5/4/23

CEMENT STRENGTH PREDICTION

Outlines ;

(i) Problem Statement
(ii) Description of data
(iii) Application Architecture and model division
(iv) Code Explanation
→ Data Ingestion
→ Data Preprocessing
→ Model Selection
→ Model Tuning
→ Prediction
→ Logging Framework
→ Deployment (on-premise / cloud).

Problem Statement ;

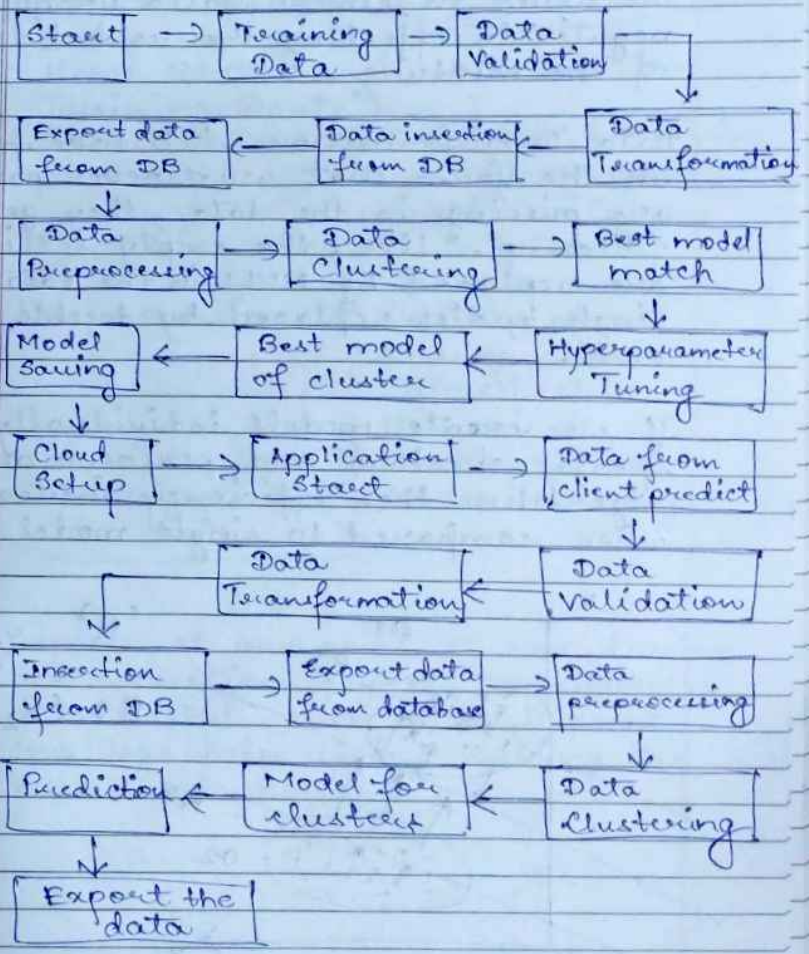
To build a regression model to predict the concrete compressive strength based on the different features in the training data.

Data Description ;

Given is the variable name, variable type, the measurement unit, and a brief description. The concrete compressive strength is the regression problem. The order of this listing corresponds to the order of numerals along the rows of the database.

Name	Data Type	Measurement	Descrip.
• Cement (Comp. 1)	quantitative	Kg in m^3	i/p var.
• Blast Furnace Slag	quantitative	Kg in m^3	i/p var. (coproduct in process)
• Fly Ash (comp. 3)	quantitative	Kg in m^3	i/p var. (coal product)
• Water	"	"	i/p var.
• Coarse Aggregate	"	"	i/p var. (large stones used)
• Fine Aggregate	"	"	i/p (fine stones in a cement mixture)
• Age	quantitative	Kg in m^3	i/p variable
• Concrete Compressive Strength	quantitative	MPa	o/p variable

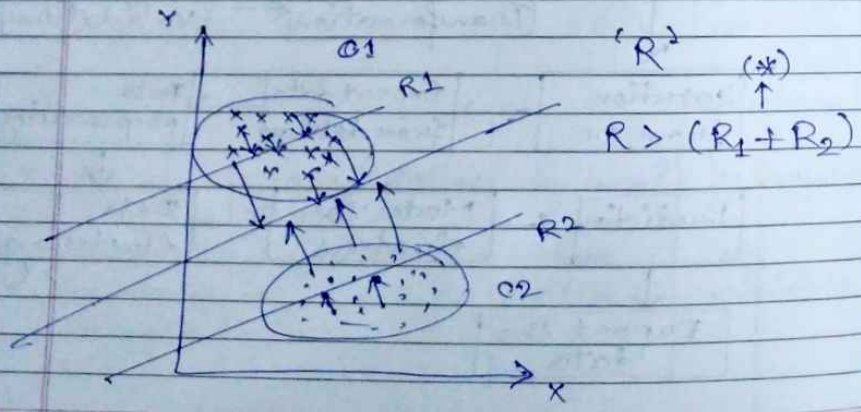
Apart from the training files, we also require a schema file from the client, which contains all the relevant information about training files such as:



- Superplasticizers → SP's, also known as high range water reducers, are additives used in making high strength concrete. Their addition to concrete or mortar allows the reduction of the water to cement ratio without negatively effecting the workability of the mixture.

