□ Machine Learning Assignment 1 - Predicting Car Prices! □

This Jupyter notebook is a template for solving the assignment problem, i.e., Chaky company makes some car but he has difficulty setting the price for the car. □. Here, I will try to apply the skills I've learned over the past lectures. This notebook contains the following structure:

- □ **1. Setup**: Import block with all necessary imports (also provide some blocks with connection to drive, kaggle, and etc. for future use)
- □ **2. Loading the Data**: Loading, EDA, data cleaning, feature selection, and preprocess the dataset.
- □ **3. Models**: Starter code for basic models to kickstart your experimentation.
- □ **4. Evaluation Metrics**: Tools to evaluate your models using various metrics.
- □ **5. Inference and Conclusion**: Testing the best model and generating Report.

Let's start! □

Some notes:

The typical workflow of data science project is following:

```
1. Problem Definition
- Objective: Clearly define the problem you're trying to solve.
Understand the business or research goals and translate them into a
data science problem.
Tasks:
- - Identify the key objectives and success metrics.
- - Understand the constraints and resources available.
- - Formulate hypotheses or research questions.
2. Data Collection
- Objective: Gather the necessary data from various sources, which
could be internal databases, APIs, web scraping, or external datasets.
Tasks:
- - Identify data sources and acquire the data.
- - Integrate data from multiple sources if needed.
- - Ensure data privacy and compliance with regulations (e.g., GDPR).
3. Data Exploration and Analysis (Exploratory Data Analysis - EDA)
- Objective: Understand the data, its patterns, and any potential
issues through visualization and basic statistical analysis.
Tasks:
- - Summarize the data using descriptive statistics.
- - Visualize distributions, correlations, and trends.
- - Identify patterns, outliers, and potential relationships between
features.
- - Formulate additional hypotheses based on the data.
```

4. Data Preprocessing
- Objective: Clean and prepare the data for modeling.
Tasks:
- - Handle missing values (imputation or removal).
- - Handle outliers.
- - Encode categorical variables.
- - Normalize or standardize numerical features.
- - Split the data into training, validation, and test sets.
5. Feature Engineering
- Objective: Create new features or modify existing ones to improve model performance.
Tasks:
- - Create new features from existing data (e.g., interaction terms, polynomial features).
- - Apply feature scaling (normalization or standardization).
- - Transform features to handle skewness (e.g., log transformations).
- - Reduce dimensionality if necessary (e.g., PCA).
6. Model Selection
- Objective: Choose the appropriate machine learning models for the problem.
Tasks:
- - Compare different algorithms (e.g., linear models, decision trees, ensemble methods, neural networks).
- - Consider baseline models for comparison.
- - Choose models based on the problem type (e.g., classification, regression).
7. Model Training
- Objective: Train the chosen models on the preprocessed data.
Tasks:
- - Train the models using the training dataset.
- - Perform hyperparameter tuning (e.g., using grid search or random search).
- - Use cross-validation to evaluate model performance.
8. Model Evaluation
- Objective: Assess the model's performance using relevant metrics and ensure it meets the project goals.
Tasks:
- - Evaluate model performance on the validation dataset.
- - Use appropriate metrics (e.g., accuracy, precision, recall, F1-score, RMSE).
- - Analyze model errors and refine the model if necessary.
9. Model Deployment
- Objective: Integrate the model into a production environment where it can be used to make predictions.
Tasks:
- - Deploy the model as a service (e.g., REST API, microservice).
- - Ensure scalability and monitor the model's performance in production.
- - Handle model retraining as needed (e.g., with new data).

```
10. Monitoring and Maintenance
- Objective: Continuously monitor the model’s performance and maintain
its accuracy over time.
Tasks:
- - Track model performance using key metrics.
- - Monitor for data drift and update the model if necessary.
- - Address any issues in production and ensure model reliability.
11. Documentation and Reporting
- Objective: Document the entire process and communicate the results
to stakeholders.
Tasks:
- - Prepare detailed reports and visualizations.
- - Document the data, model, and processes.
- - Share insights and actionable recommendations with stakeholders.
12. Iteration and Optimization
- Objective: Refine the project by iterating over the steps to improve
results.
Tasks:
- - Revisit earlier steps based on feedback and new insights.
- - Optimize the model and the workflow for better performance.
```

