**MACHINE LEARNING PROJECT REPORT**

**Objective**

The objective of the project is to identify large-scale electrical energy consumption , a more flexible power consumption so we can reduce energy costs and the impact on the environment as well as utilise periods with large environmental power generation and minimum load on the grid.

**Understanding the machine learning workflow**

We can define the machine learning workflow in 3 stages.

1. Gathering data
2. Data pre-processing
3. Researching the model that will be best for the type of data
4. Training and testing the model
5. Evaluation

**Gathering Data**

The process of gathering data depends on the type of project we desire to make, if we want to make an ML project that uses real-time data. The data set can be collected from various sources such as a file, database, sensor and many other such sources but the collected data cannot be used directly for performing the analysis process as there might be a lot of missing data, extremely large values, unorganized text data or noisy data. Therefore, to solve this problem Data Preparation is done.

Data pre-processing is one of the most important steps in machine learning. It is the most important step that helps in building machine learning models more accurately. In machine learning, there is an 80/20 rule. Every data scientist should spend 80% time for data pre-processing and 20% time to actually perform the analysis.

**Data Loading(From csv file)**

We read a dataset that is related to household power consumption for couple of years. Then, we split our data into training and test sets, create a model using training set, Evaluate your model using test set, and finally use model to predict unknown value

Libraries are imported,file is read using pandas library & datetime index is created using parse parameter & index\_col.

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

df=pd.read\_csv("hpc\_breakdowns.csv",parse\_dates=['Date'],index\_col=['Date'])

ptest=df.iloc[-200000:,:]

cdf=df.iloc[0:-200000,:]

cdf

Iloc & loc function extract or slice out the required rows & columns.

cdf.loc[cdf.duplicated(),:]

We can check duplicated rows in accordance with date time index.

As we know that data pre-processing is a process of cleaning the raw data into clean data, so that can be used to train the model. So, we definitely need data pre-processing to achieve good results from the applied model in machine learning and deep learning projects. Our main goal is to train the best performing model possible, using the pre-processed data.

Most of the real-world data is messy, some of these types of data are:

1. **Missing data:** Missing data can be found when it is not continuously created or due to technical issues in the application (IOT system).

2. **Noisy data:** This type of data is also called outliners, this can occur due to human errors (human manually gathering the data) or some technical problem of the device at the time of collection of data.

3. **Inconsistent data:** This type of data might be collected due to human errors (mistakes with the name or values) or duplication of data.

## Three Types of Data

1. Numeric e.g. income, age

2. Categorical e.g. gender, nationality

3. Ordinal e.g. low/medium/high

**Data Analysis(sample)**

A sample of 10000 rows taken for data analysis.

sample=cdf.iloc[:10000,:]

Now we plot relationships between features using seaborn library.

sns.pairplot(sample)

info() provided the details & type of feature or column values.

sample.info()

Describe() calculates different parameters of each column value such as mean,std deviation,quartile.

sample.describe()

Heatmap method in seaborn library is used to find the feature importances.

plt.figure(figsize=[15,5])

sns.heatmap(sample.corr(),annot=True)

plt.show()

isnull().sum() gives the total null values in each column.

sample.isnull().sum()

Boxplot() finds out the outlier values in a column in visualization.

sns.boxplot(sample['Voltage'])

sns.boxplot(sample['Global\_reactive\_power'])

sns.boxplot(sample['Global\_active\_power'])

**Data Cleaning**

Now we drop the unwanted columns based on feature importance by using drop method.

sample.drop(['Unnamed:0','Time','Sub\_metering\_1','Sub\_metering\_2','Sub\_metering\_3'],axis=1,inplace=True)

sample['Global\_active\_power'].median()

Median() is the majority value in a particular column.

Fill na values(missing values) with median().

sample['Global\_active\_power'].fillna(sample['Global\_active\_power'].median(),inplace=True)

sample['Global\_intensity'].fillna(sample['Global\_intensity'].median(),inplace=True)

**Data Transformation**

sample.info()

Output is global intensity column.

sy=sample['Global\_intensity']

Input:

SX=sample[['Global\_active\_power','Global\_reactive\_power','Voltage']]

**Pre processed Data Splitting**

Train/Test Split involves splitting the dataset into training and testing sets respectively, which are mutually exclusive. After which, you train with the training set and test with the testing set. This will provide a more accurate evaluation on out-of-sample accuracy because the testing dataset is not part of the dataset that have been used to train the data. It is more realistic for real world problems.

This means that we know the outcome of each data point in this dataset, making it great to test with! And since this data has not been used to train the model, the model has no knowledge of the outcome of these data points. So, in essence, it is truly an out-of-sample testing.

In Supervised learning, an AI system is presented with data which is labelled, which means that each data tagged with the correct label.

The supervised learning is categorized into 2 other categories which are “Classification” and “Regression”

**Classification**problem is when the target variable is **categorical**(i.e. the output could be classified into classes — it belongs to either Class A or B or something else).

