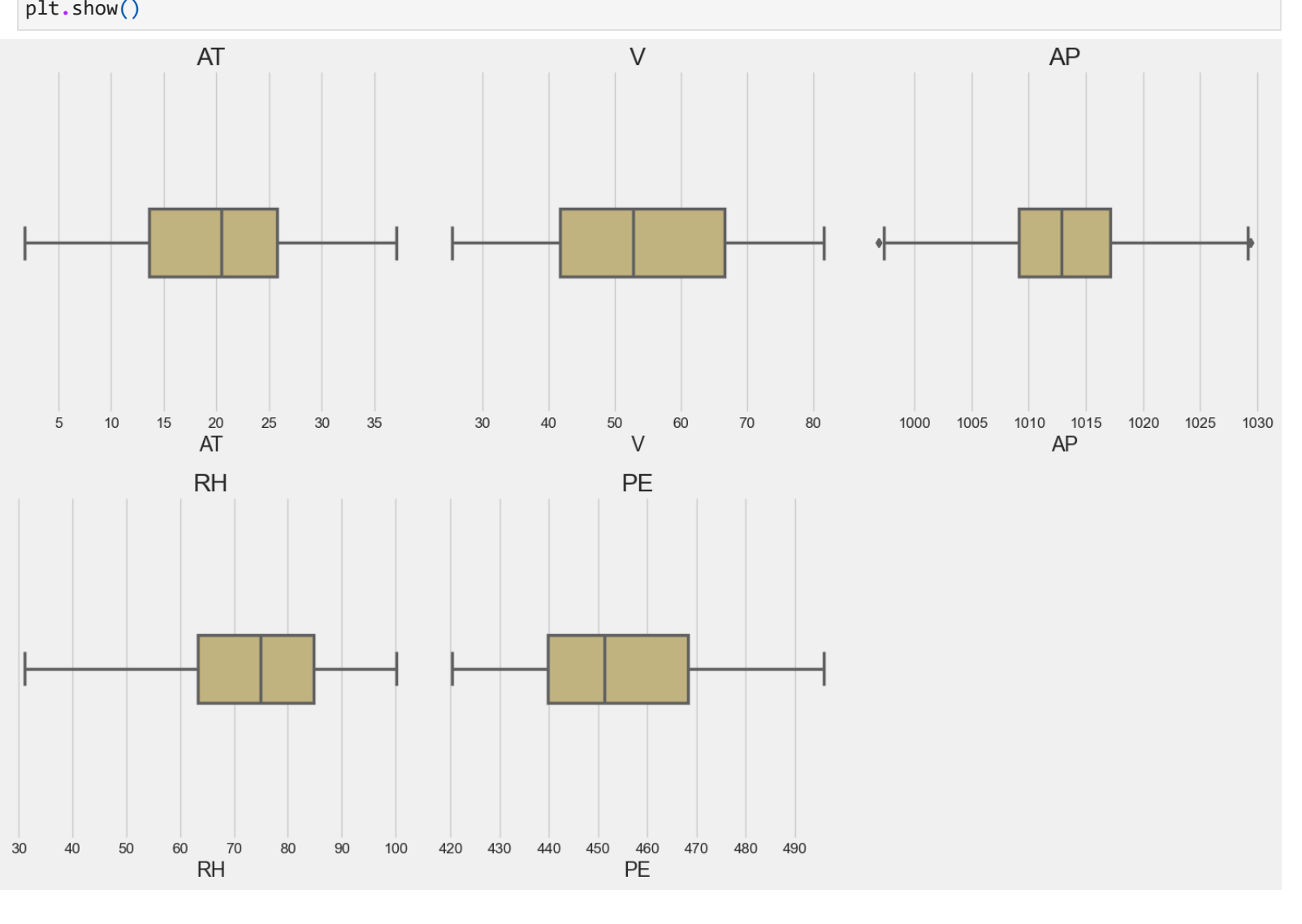
In [ ]:
```python
# MeTHOD FOR REMOVING OUTLIERS

"""
# Method 1: Z-Score
z_scores = (df - df.mean()) / df.std()
threshold = 2  # Set the z-score threshold for outlier detection
df_no_outliers = df[(z_scores < threshold).all(axis=1)]


# Method 2: Percentiles or IQR
q1 = df.quantile(0.25)
q3 = df.quantile(0.75)
iqr = q3 - q1
lower_bound = q1 - 1.5 * iqr
upper_bound = q3 + 1.5 * iqr
df_no_outliers = df[((df > lower_bound) & (df < upper_bound)).all(axis=1)]

"""
```

In [42]:
```python
# Let's use the Percentiles or IQR Method
q1 = df1.quantile(0.25)
q3 = df1.quantile(0.75)
iqr = q3 - q1

lower_bound = q1 - 1.5 * iqr
upper_bound = q3 + 1.5 * iqr

# save it in new dataframe
df2 = df1[((df1 > lower_bound) & (df1 < upper_bound)).all(axis=1)]


print(f'shape of df1: {df1.shape}')
print(f'shape of df2: {df2.shape}')
```

```
shape of df1: (47840, 5)
shape of df2: (47340, 5)
```