Data Transformation will take place in the form that whatsoever parts are missing in the data, they get filled-up. Like, the empty sections are replaced by NULL in the database. Single quotes replaced by double ones.

If we create models individually for specific clusters and perform our algorithm then efficiency increases, when compared to single model.



Using hyperparameter tuning we will enhance our model to an extent in which the efficiency becomes higher.

Flow of Application;

(a) Home Route → Validation
(b) Training Route → Actual training [Preprocessing (Feature / Column Selection)]
(c) Prediction

Preprocessing → Feature of csv
→ Missing value
→ Categorical to Numerical
→ Normalization

Clustering and Model Training

* Suppose at any point of time during the execution process, the process gets halted, then at that point the logs help us in tracking the execution process.

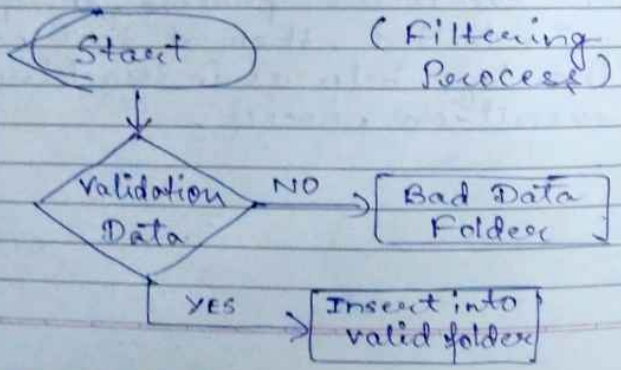
Logger File ; File / Database / Codepart

```
from datetime import datetime
class App_Logger:
    def __init__(self):
        pass

    def log(self, file_object, log_message):
        self.now = datetime.now()
        self.date = self.now.date()
        self.current_time =
            self.now.strftime("%H:%M:%S")
        file_object.write(str(self.date)+
        "/" + str(self.current_time) +
        "\t\t" + log_message + "\n")
```

* Training Validation Insertion :-

Whenever we are moving ahead with a project there are some conversations between the developers and clients on the requests like file type name, data type.
If anything outside these validation rule happens, then the format is rejected.



Code :

```
def trainRouteClient():
    try:
        if request.json['path'] is not None:
            ->path = request.json['path']
            ->train_valObj = train_validation
                                (path)
            ->train_valObj.train_validation()
            ->trainModelObj = trainModel()
            ->trainModelObj.trainingModel()
```

Individual functionalities are defined separately.

Schema Training JSON file ;

```
"ColName": {
"CementComp": "Float";
"Blast Furnace": "Float";
"Water Component": "Float",
        :
        :
"Aging Days": "INT",
};
```

* Hyperparameter allows us to enhance the model and helps us in finding out the best possible set of rules and regulations for a given set of data.

Flow of program ;

[Flask App]

(a) Home Route (web UI)
(b) Training
(c) Prediction

main.py app.py

After training of individual models we will perform the hypermodel clustering and tuning and find out best process for validation.

Usage of Frameworks ;

(a) Flask → Flask is a python framework in which we can render requests and templates to link the backend and frontend. It has both the capability of rendering the database and as well as script logic.

(b) Tensorflow → Tensorflow is basically associated with the training model part. It has the capability of having varying validations of data and helps in training the model based on their data.

Abstraction between processes

Different i/p nodes :-

(a) Simple Server → Whenever we use any training feature, we maintain two memory blocks. One which contains the data to be preprocessed called as server, and the model environment acting as the client.

(b) Prediction Validation Insertion →
→ Feature Selection
→ ~~Normalization~~ Normalization
→ Missing Values
→ Numerical ⇆ Categorical

(c) Training Model → This consists of the set of functionalities used for categorical training, i.e., training with the section of features.

* Database Operations :-

Now why do we require this operations. Consider a situation in which we need to provide csv data files into the backend of our AI machine. It is only possible if we cluster all the csv files and convert it into tables of the database.

That's why databases are created in order to have a complete understanding about the frontend

and backend linking.

Once the processing on the given sqlite database is done, then the complete tabular data is again converted into the excel file which is further shared with client or users.