While a **Regression**problem is when the target variable is **continuous**(i.e. the output is numeric)

Import libraries for splitting data.

import sklearn

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

from sklearn.linear\_model import LinearRegression

Linear Regression fits a linear model with coefficients B = (B1, ..., Bn) to minimize the 'residual sum of squares' between the independent x in the dataset, and the dependent y by the linear approximation.

X\_train, x\_test, y\_train, y\_test=train\_test\_split(SX,sy,test\_size=0.20,random\_state=26)

**Modelling**

We apply linear regression model & train the model using the train dataset.

regr=LinearRegression()

regr.fit(X\_train,y\_train)

prediction=regr.predict(x\_test)

regr.score(X\_train,y\_train)

Rms=root mean square(it calculates difference of actual & predicted outputs)

**Results**

from sklearn.metrics import mean\_squared\_error

from math import sqrt

rms=sqrt(mean\_squared\_error(y\_test,prediction))

rms

**Full dataset applying Linear Algorithm**

Data Analysi

Pairplot method shows any linear or non linear relationships between featues.

sns.pairplot(cdf)

cdf.info()

cdf.corr()

plt.figure(figsize=[15,5])

sns.heatmap(cdf.corr(),annot=True)

plt.show()

cdf.isnull().sum()

cdf.describe()

sns.boxplot(cdf['Voltage'])

sns.boxplot(cdf['Global\_active\_power'])

sns.boxplot(cdf['Global\_reactive\_power'])

sns.boxplot(cdf['Global\_intensity'])

**Data Cleaning**

cdf.drop(['Unnamed:0','Time','Sub\_metering\_1','Sub\_metering\_2','Sub\_metering\_3'],axis=1,inplace=True)

cdf.shape

cdf['Global\_active\_power'].fillna(cdf['Global\_active\_power'].median(),inplace=True)

cdf['Global\_reactive\_power'].fillna(cdf['Global\_reactive\_power'].median(),inplace=True)

cdf['Voltage'].fillna(cdf['Voltage'].median(),inplace=True)

cdf['Global\_intensity'].fillna(cdf['Global\_intensity'].median(),inplace=True)

**Data Transformation**

y=cdf['Global\_intensity']

X=cdf[['Global\_active\_power','Global\_reactive\_power','Voltage']]

**Pre Processed Data Splitting**

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

from sklearn.linear\_model import LinearRegression

X\_train, x\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size=0.30,random\_state=26)

X\_train, x\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size=0.30,random\_state=26)

**Modelling**

regr=LinearRegression()

regr.fit(X\_train,y\_train)

prediction=regr.predict(x\_test)

regr.score(X\_train,y\_train)

**Results**

from sklearn.metrics import mean\_squared\_error

from math import sqrt

rms=sqrt(mean\_squared\_error(y\_test,prediction))

rms

We compare the actual values and predicted values to calculate the accuracy of a regression model. Evaluation metrics provide a key role in the development of a model, as it provides insight to areas that require improvement.

**Full Dataset Classification Algorithms**

While Linear Regression is suited for estimating continuous values (e.g. estimating house price), it is not the best tool for predicting the class of an observed data point. In order to estimate the class of a data point, we need some sort of guidance on what would be the most probable **class** for that data point. For this, we use Logistic Regression.

import pandas as pd

df=pd.read\_csv("hpc\_breakdowns.csv",parse\_dates=['Date'],index\_col=['Date'])

bdf=df.iloc[0:-200000,:]

bdf

Here, we convert output column to binary classes – of global intensity greater than 12 & less than or equal to 12.

bdf.loc[bdf['Global\_intensity']<=12,'Global\_intensity']=0

bdf.loc[bdf['Global\_intensity']>12,'Global\_intensity']=1

**Data Analysis**

import seaborn as sns

sns.pairplot(bdf)

bdf.isnull().sum()

bdf.info()

bdf.corr()

import matplotlib.pyplot as plt

plt.figure(figsize=[15,5])

sns.heatmap(bdf.corr(),annot=True)

plt.show()

bdf.describe()

sns.boxplot(bdf['Voltage'])

sns.boxplot(bdf['Global\_active\_power'])

sns.boxplot(bdf['Global\_reactive\_power'])

**Data Cleaning**

bdf.drop(['Time','Sub\_metering\_1','Sub\_metering\_2','Sub\_metering\_3'],axis=1,inplace=True)

bdf.shape

bdf.isnull().sum()

bdf['Global\_active\_power'].fillna(bdf['Global\_active\_power'].median(),inplace=True)

bdf['Global\_reactive\_power'].fillna(bdf['Global\_reactive\_power'].median(),inplace=True)

bdf['Voltage'].fillna(bdf['Voltage'].median(),inplace=True)

bdf['Global\_intensity'].fillna(bdf['Global\_intensity'].median(),inplace=True)

bdf.isnull().sum()

bdf.shape

Data Transformation

ys=bdf['Global\_intensity']

Xs=bdf[['Global\_active\_power','Global\_reactive\_power','Voltage']]

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import RandomForestClassifier

from sklearn.svm import SVC

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import GridSearchCV

**PreProcessed Data Splitting**

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

X\_train, x\_test, y\_train, y\_test = train\_test\_split(Xs,ys,test\_size=0.30,random\_state=26)

X\_train=X\_train.sample(n=5000)

y\_train=y\_train.sample(n=5000)

For training a model we initially split the model into 3 three sections which are ‘Training data’ ,‘Validation data’ and ‘Testing data’.