- As we see that we loss abot 500 data point represented as outliers

In [44]:
```python
plt.style.use('fivethirtyeight')
plt.figure(figsize=(15, 10))

for i, col in enumerate(df2.columns):
    plt.subplot(2, 3, i+1)
    sns.boxplot(df2[col],color='y',width=0.2,dodge=True,linewidth=2.5).set_title(col)

plt.tight_layout()
sns.set(font_scale=1.1)
sns.despine()
plt.show()
```



5. ML Models

5.1 Prepare data for ML models

5.1.1 Split data

Splitting the data into training, validation, and test sets is an essential step in machine learning model development.

The purpose of this split is to evaluate and fine-tune the model's performance on unseen data.

Here's a common approach to splitting the data:

- Training set: This is the largest portion of the data used for training the machine learning model. It is used to establish the patterns and relationships between the input features and the target variable. Typically, we use 60% of the data is allocated to the training set.

- Validation set: The validation set is used to fine-tune the model's hyperparameters and evaluate its performance during the training process. It helps in selecting the best-performing model based on the validation metrics. Typically, we use 20% of the data is allocated to the validation set.

- Test set: The test set is used to assess the final performance of the trained model on unseen data. It provides an unbiased evaluation of the model's generalization ability. The test set should only be used once, after the model is completely trained and tuned. Typically, we use 20% of the data is allocated to the test set.

In [45]:
```python
DF_PATH = "train.pkl"
DF_PATH1 = "test.pkl"
```

In [48]:
```python
train = pd.read_pickle(DF_PATH)
test = pd.read_pickle(DF_PATH1)

print(f' train set represent 80% of data: {train.shape}')
print(f' test set represent 20% of data: {test.shape}')
```

```
train set represent 80% of data: (37872, 5)
test set represent 20% of data: (9468, 5)
```

In [47]:
```python
train.head(10)
```

Out[47]:

	AT	V	AP	RH	PE
5444	6.75	39.40	1011.28	90.84	483.77
44734	10.08	41.16	1023.14	96.03	469.17
39601	14.32	44.60	1013.85	68.13	466.36
2928	19.04	51.86	1018.05	79.01	458.64
29411	29.17	67.45	1014.10	46.85	435.08
24090	25.69	59.54	1004.05	81.69	439.06
42640	30.22	74.90	1003.57	67.15	434.17
32239	10.40	40.43	1025.46	75.09	492.09
34189	24.13	73.18	1012.62	91.52	436.22
43860	18.26	69.94	1004.10	97.39	443.59

In [49]:
```python
# remove target feature
X = train.drop('PE',axis=1)
y = train['PE']
```

- Split 80% of data into train set 60% and validation set 20%

In [50]:
```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

In [51]:
```python
# split train data into train set and validation set
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

print(f'--> X_train: {X_train.shape}')
print(f'--> X_val: {X_val.shape}')
print(f'--> y_train: {y_train.shape}')
print(f'--> y_val: {y_val.shape}')
```

```
--> X_train: (30297, 4)
--> X_val: (7575, 4)
--> y_train: (30297,)
--> y_val: (7575,)
```

5.1.2 Feature Scaling

Feature scaling is a data preprocessing technique used in machine learning to bring all features of the dataset onto a similar scale.

It aims to standardize or normalize the features, ensuring that no single feature dominates the learning algorithm due to its larger magnitude.

Standardization (Z-score normalization): This method scales the features to have zero mean and unit variance. Each feature is transformed by subtracting the mean of the feature and dividing by its standard deviation. Standardization preserves the shape of the distribution and works well when the data does not have outliers.

it is usually a good idea to perform feature scaling to help your model converge faster. This is especially true if your input features have widely different ranges of values.

You will only use x for this first model but it's good to practice feature scaling now so you can apply it later. For that, you will use the `StandardScaler` class from scikit-learn. This computes the z-score of your inputs.

the z-score is given by the equation:

$$z = \frac{x - \mu}{\sigma}$$

where μ is the mean of the feature values and σ is the standard deviation.

The importance of feature scaling in machine learning cannot be overstated, as it directly impacts the performance and convergence of many algorithms. Here are some key reasons why feature scaling is crucial:

- Improves Convergence: Many machine learning algorithms rely on optimization techniques that aim to minimize a loss function during training. Feature scaling helps the optimization process converge faster, as features on similar scales allow the algorithm to reach the optimal solution more efficiently.

- Equalizes Feature Influence: When features have different scales, those with larger magnitudes can dominate the learning process. By scaling the features, each one contributes more equally to the model's training, avoiding biased results.

- Avoids Numerical Instabilities: Some algorithms, such as gradient descent-based methods, may encounter numerical instabilities when dealing with large differences in feature scales. Feature scaling helps to stabilize the computations and avoids overflow/underflow issues.

- Improves Model Performance: Scaling can lead to improved model performance, especially in algorithms sensitive to feature scales, such as support vector machines, K-nearest neighbors, and neural networks.

- Interpretability: Scaling can improve the interpretability of the model, as the model coefficients represent the relative importance of the features when they are on the same scale.

- Regularization: In regularization techniques like L1 or L2 regularization, feature scaling ensures that the regularization term applies uniformly to all features, leading to fairer regularization penalties.

- Consistent Preprocessing: Feature scaling provides a consistent preprocessing step, which can be essential when comparing different models or applying the same model to new data.

In [81]:
```python
print(f"Temperature Max, Min pre normalization for X_train: {X_train['AT'].max()}, {X_train['AT'].min()}")
print(f"Temperature Max, Min pre normalization for X_val: {X_val['AT'].max()}, {X_val['AT'].min()}")
```