Codebase Snippet :-

```
try:
conn = self.dataBaseConnection(DatabaseName)
c= conn.cursor()
c.execute("SELECT count(name) FROM
sqlite_master WHERE type='table'
and name="Good-Raw-Data"")
```

---

* $x(x+2)(x+4)(x+6)(x+8) = 2$

  $x^5 + 384 = 7$

* 107
  +44
  151
  +44
  195

$$\frac{x + x^2 + x^3 + x + (x+d) + (x+2d)}{6} = 66$$

$$\frac{x^3 + x^2 + x}{3} = 62$$

3 6 9 12 15 18
7 14
15 11 13 17 19

$100 + 90 + 60 = 250 - 120$

$= 130 - 60$

$= 70$

2 4 8 10 16 20

$\frac{7}{8} = \frac{x}{20}$

$x = \frac{35}{82} = 17.5$

CREDIT CARD

Data Description ;

The client will send data in multiple sets of files in batches at a given location. The data has been extracted from the census bureau.

The data contains 32561 instances with following attributes ;

Features ;

(a.) LIMIT_BAL : (Continuous) Credit limit of a person.
(b.) Sex : (Categorical) 1 = Male ; 2 = Female ; ~~3 = High School ;~~
~~4 = Others~~.
(c.) Marriage : 1 = Married ; 2 = Single ; 3 = Others.
(d.) Age : (Num) → Continuous
(e.) Education (Categorical) : 1 = Graduate School ; 2 = University ; 3 = High School ; 4 = Others.
(f.) Pay_0 to Pay_6 : History of past payments. We tracked the past monthly payment records (from April to September, 2005).
(g.) Bill_Amt1 to Bill_Amt6 : Amount of Bill Statements
(h.) Pay_Amt1 to Pay_Amt6 : Amount of previous payments

Target Label :

When a person shall default in the credit card payment or not.

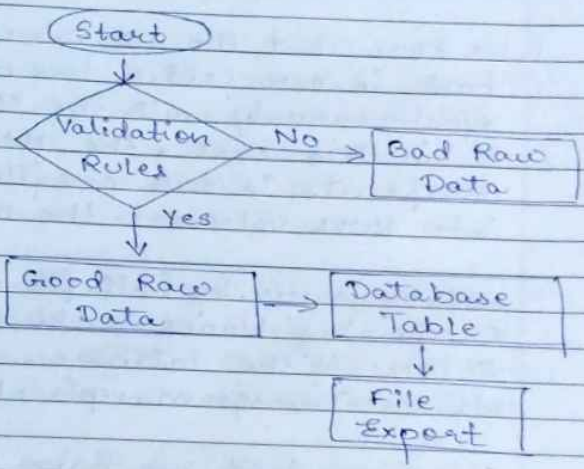
(a) Default payment next month :
Yes = 1 , No = 0 .

Apart from training files , we also require a "schema" file from the client , which contains all the relevant information about training files such as :

Name of files , Length of date value in filename , Length of Time value in filename , Number of Columns and their datatype .

* Code Validation :-

```
@ app.route ("/train" , methods = ["POST"])
@ cross_origin()
def trainRouteClient():
    try:
        if request.json['filepath'] is not None:
            path = request.json['filepath']
            train_valObj = train_validation(path)
            train_valObj = train_validation()
            # calling training validation functions
            trainModelObj = trainModel()
            trainModelObj.trainingModel()
            # training the model for files
    except ValueError:
        return Response("Error Occured "%
                        ValueError)
```




```
for filenames in onlyfiles:
    if (re.match(regex, filename)):
        splitAtDot = re.split('.csv', filename)
        splitAtDot = (re.split('_', splitAtDot[0]))
        if (len(splitAtDot[1] == lengthofdate)):
            if (len(splitAtDot[2]==lengthoftime)):
                shutil.copy("Training"+filename)

            else
                shutil.copy("Training"+filename)
                self.logger.log("Invalid filename")

        else
            shutil.copy("Training"+filename)
            self.logger.log("Invalid filename")

# Validation for filename
```

* Data Transformation after the Validation :-

We know that the NA values which we have in our folder does not renders synchronously with the SQL language. So in this case, the N.A values of datapoints in our csv file is converted into NULL values in the SQL.

→ self.log_writer.log(self.file_object, "starting")
# replacing blanks in csv with the
# NULL in the database.
self.dataTransform.replaceMissingWithNull()

Since we don't ~~we~~ have strings so it isn't needed to be converted into double quotes.

* Database Operations :-

(A) for key in column_names.keys():
    type = column_names[key]
    try:
        conn.execute('ALTER TABLE
            Good_Raw_Data ADD COLUMN')
    except:
        conn.execute('CREATE TABLE
            Good_Raw_Data, format)

This is the creation of table initially with single column and then altering it by adding other columns as well.