1. Setup

The following libraries are required to run this notebook. If you are running this on Colab it should be all smooth sailing. If you are running it locally please make sure you have all of these installed.

```
# Import section, basically importing everything what I need later +
default imports
import os
import random
import zipfile
from collections import defaultdict

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

import mlflow

# For sklearn imports I will import them in model sections for better
explanation purposes
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
```

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import GradientBoostingRegressor

from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import ShuffleSplit, KFold

from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
```

2.1 DataLoading

First thing we need to do is load in the data. We will be looking at the cars dataset (shared for this assignment cars). This dataset is tabular and contains information regarding car details(year, brand, mileage, and etc.) and we need to predict the price of the car(regression).

```
# Loading the data
train_csv_path = 'cars.csv'
df = pd.read_csv(train_csv_path)
```

Data Preprocessing and Label Encoding

We need to represent categorical data into numerical form via encoding. This step should be done before EDA

```
# Let's observe what are columns and their data types
df.dtypes

name              object
year               int64
selling_price      int64
km_driven          int64
fuel              object
seller_type       object
transmission      object
owner             object
mileage           object
engine            object
max_power         object
torque            object
seats            float64
dtype: object
```

The task is following (note: I will deal with nan valuese on the fly alongs tasks, I am thinking since its regression task our predictions is approximate (it is okay to change with mean/median based on distribution), but for classification tasks I think it is better to drop such rows. Therefore, I will keep them.):

```
1. Feature owner - map First owner to 1, ..., Test drive car to 5
2. Feature fuel - remove all rows with CNG and LPG because CNG and LPG
use a different mileage system (km/kg) which is different from kmpl
for Diesel and Petrol
3. Feature mileage - remove "kmpl" and convert to float
4. Feature engine - remove "CC" and convert to numerical
5. Feature max power - same as engine
6. Feature brand - take first word and remove other
7. Drop feature torque
8. Test Drive cars are expensive, so delete all samples
```

```python
# task 1 - Feature owner - map First owner to 1, ..., Test drive car
to 5
df_copy = df.copy()

# First Owner   5289
# Second Owner  2105
# Third Owner   555
# Fourth & Above Owner     174
# Test Drive Car 5

# Better to use one-hot encoding, but as per hw instructions doing
mapping.
owner_map = {
    'owner': {
        "First Owner": 1,
        "Second Owner": 2,
        "Third Owner": 3,
        "Fourth & Above Owner": 4,
        "Test Drive Car": 5,
    }
}

df_copy.replace(owner_map, inplace=True)

# task8 - Test Drive cars are expensive, so delete all samples
df_copy = df_copy[df_copy.owner != 5]
print(df_copy.owner.value_counts())

# doing in such a sandwich way for testing purposes on the fly
df = df_copy.copy()
```

```
owner
1    5289
2    2105
3     555
4     174
Name: count, dtype: int64
```

```
C:\Users\eraco\AppData\Local\Temp\ipykernel_31172\1370062162.py:21:
FutureWarning: Downcasting behavior in `replace` is deprecated and
will be removed in a future version. To retain the old behavior,
explicitly call `result.infer_objects(copy=False)`. To opt-in to the
future behavior, set `pd.set_option('future.no_silent_downcasting',
True)`
  df_copy.replace(owner_map, inplace=True)
```

```python
# Encoder for binary categorical values
from sklearn.preprocessing import LabelEncoder

# defining encoder
le = LabelEncoder()

# task2 Feature fuel - remove all rows with CNG and LPG
df_copy = df.copy()

# df_copy['fuel'].value_counts()
# CNG 57
# LPG 38

print(df_copy.shape)
df_copy = df_copy[~df_copy.fuel.isin(['CNG', 'LPG'])]

print(df_copy.shape)
df_copy.fuel.value_counts()