```
Temperature Max, Min pre normalization for X_train: 37.11, 1.81
Temperature Max, Min pre normalization for X_val: 37.11, 1.81
```

In [52]:
```python
# scale
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
```

In [88]:
```python
import numpy as np
print(f" Temperature Max post normalization for X_train: {np.max(X_train_scaled[:,0])}")
print(f" Temperature Min post normalization for X_train: {np.min(X_train_scaled[:,0])}")
```

```
Temperature Max post normalization for X_train: 2.334411080307789
Temperature Min post normalization for X_train: -2.4163464525398117
```

Import libraries

In [126…
```python
# import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from lightgbm import LGBMRegressor
from xgboost import XGBRegressor
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error, make_scorer

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.activations import relu,linear
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras.optimizers import Adam

import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)

tf.keras.backend.set_floatx('float64')
tf.autograph.set_verbosity(0)
```

We use framework optimization GridSearchCV to get best value of it's hyperparameter in seperate notebook you can check the full work in Github

5.2 Linear Regression

In [91]:
```python
RMSE_DIC = {}
```

In [92]:
```python
linear_model = LinearRegression()
# fit on train data
model_fit = linear_model.fit(X_train_scaled,y_train)

# predict on train data
prediction = model_fit.predict(X_val_scaled)
```

In [93]:
```python
def RMSE(y,y_hat):
  return np.sqrt(mean_squared_error(y,y_hat))
```

In [94]:
```python
print(f"RMSE: {RMSE(y_val,prediction)}")
print(f"MAE: {mean_absolute_error(y_val,prediction)}")
```

```
RMSE: 4.562937139838695
MAE: 3.637839123894593
```

In [95]:
```python
RMSE_DIC["LinearRegression"] = 4.5629
```

5.3 DecisionTreeRegressor

In [96]:
```python
# Create the Decision Tree Regressor
decision_tree_model = DecisionTreeRegressor(min_samples_split = 2,
                                            max_depth = 64,
                                            random_state = 42).fit(X_train_scaled,y_train)
```

In [97]:
```python
print(f"RMSE 0f DecisionTreeRegressor:
      {RMSE(y_val, decision_tree_model.predict(X_val_scaled)):.4f}")
print(f"MAE:
      {mean_absolute_error(y_val, decision_tree_model.predict(X_val_scaled)):.4f}")
```

```
RMSE 0f DecisionTreeRegressor: 0.4593
MAE: 0.0452
```

In [98]:
```python
RMSE_DIC["DecisionTreeRegressor"] = 0.4593
```

5.4 RandomForestRegressor

In [99]:
```python
# Create the RandomForestRegressor
RandomForest_model = RandomForestRegressor(min_samples_split = 2,
                                           max_depth = 64,
                                           random_state = 42).fit(X_train_scaled,y_train)
```

In [100…
```python
print(f"RMSE 0f RandomForestRegressor:
      {RMSE(y_val, RandomForest_model.predict(X_val_scaled)):.4f}")
print(f"MAE of RandomForestRegressor:
      {mean_absolute_error(y_val, RandomForest_model.predict(X_val_scaled)):.4f}")
```