(B) for line in enumerate(reader):
    for list_ in (line[1]):
        try:
            conn.execute('INSERT INTO Good_
            Raw_Data values ({values})'.
            format(values=(list_)))
            self.logger.log(log_file, "file")
            conn.commit()
        except Exception as e:
            raise e

Simply adding the values present in the rows of the csv dataset to the database in a row-wise format.

(C) Delete existing Good Data Training Folder and Bad Data Training folders because they are no longer required.

(D) Finally whatever data is present inside the table. It is exported by further converting into csv format.

self.dBOperation.selectingDatafromtable
intocsv('Training')
self.file_object.close()

* Data Preprocessing :-

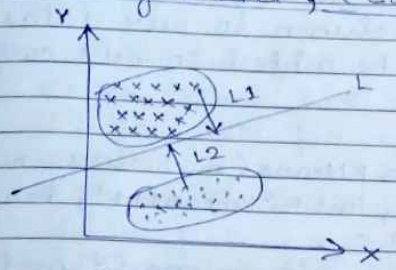
(I) Credit Card EDA

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
% matplotlib inline

data = pd.read_excel ('folder path')
data.head()
data.shape()
data.isna().sum()
data.info()
```

There are certain conditions in which it can be seen that there are some relationships between feature columns, they are not entirely independent.
But in this scenario, there is a co-relation because a customer who was not able to pay the bill for 1 month was again not able to pay it for subsequent months and hence the corelation.
Again for bill payment, the same has happened. If the customer was not able to pay then the bill amount got reduced.
The columns are removed when it tends to convey the same information.

(*) Training Model ; (Clustering)



Hence applying training models to individual clusters increases the level of optimization in the code.

For finding value of 'k' in the k-means clustering, we use elbow method which can be imported from kneed libraray.

```
for i in range (1, 11):
    kmeans = KMeans (n_clusters = i, init =
    'kmeans++', random_state = 42)
    kmeans.fit (data)
    wcss.append (kmeans.inertia_)
plt.plot (range (1, 11), wcss)
plt.title ('The Elbow Method')
plt.xlabel ('Clusters')
plt.ylabel ('wcss')
plt.savefig ('preprocessing data / k-means elbow.png')
self.kn = KneeLocator (range (1, 11)), wcss,
curve = 'convex', direction = 'decreasing')
self.loggere_object.log (self.file_object,
return self.kn.knee
```

Now we need to add a cluster definition column to our dataset which will be added to our csv file.

```
try:
    self.kmeans = KMeans(n_clusters = number_of_clusters, init = 'kmeans++', random_state = 42)
    self.y_means = self.kmeans.fit_predict(data)

    self.file_op = file_methods.File_Operation(self.file_object, self.logger_object)
    self.save_model = self.file_op.save_model(self.kmeans, 'K-Means')

    self.data['Cluster'] = self.y_means
    self.logger_object.log(self.file_object)
    return(self.data)
```

New Feature

x_1	x_2	x_3	x_4	Cluster
				0
				1

0 → Group 0
1 → Group 1
2 → Group 2

} → Hyperparameter Tuning

* Model Selection and Tuning :-

```
for i in list_of_clusters:
    cluster_data = x[x['Cluster']==i]
    # filter data for one cluster

    # Prepare feature and label columns
    cluster_feature = cluster_data.drop
    (['Labels','Cluster'], axis=1)
    cluster_label = cluster_data['Labels']

    # Splitting data into training and test set
    x_train, x_test, y_train, y_test =
    train_test_split(cluster_features,
      cluster_data)
    # data pre-processing steps
    train_x = preprocessor.scale_numerical
    _columns(x_train)
    test_x = preprocessor.scale_numerical
    _columns(x_test)
    model_finder = tuner.Model_Finder
    (self.file_object, self.log_writer)

    # Getting best model for clusters
    best_model_name, best_model = model_
    finder.get_best_model(train_x, y_train,
    test_x, y_test)

    # saving best model
    file_op = file_methods.File_Operation
    (self.file_object, self.log_writer)
    save_model = file_op.save_model(best_model,
      best_model_name + str(i))
```

* Predictions :-

Read Client Data → Data Validation
↓
CSV ← DB Operations
↓
Data Preprocessing → Clustering
↓
Prediction ← Load Clusters Model
↙ ↘
GUI Postman

Home Page Route ;

```
@ app.route ("/", methods = ['GET'])
@ cross_origin ()
def home ():
    return render_template ('index.html')
```

Prediction Route ;

```
@ app.route ("/predict", methods = ['POST'])
@ cross_origin ()
```

These two will be called in our homepage route.
It is inlined in index.html file.

* Deployment :-

Files to be added ;

(a) requirements.txt → It will be created in the terminal by command [pip freeze > requirements.txt].

(b) procfile → web: gunicorn main:app

Using ~~Red~~ Render deployment :-

(a) Create app-name.
(b) Follow steps for deployment to cloud.