You train the classifier using ‘training data set’, tune the parameters using ‘validation set’ and then test the performance of your classifier on unseen ‘test data set’. An important point to note is that during training the classifier only the training and/or validation set is available. The test data set must not be used during training the classifier. The test set will only be available during testing the classifier.

**Training set:** The training set is the material through which the computer learns how to process information. Machine learning uses algorithms to perform the training part. A set of data used for learning, that is to fit the parameters of the classifier.

**Validation set:** Cross-validation is primarily used in applied machine learning to estimate the skill of a machine learning model on unseen data. A set of unseen data is used from the training data to tune the parameters of a classifier.

**Test set:** A set of unseen data used only to assess the performance of a fully-specified classifier.

**Modelling**

ran = RandomForestClassifier(random\_state=1)

log = LogisticRegression()

svc = SVC(probability=True)

models = [ran,log,svc]

scores = []

for mod in models:

mod.fit(X\_train,y\_train)

acc = cross\_val\_score(mod,X\_train,y\_train,scoring = "accuracy", cv =5)

scores.append(acc.mean())

results = pd.DataFrame({

'Model': ['Random Forest','Logistic Regression','SVC'],

'Score': scores})

result\_df = results.sort\_values(by='Score', ascending=False).reset\_index(drop=True)

result\_df

**HyperParameter Tuning**

A hyperparameter is a parameter whose value is set before the learning process begins.

Grid Search is a method for optimizing hyperparameters.

n\_estimators = [10, 25, 50, 75, 100]

max\_depth = [3, None]

max\_features = [1, 3, 5, 7]

min\_samples\_split = [2, 4, 6, 8, 10]

min\_samples\_leaf = [2, 4, 6, 8, 10]

hyperparams = {'n\_estimators': n\_estimators, 'max\_depth': max\_depth,

'min\_samples\_split': min\_samples\_split, 'min\_samples\_leaf': min\_samples\_leaf}

gd=GridSearchCV(estimator = RandomForestClassifier(), param\_grid = hyperparams,

verbose=True,cv=5, scoring = "accuracy")

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_estimator\_)

The benefit of grid search is that it is guaranteed to find the optimal combination of parameters supplied. The drawback is that it can be very time consuming and computationally expensive.

Cs = [0.001, 0.01, 0.1, 1, 5, 10, 15, 20, 50, 100]

gammas = [0.001, 0.01, 0.1, 1]

hyperparams = {'C': Cs, 'gamma' : gammas}

gd=GridSearchCV(estimator = SVC(probability=True), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy")

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_estimator\_)

ran = RandomForestClassifier(n\_estimators=10,

max\_depth=3,

max\_features=3,

min\_samples\_leaf=2,

min\_samples\_split=8,

random\_state=1)

log = LogisticRegression(C=2.7825594022071245,

penalty='l2')

svc = SVC(probability=True)

models = [ran,log,svc]

scores\_v3 = []

for mod in models:

mod.fit(X\_train, y\_train)

acc = cross\_val\_score(mod, X\_train, y\_train, scoring = "accuracy", cv=5)

scores\_v3.append(acc.mean())

results = pd.DataFrame({

'Model': ['Random Forest','Logistic Regression','SVC',],

'Original Score': scores,'Score with tuned parameters': scores\_v3})

result\_df = results.sort\_values(by='Score with tuned parameters', ascending=False).reset\_index(drop=True)

result\_df

yprediction=log.predict(x\_test)

yactual=y\_test

**Results**

Accuracy of classifier can be measured using classification report & confusion matrix.

from sklearn.metrics import classification\_report

print(classification\_report(yprediction,yactual))

from sklearn.metrics import confusion\_matrix

print(confusion\_matrix(yactual,yprediction))

Once the model is trained we can use the same trained model to predict using the testing data i.e. the unseen data. Once this is done we can develop a confusion matrix, this tells us how well our model is trained. A confusion matrix has 4 parameters, which are ‘True positives’,‘TrueNegatives’, ‘False Positives’and ‘False Negative’. We prefer that we get more values in the True negatives and true positives to get a more accurate model. The size of the Confusion matrix completely depends upon the number of classes.

**Evaluation**

Model Evaluation is an integral part of the model development process. It helps to find the best model that represents our data and how well the chosen model will work in the future.

To improve the model we might tune the hyper-parameters of the model and try to improve the accuracy and also looking at the confusion matrix to try to increase the number of true positives and true negatives.