```
RMSE 0f RandomForestRegressor: 0.6732
MAE of RandomForestRegressor: 0.2985
```

In [101…
```python
RMSE_DIC["RandomForestRegressor"] = 0.6732
```

5.5 LGBMRegressor

In [104…
```python
lgb_clf = LGBMRegressor(class_weight = 'balanced',
                        learning_rate = 0.9,
                        max_depth = 20,
                        n_estimators = 700,
                        random_state = 42,
                        )
lgb_clf.fit(X_train_scaled, y_train,
           eval_set = [(X_train_scaled, y_train),(X_val_scaled, y_val)],
           early_stopping_rounds = 300,
           verbose=0,
           eval_metric='rmse'
          )
```

Out[104]:
```
LGBMRegressor(class_weight='balanced', learning_rate=0.9, max_depth=20,
              n_estimators=700, random_state=42)
```

In [105…
```python
RMSE_DIC["LGBMRegressor"] = 0.5692
```

5.6 XGBRegressor

In [107…
```python
xgb_rg = XGBRegressor(learning_rate = 0.1,
                      max_depth = 70,
                      n_estimators = 500,
                      random_state = 42,
                      )
xgb_rg.fit(X_train_scaled, y_train,
           eval_set = [(X_val_scaled, y_val)],
           early_stopping_rounds = 300,
           verbose=0,
           eval_metric=['rmse','mae']
          )
```

Out[107]:
```
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, early_stopping_rounds=None,
             enable_categorical=False, eval_metric=None, feature_types=None,
             gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
             interaction_constraints=None, learning_rate=0.1, max_bin=None,
             max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=70, max_leaves=None,
             min_child_weight=None, missing=nan, monotone_constraints=None,
             n_estimators=500, n_jobs=None, num_parallel_tree=None,
             predictor=None, random_state=42, ...)
```

In [109…
```python
RMSE_DIC["XGBRegressor"] = 0.3080
```

5.7 SVM

In [110…
```python
regr = svm.SVR()
# fit
regr.fit(X_train_scaled, y_train)
# predict
y_hat = regr.predict(X_val_scaled)

print(f"RMSE: {RMSE(y_val, y_hat)}")
print(f"MAE: {mean_absolute_error(y_val, y_hat)}")
```

```
RMSE: 4.101138865225424
MAE: 3.0602853705365365
```

In [111…
```python
RMSE_DIC['SVM'] = 4.1011
```

5.8 Ensembles Voting

In [112…
```python
from sklearn.ensemble import VotingRegressor


voting_rg = VotingRegressor(estimators = [
    ('Dt',decision_tree_model ),
    ('rf',RandomForest_model ),
    ('lgb',lgb_clf),
    ('xgb',xgb_rg),
    ])

voting_rg.fit(X_train_scaled, y_train)
```

Out[112]:
```
VotingRegressor(estimators=[('Dt',
                             DecisionTreeRegressor(max_depth=64,
                                                   random_state=42)),
                            ('rf',
                             RandomForestRegressor(max_depth=64,
                                                   random_state=42)),
                            ('lgb',
                             LGBMRegressor(class_weight='balanced',
                                           learning_rate=0.9, max_depth=20,
                                           n_estimators=700, random_state=42)),
                            ('xgb',
                             XGBRegressor(base_score=None, booster=None,
                                          callbacks=None,
                                          colsample_bylevel=None,
                                          colsam...
                                          gamma=None, gpu_id=None,
                                          grow_policy=None,
                                          importance_type=None,
                                          interaction_constraints=None,
                                          learning_rate=0.1, max_bin=None,
                                          max_cat_threshold=None,
                                          max_cat_to_onehot=None,
                                          max_delta_step=None, max_depth=70,
                                          max_leaves=None,
                                          min_child_weight=None, missing=nan,
                                          monotone_constraints=None,
                                          n_estimators=500, n_jobs=None,
                                          num_parallel_tree=None,
                                          predictor=None, random_state=42, ...))])
```

In [113…
```python
print(f"RMSE 0f stacking: {RMSE(y_val, voting_rg.predict(X_val_scaled)):.4f}")
print(f"MAE of stacking: {mean_absolute_error(y_val, voting_rg.predict(X_val_scaled)):.4f}")
```

