**ASSIGNMENT**

The dataset can be found [here](https://drive.google.com/file/d/16ks3fowx7oLJNCZjf4dLsKOy4GyhKVdi/view?usp=sharing).

The objective is to predict the revenue of shops. Each row of our data contains the following information:

* shop\_ID : Shop's unique identifier.
* day\_of\_the\_week : Encoded from **0** to **6**.
* date : day, month and year of the data point.
* number of customers : Quantity of customers that showed up that day.
* open : Binary variable equal to **0** if shop closed that day and **1** if shop open.
* promotion : Binary variable equal to **0** if shop had no promotions that day and **1** if it did.
* state\_holiday : Encoded **0**, **a**, **b**, **c** indicating if there was a state holiday at all (**0** if not), and otherwise, the number indicates which state holiday it was.
* school\_holiday : Binary variable equal to **1** if there was a school holiday that day and **0** if not.

At 16:00 a validation file will be uploaded here (see [here](https://drive.google.com/file/d/18q20AFPPIs5ABFHqsyOOyrx7vqqsyfzz/view?usp=sharing) for template - the validation file will be much larger). Your submission is due at 16:30 and should consist of an output file with just the index and sales prediction for the row (see [here](https://drive.google.com/file/d/16pfvXy1D-m9V4IZ6iLBFLidHbIDRGUO7/view?usp=sharing) for template), as well as a pickled version of your model, a requirements.txt file and any feature engineering steps you may have used (preferably pickled as well).

We must be able to run your model and generate predictions in our computers, so we strongly advise you to try to emulate in a clean environment what we would do and see if you are delivering all necessary artifacts.

The scoring metric will be R2.

**FRAMEWORK**

### **Step 1: Setting up the environment - working on google collab**

* Import **libraries**: Pandas, NumPy, Scikit-learn..
  + import pandas as pd
  + import numpy as np
  + from sklearn.model\_selection import train\_test\_split
  + from sklearn.preprocessing import OneHotEncoder
  + from sklearn.ensemble import RandomForestRegressor
  + from sklearn.metrics import r2\_score import pickle
  + from sklearn.neighbors import KNeighborsRegressor
* Load the **dataset**: as always, use copy path file
  + data = pd.read\_csv("/content/IH - Kaggle project -sales - sales.csv")

### **Step 2: Data preprocessing and feature engineering**

* Handle **missing values**: check for missing data
  + missing\_values = data.isnull().sum(). How do we handle them?
    - **Numerical Columns**: Using the median value is especially useful when dealing with outliers. The median is less sensitive to extreme values compared to the mean, making it a suitable choice to impute missing values in numerical data.
    - **Categorical/Binary Columns**: It helps maintain the distribution of categories within the column and doesn't introduce bias toward a specific category.
      * remaining\_missing = data.isnull().sum()
* **Encode categorical variables**: very important step where I need to convert categorical variables like state\_holiday into numerical form using techniques such as one-hot encoding.
* **Feature engineering**: here we create additional features if needed based on insights gained from the dataset.

### **Step 3: Data splitting**

* **Split Data into Training and Testing Set**s: using train\_test\_split from Scikit-learn

### **Step 4: Model building**

* **Algorithm choice**: e.g., Random Forest Regressor, Gradient Boosting Regressor, etc. to predict sales.
* **Model training**: to fit the selected model using the training data.

### **Step 5: Model evaluation and saving**

* **Predictions**: we generate predictions on the test set and evaluate model performance using R2 score as requested
* **Last 2 classes - review** 
  + **Save Model**: pickle the trained model and save it for future use.
  + **Create Outputs**: Prepare the required output files, including the index and sales prediction for each row.

### **Step 6: Generate necessary files - request of the exercise**

* **Requirements.txt:** need to check what needs to be here exactly
* **Feature engineering steps**: Save any feature engineering steps as a separate file, preferably pickled. I will already do it during the model development right?

### **Step 1: Setting up the environment - working on google collab**

import pandas as pd

import numpy as np

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

from sklearn.preprocessing import OneHotEncoder

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import r2\_score

import pickle

data = pd.read\_csv("/content/IH - Kaggle projeect -sales - sales.csv")

### **Step 2: Data preprocessing and feature engineering**

# Check for missing values

missing\_values = data.isnull().sum()

missing\_values

#### **Numerical Columns:** Using the median value is specially useful when dealing with outliers. The median is less sensitive to extreme values compared to the mean, making it a suitable choice to impute missing values in numerical data.

#### **Categorical/Binary Columns**: It helps maintain the distribution of categories within the column and doesn't introduce bias toward a specific category.