```
RMSE 0f stacking: 0.3823
MAE of stacking: 0.1372
```

In [114…
```python
RMSE_DIC['Stacking'] = 0.3823
```

In [115…
```python
RMSE_DIC
```

Out[115]:
```
{'LinearRegression': 4.5629,
 'DecisionTreeRegressor': 0.4593,
 'RandomForestRegressor': 0.6732,
 'LGBMRegressor': 0.5692,
 'XGBRegressor': 0.308,
 'SVM': 4.1011,
 'Stacking': 0.3823}
```

> In such cases, the ensemble might not bring significant improvements as we see that `XGBRegressor` has better score.

- It's important to note that the performance of ensemble models can vary depending on the specific dataset and problem.

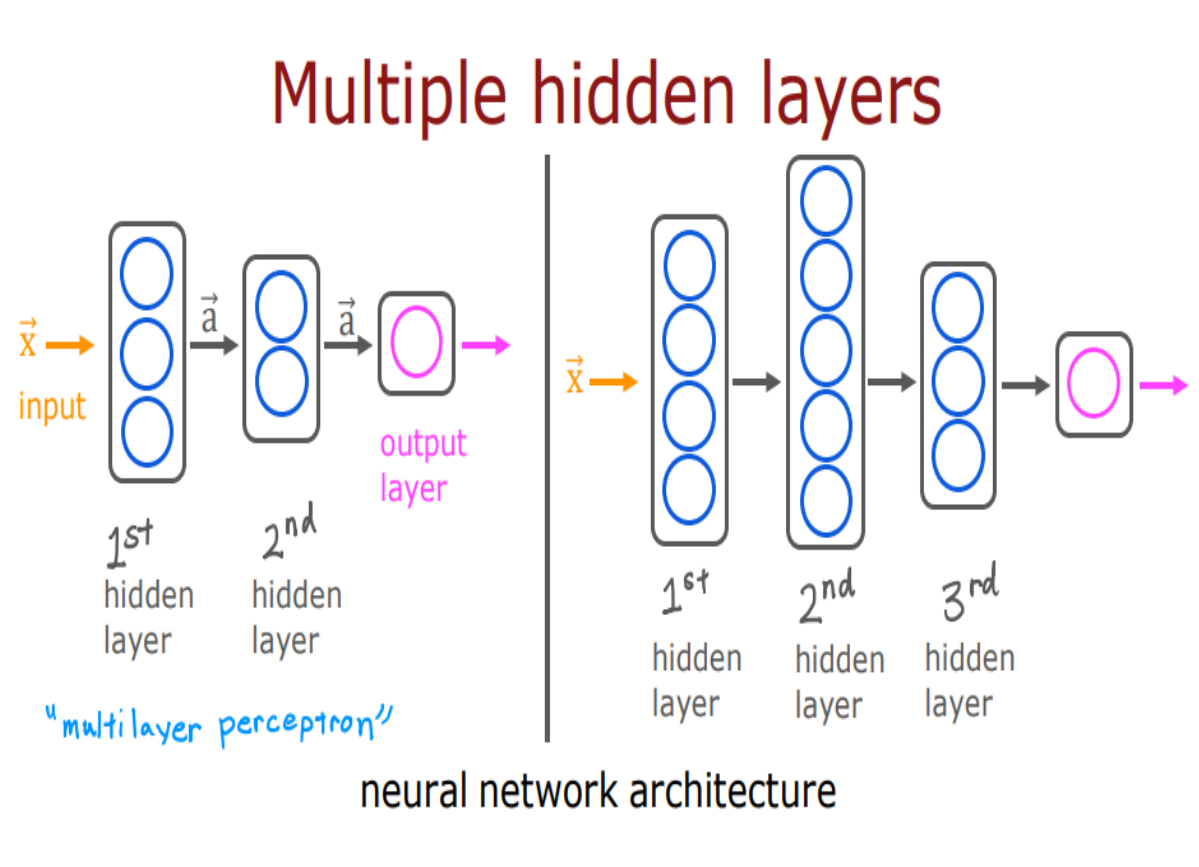
In [125…
```python
import plotly.express as px
# Example data
x = list(RMSE_DIC.keys())
y = list(RMSE_DIC.values())

# Create the bar plot
fig = px.bar(x=x, y=y, title='RMSE FOR MODELS',
             color = x, labels={'x': 'Models Name', 'y': 'RMSE'},
             width=1000, height=400)

# Display the plot
fig.show()
```

5.9 artificial neural network

It is a computational model inspired by the structure and functioning of biological neural networks, such as the human brain. An ANN consists of interconnected artificial neurons, or nodes, organized in layers. Each node receives input signals, performs a computation, and produces an output signal that may serve as input for other nodes.

The basic building block of an ANN is the artificial neuron, also known as a perceptron. It takes multiple inputs, applies weights to each input, sums them up, and passes the sum through an activation function to produce an output. The activation function introduces non-linearities and allows the network to learn complex relationships between inputs and outputs.

ANNs are capable of learning from data through a process called training. During training, the network adjusts the weights of its connections based on a specified learning algorithm and a set of training examples. This process enables the network to approximate or learn patterns, make predictions, and perform various tasks, including classification, regression, and pattern recognition.



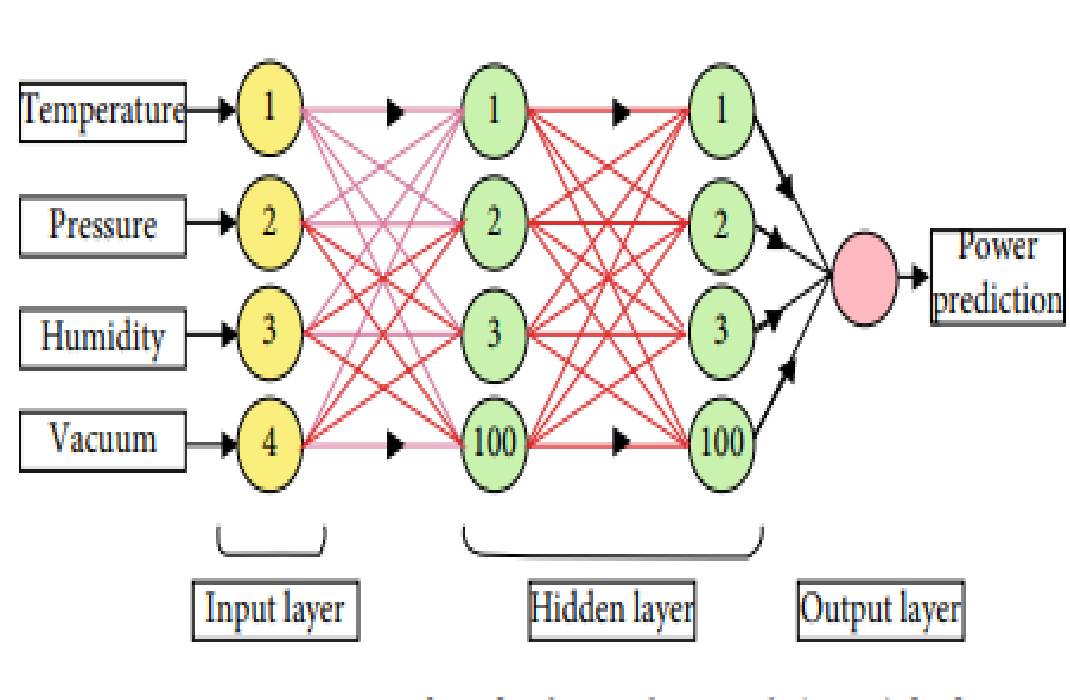


FIGURE 4: Structure of artificial neural network (ANN) [34].

5.9.1 Cost function for regularized linear regression

The equation for the cost function regularized linear regression is:

$$J(\mathbf{w},b) = \frac{1}{2m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=0}^{n-1} w_j^2 \tag{1}$$

where:

$$f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x}^{(i)} + b \tag{2}$$

Compare this to the cost function without regularization (which we implemented in a previous work above), which is of the form:

$$J(\mathbf{w},b) = \frac{1}{2m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2$$

The difference is the regularization term, $\frac{\lambda}{2m} \sum_{j=0}^{n-1} w_j^2$

Including this term encourages gradient descent to minimize the size of the parameters. Note, in this example, the parameter b is not regularized. This is standard practice.

5.9.2 Gradient descent with regularization

The basic algorithm for running gradient descent does not change with regularization, it is:

$$\begin{align*}
&\text{repeat until convergence:} \; \lbrace \\
&\; \; \;w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \; & \text{for j := 0..n-1} \tag{1} \\
&\; \; \; \; \;b = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \\
&\rbrace
\end{align*}$$

Where each iteration performs simultaneous updates on w_j for all j.

What changes with regularization is computing the gradients.

The gradient calculation for linear regression

$$\begin{align*}
\frac{\partial J(\mathbf{w},b)}{\partial w_j} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_{j}^{(i)} + \frac{\lambda}{m} w_j \tag{2} \\
\frac{\partial J(\mathbf{w},b)}{\partial b} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \tag{3}
\end{align*}$$

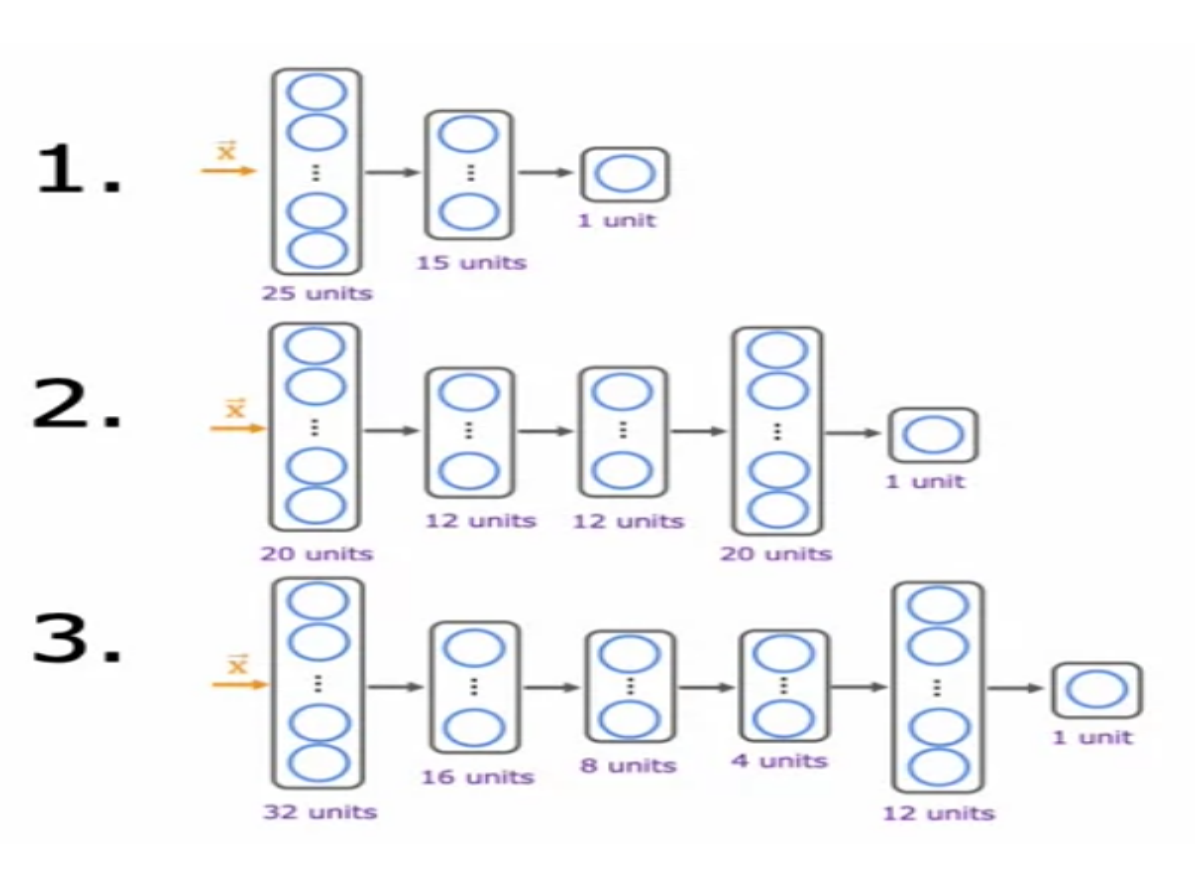
- m is the number of training examples in the data set
- $f_{\mathbf{w},b}(x^{(i)})$ is the model's prediction, while $y^{(i)}$ is the target
- For a linear regression model
  $f_{\mathbf{w},b}(x) = \mathbf{w} \cdot \mathbf{x} + b$

The term which adds regularization is $\frac{\lambda}{m} w_j$.

5.9.3 model selection process from different neural network architectures

The same model selection process can also be used when choosing between different neural network architectures.

we 'll loop over some different architectures below and evaluate RMSE on each one to get best architecture for our problems.