# Handling missing values for numerical columns

numerical\_cols = ['nb\_customers\_on\_day', 'sales'] # here for numerical values

for col in numerical\_cols:

median\_value = data[col].median()

data[col].fillna(median\_value, inplace=True)

# Handling missing values for categorical/binary columns

categorical\_cols = ['open', 'promotion', 'state\_holiday', 'school\_holiday'] # here for categorical values

for col in categorical\_cols:

mode\_value = data[col].mode()[0]

data[col].fillna(mode\_value, inplace=True)

remaining\_missing = data.isnull().sum()

remaining\_missing

# Encode categorical variables

#### **Purpose:**

* Handling Categorical Data: Categorical variables, like 'state\_holiday', can't be directly used in most machine learning models as they expect numerical input. Encoding these categorical variables converts them into a format that the model can understand.

#### **Process - One-Hot Encoding:**

* pd.get\_dummies(): This function from pandas is used to convert categorical variables into a numerical format using a technique called one-hot encoding.
* columns=categorical\_cols: This specifies which columns to encode. In this case, it's 'state\_holiday', but you can add more columns to this list if you have other categorical columns in your dataset.

# Encode categorical variables

categorical\_cols = ['state\_holiday'] # we may add any other categorical columns if needed

data\_encoded = pd.get\_dummies(data, columns=categorical\_cols)

# Perform feature engineering steps

**1. Convert date column to date time format (YYYY-MM-DD)**

I need to convert the 'date' column from a string into a datetime format.

**2. Create 'date\_numeric' from the 'date' column**

I need to convert the 'date' column from a string into a datetime format.

# Perform feature engineering steps

# Convert 'date' column to datetime format

data\_encoded['date'] = pd.to\_datetime(data\_encoded['date'])

data\_encoded['date\_numeric'] = data\_encoded['date'].astype(np.int64) // 10\*\*9

**3. Extracting features from the 'date' column:**

I need to extract specific date-related features (month, day, year, weekday) from the 'date' column to provide additional information that might be valuable for the machine learning model. **Example**: month, day, and year can capture seasonal trends, day-of-week might capture weekly patterns,... enhancing the dataset with more granular time-related information that could improve the model's predictive power when using time-based patterns

# Extracting features from the date column

data\_encoded['month'] = data\_encoded['date'].dt.month

data\_encoded['day'] = data\_encoded['date'].dt.day

data\_encoded['year'] = data\_encoded['date'].dt.year

data\_encoded['weekday'] = data\_encoded['date'].dt.weekday

**4. What are we doing in these next steps?**

We are creating a new set of features to capture the interaction between 2 variables. In the code below, we are capturing the interaction between the number of customers on a day and whether the shop was open. **Even if it may seem obvious, we want to set the base - ask Sabina**

# interaction between 'nb\_customers\_on\_day' and 'open'

data\_encoded['customers\_open\_interaction'] = data\_encoded['nb\_customers\_on\_day'] \* data\_encoded['open']

**5. Mean sales per store ID**

We are computing the **average sales for each store based on their 'store\_ID**' while creating a new feature **('mean\_sales\_per\_store')** to capture this information for each record in the dataset. **We are actually assigning to each record** the average sales of the respective store, essentially providing a measure of the typical sales performance for each store.

# Mean sales per store ID

mean\_sales\_per\_store = data\_encoded.groupby('store\_ID')['sales'].mean()

data\_encoded['mean\_sales\_per\_store'] = data\_encoded['store\_ID'].map(mean\_sales\_per\_store)

**5. Total sales per day of the week:**

* **Groupby** → Groups the data by 'day\_of\_week' and calculates the sum of sales for each day of the week
* **Data\_encoded** → Maps the total sales values to corresponding 'day\_of\_week' entries in a new column named 'total\_sales\_per\_day'.

# Total sales per day of the week

total\_sales\_per\_day = data\_encoded.groupby('day\_of\_week')['sales'].sum()

data\_encoded['total\_sales\_per\_day'] = data\_encoded['day\_of\_week'].map(total\_sales\_per\_day)

**6. Why are we applying this transformation?**

**Skewed distributions** in data can negatively impact the performance of machine learning models. We want to normalize skewed distributions, making the data more symmetrical and better fitting the assumptions of certain models.

**Handling Zero Values**: The addition of 1 to the values ('np.log1p') helps handle cases where the original data contains zero values **- I use it as a security measure even though I cleaned it. Does it make sense? Check with Sabina**. Logarithm of zero is undefined, so adding 1 prevents this issue and effectively transforms zero values.