```
In [131…
# set seed for reproducibility
tf.random.set_seed(1234)

# Define a list of layer architectures to try
layer_architectures = [

    [Dense(units = 40, activation='relu', name = 'L1', input_shape=(4,)),
     Dense(units = 1, name = 'L11')],

    [Dense(units = 65, activation='relu', name = 'L2222', input_shape=(4,)),
     Dense(units = 1, name = 'L22222')],

     [Dense(units = 65, activation='relu', name = 'L999999', input_shape=(4,)),
      Dense(units = 25, activation='relu', name = 'L99999'),
      Dense(units = 1, name = 'L9999')],

    [Dense(units = 65, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L2', input_shape=(4,)),
     Dense(units = 1, name = 'L22')],

     [Dense(units = 65, activation='relu',
            kernel_regularizer = tf.keras.regularizers.l2(0.1),
            name = 'L9', input_shape=(4,)),
      Dense(units = 25, activation='relu',
            kernel_regularizer = tf.keras.regularizers.l2(0.1), name = 'L99'),
      Dense(units = 1, name = 'L999')],

    [Dense(units = 125, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L3', input_shape=(4,)),
     Dense(units = 65, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L33'),
     Dense(units = 1, name = 'L333')],

    [Dense(units = 65, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L4', input_shape=(4,)),
     Dense(units = 64, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L44'),
     Dense(units = 1, name = 'L444')],


    [Dense(units = 125, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L5', input_shape=(4,)),
     Dense(units = 64, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L55'),
     Dense(units = 1, name = 'L555')],


    [Dense(units = 164, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L6', input_shape=(4,)),
     Dense(units = 125, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L66'),
     Dense(units = 64, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L666'),
     Dense(units = 1, name = 'L6666')],

    [Dense(units = 200, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L7', input_shape=(4,)),
     Dense(units = 164, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L77'),
     Dense(units = 125, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L777'),
     Dense(units = 64, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L7777'),
     Dense(units = 1, name = 'L77777')],

    [Dense(units = 300, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L8', input_shape=(4,)),
     Dense(units = 200, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L88'),
     Dense(units = 164, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L888'),
     Dense(units = 125, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L8888'),
     Dense(units = 64, activation='relu',
           kernel_regularizer = tf.keras.regularizers.l2(0.1),
           name = 'L88888'),
     Dense(units = 1, name = 'L888888')],

    # Add more layer configurations as desired
]

best_model = None
best_loss = float('inf')

# Loop over each layer architecture and train/evaluate the model
for architecture in layer_architectures:
    model = tf.keras.Sequential(architecture)

    # Compile the model
    model.compile(optimizer= Adam(learning_rate=0.01), loss= MeanSquaredError())
    # Fit the model
    model.fit(X_train_scaled, y_train, epochs=50, verbose=0)
    # Predict
    y_pred = model.predict(X_val_scaled)

    loss = RMSE(y_val, y_pred)

    print(f"Architecture: {architecture}")
    print(f"Loss: {loss:.4f}")