# Log transformation of 'nb\_customers\_on\_day' if it follows a skewed distribution

data\_encoded['log\_customers'] = np.log1p(data\_encoded['nb\_customers\_on\_day'])

**7. Now we need to normalize and check for performance improvements**:

* From…. → we use this to import minmax scaler → technique for scaling numerical features
* Data…→ important to remember that it applies the MinMaxScaler transformation to 'nb\_customers\_on\_day' column and replaces the original values with the scaled values.

**But why are we doing it?**

* **Normalization:** we need to scale the values of the 'nb\_customers\_on\_day' column to a fixed range (0 to 1). It transforms the data in a way that the minimum value becomes 0, the maximum becomes 1, and all other values are proportionally adjusted in between.
* **Model Performance Improvement**: It ensures that all features contribute equally to the analysis and prevents certain features from dominating based on their scale.

# Feature scaling or normalization if necessary

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

data\_encoded['nb\_customers\_on\_day'] = scaler.fit\_transform(data\_encoded[['nb\_customers\_on\_day']])

In here we are removing the date column as it was processed into a new one. No longer required. **Is that correct? Check with Sabina**

data\_encoded.drop(columns=['date'], inplace=True)

### 

### 

### 

### 

### 

### **Step 3: Data splitting**

**1. We need to first separate features and target variable**

* **X:** we assign the **features** (which we need to remember are **independent variables**) to 'X' by **excluding the 'sales'** column. 'X' contains all columns except the **target variable.**
* **y:** we assign the **target variable** (dependent variable to predict) to 'y' → **sales** column

**2. Splitting into training and test sets:** This splitting process creates two sets of data:

* one for training the machine learning model ('X\_train' and 'y\_train')
* and the other for evaluating its performance ('X\_test' and 'y\_test').

**We want to avoid overfitting** and the separation between training and test sets helps in assessing how well the model will perform unseen data (which is the testing set) after being trained on the training set.

And let’s remember that (mainly for my to repeat over and over again what it means)

* *X\_train, X\_test, y\_train, y\_test*: These variables **store** the resulting splits of **features and target variable for both training and testing sets.**
* X, y: passed into functions where
  + 'test\_size=0.2' specifies that 20% of the data will be allocated for testing
  + and the remaining 80% will be used for training.
* Random state = 0 → to ensure we will obtain the same random split every time we run the code. Following the same flow as during class

# Step 4: Data Splitting

X = data\_encoded.drop(columns=['sales']) # Features

y = data\_encoded['sales'] # Target variable

# Splitting the dataset into 80% training and 20% testing

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=0)

### 

### 

### 

### 

### 

### 

### **Step 4: Model building**

* **Algorithm choice:** effective for classifications and regression models
  + N-estimators: We are specifying the number of trees in the random forest as 50 → which means the model will consist of 50 decision trees
  + Max depth: 10 levels maximum
* **Training the model (repeating for myself)**
  + X\_train: The features (independent variables) from the training set.
  + y\_train: The corresponding target variable (dependent variable) from the training set.

# Step 5: Model Building

model = RandomForestRegressor(n\_estimators=50, max\_depth=10)

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

**Why did I choose GBR?** it builds an ensemble of decision trees sequentially, where each new tree tries to correct the errors made by the previous ones.

* It combines multiple weak learners (trees) to create a strong predictive model.

from sklearn.ensemble import GradientBoostingRegressor

model = GradientBoostingRegressor(n\_estimators=100, max\_depth=5) # we can adjust parameters as needed depending on how long it may take to load

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

### 

### 

### 

### 

### 

### 

### 

### 

### 

### **Step 5: Model evaluation and saving**

We now use the model’s predict function to generate the predicted sales values (y\_pred) based on the features (X\_test) that the model has not seen during training. In my model, the **R2 score** indicates that, approximately, **93.7% of the variability in sales can be explained by the features included in your model.**

# Step 6: Model Evaluation and Saving

y\_pred = model.predict(X\_test)

r2 = r2\_score(y\_test, y\_pred)

print(f"R2 Score: {r2}")

**De-scaling it - Thanks Sabina/Isi**

# What was the range used in scaling?

min\_sales = 0

max\_sales = 10000

# saving the prediction

scaled\_predictions = y\_pred.copy()

# now inverting it

descaled\_predictions = scaler.inverse\_transform((scaled\_predictions.reshape(-1, 1) - min\_sales) / (max\_sales - min\_sales))

# updating the sales prediction in my data set

sales\_predictions = descaled\_predictions.flatten()

# Save the model

with open('sales\_prediction\_model.pkl', 'wb') as model\_file:

pickle.dump(model, model\_file)

# Create output file with index and sales prediction

output = pd.DataFrame({'index': X\_test.index, 'sales\_prediction': y\_pred})

output.to\_csv('sales\_predictions.csv', index=False)