    # Track the best model based on the lowest loss
    if loss < best_loss:
        best_loss = loss
        best_model = model



print(f'<-- best_loss: {best_loss}')
print(f'<-- best_model: {best_model}')
```

```
237/237 [==============================] - 1s 2ms/step
Architecture: [<keras.layers.core.dense.Dense object at 0x0000017ECE31A940>, <keras.layers.core.dense.Dense object at 0x0000017ECC2B1A60>]
Loss: 4.2179
237/237 [==============================] - 1s 2ms/step
Architecture: [<keras.layers.core.dense.Dense object at 0x0000017E8443FEE0>, <keras.layers.core.dense.Dense object at 0x0000017ECB228FA0>]
Loss: 4.1791
237/237 [==============================] - 0s 2ms/step
Architecture: [<keras.layers.core.dense.Dense object at 0x0000017E8444E2B0>, <keras.layers.core.dense.Dense object at 0x0000017E84446430>, <keras.layers.core.dense.Dense object at 0x0000017E8443F0D0>]
Loss: 4.3141
237/237 [==============================] - 1s 2ms/step
Architecture: [<keras.layers.core.dense.Dense object at 0x0000017E84446F10>, <keras.layers.core.dense.Dense object at 0x0000017E84446EE0>]
Loss: 4.3159
237/237 [==============================] - 1s 2ms/step
Architecture: [<keras.layers.core.dense.Dense object at 0x0000017E84443760>, <keras.layers.core.dense.Dense object at 0x0000017E844432E0>, <keras.layers.core.dense.Dense object at 0x0000017E84443370>]
Loss: 4.5455
237/237 [==============================] - 1s 3ms/step
Architecture: [<keras.layers.core.dense.Dense object at 0x0000017E844564C0>, <keras.layers.core.dense.Dense object at 0x0000017E84456940>, <keras.layers.core.dense.Dense object at 0x0000017E844568B0>]
Loss: 4.4240
237/237 [==============================] - 0s 1ms/step
Architecture: [<keras.layers.core.dense.Dense object at 0x0000017E84468C70>, <keras.layers.core.dense.Dense object at 0x0000017E8451C130>, <keras.layers.core.dense.Dense object at 0x0000017E84468CD0>]
Loss: 4.4307
237/237 [==============================] - 1s 2ms/step
Architecture: [<keras.layers.core.dense.Dense object at 0x0000017E8451C8B0>, <keras.layers.core.dense.Dense object at 0x0000017E8451CD30>, <keras.layers.core.dense.Dense object at 0x0000017E8451CCA0>]
Loss: 4.4729
237/237 [==============================] - 1s 3ms/step
Architecture: [<keras.layers.core.dense.Dense object at 0x0000017E845224C0>, <keras.layers.core.dense.Dense object at 0x0000017E84522940>, <keras.layers.core.dense.Dense object at 0x0000017E84522D30>, <keras.layers.core.dense.Dense object at 0x0000017E845229A0>]
Loss: 4.4496
237/237 [==============================] - 1s 4ms/step
Architecture: [<keras.layers.core.dense.Dense object at 0x0000017E845254C0>, <keras.layers.core.dense.Dense object at 0x0000017E84525940>, <keras.layers.core.dense.Dense object at 0x0000017E84525D30>, <keras.layers.core.dense.Dense object at 0x0000017E84525D60>, <keras.layers.core.dense.Dense object at 0x0000017E84525DC0>]
Loss: 4.4446
237/237 [==============================] - 1s 4ms/step
Architecture: [<keras.layers.core.dense.Dense object at 0x0000017E8452A8E0>, <keras.layers.core.dense.Dense object at 0x0000017E8452AD60>, <keras.layers.core.dense.Dense object at 0x0000017E8452AD90>, <keras.layers.core.dense.Dense object at 0x0000017E8452D580>, <keras.layers.core.dense.Dense object at 0x0000017E8452D970>, <keras.layers.core.dense.Dense object at 0x0000017E8452D610>]
Loss: 4.4043
<-- best_loss: 4.17910971073392
<-- best_model: <keras.engine.sequential.Sequential object at 0x0000017E8479C190>
```

Model two get best_loss with 4.1791.

As we see that more simple our architecture without regularization more the score we get, however, we more complex model we should use regularization parameter.

Neural network is better with using more complex data so here we see that it's not that good for our data here as we get onlt 4.17 score compared to XGBRegressor it's not that good, so using Neural network here is not sufficient

```
In [132…
RMSE_DIC["ANN"] = 4.1791
```

Result

```
In [136…
RMSE_DIC
```

```
Out[136]:
{'LinearRegression': 4.5629,
 'DecisionTreeRegressor': 0.4593,
 'RandomForestRegressor': 0.6732,
 'LGBMRegressor': 0.5692,
 'XGBRegressor': 0.308,
 'SVM': 4.1011,
 'Stacking': 0.3823,
 'ANN': 4.1791}
```

```
In [135…
import plotly.express as px
# Example data
x = list(RMSE_DIC.keys())
y = list(RMSE_DIC.values())

# Create the bar plot
fig = px.bar(x=x, y=y, title='RMSE FOR MODELS',
             color = x, labels={'x': 'Models Name', 'y': 'RMSE'},
             width=1000, height=550)

# Display the plot
fig.show()
```


6.Deployment

```
In [ ]:
```