# And also let's encode it
df_copy.fuel = le.fit_transform(df_copy.fuel)
print(df_copy.fuel.value_counts())


df = df_copy.copy()
```

```
(8123, 13)
(8028, 13)
fuel
0    4401
1    3627
Name: count, dtype: int64
```

```python
# task6 - Feature brand - take first word and remove other
# same approach

df_copy = df.copy()

# Changing name to brand
df_copy.rename(columns = {'name': 'brand'}, inplace=True)
df_copy.brand = df_copy.brand.str.split().str[0]

print(df_copy.brand.isna().sum())
```

```
# Doing mapping

# Bad choice, I will proceed with one-hot encoding (though too much
values)
# brand_name_map = {'brand': {v:k for k, v in zip(range(1, 33),
#          ['Maruti', 'Skoda', 'Honda', 'Hyundai', 'Toyota', 'Ford',
'Renault',
#         'Mahindra', 'Tata', 'Chevrolet', 'Fiat', 'Datsun', 'Jeep',
#         'Mercedes-Benz', 'Mitsubishi', 'Audi', 'Volkswagen', 'BMW',
#         'Nissan', 'Lexus', 'Jaguar', 'Land', 'MG', 'Volvo', 'Daewoo',
#         'Kia', 'Force', 'Ambassador', 'Ashok', 'Isuzu', 'Opel',
'Peugeot'])
#     }
# }

# I will proceed with groupped one-hot encoding
group_map = {
    'Economy': ['Maruti', 'Tata', 'Hyundai', 'Datsun', 'Renault',
'Ford', 'Chevrolet', 'Fiat'],
    'Midrange': ['Honda', 'Toyota', 'Mahindra', 'Nissan', 'Skoda',
'Mitsubishi', 'Kia', 'MG'],
    'Luxury': ['Audi', 'BMW', 'Mercedes-Benz', 'Volvo', 'Jaguar',
'Lexus', 'Jeep', 'Land'],
    'Others': ['Daewoo', 'Ambassador', 'Ashok', 'Isuzu', 'Opel',
'Peugeot', 'Force']
}

# local mapper - later maybe need to define this in backend code
brand_to_group = {brand: group for group, brands in group_map.items()
for brand in brands}

# mapping cars to its groups
df_copy.brand = df_copy.brand.map(brand_to_group)

# creating columns of brand groupping
df_encoded = pd.get_dummies(df_copy, columns=['brand'],
drop_first=True)

df_encoded.head()

df = df_encoded.copy()

0

# Transmission feature has 2 classes only, so use LabelEncoder
df_copy = df.copy()

df_copy.transmission = le.fit_transform(df_copy.transmission)
print(df_copy.transmission.value_counts())
```

```
df = df_copy.copy()
```

```
transmission
1    6982
0    1046
Name: count, dtype: int64
```

```
# seller_type feature has 3 classes: individual,  dealer, trustmark
dealer -> use one-hot encoding
df_copy = df.copy()

# one-hot encoding, drop_first=True to drop one not required column
df_copy = pd.get_dummies(df_copy, columns=['seller_type'],
drop_first=True)
df_copy.head()

df = df_copy.copy()

# task3 - Feature mileage - remove "kmpl" and convert to float
# Hint: use df_copy.mileage.str.split

df.mileage = df.mileage.str.split().str[0].astype(float)

# task4 - Feature engine - remove "CC" and convert to numerical
# Same as task3

df.engine = df.engine.str.split().str[0].astype(float)

# task5 - Feature max power - same as engine

df.max_power = df.max_power.str.split().str[0].astype(float)

# task7 - dropping torque column
# so that it would not have impact on EDA - even though its bad
practice

df.drop(columns=['torque'], inplace=True)

# Checking if everything is fine
# But probably, it would be better to keep torque and transfer for
numerical form for the EDA basis
# I will test it in next iteration
df.dtypes
```

```
year                                     int64
selling_price                            int64
km_driven                                int64
fuel                                     int32
transmission                             int32
owner                                    int64
mileage                                float64
```

```
engine                                float64
max_power                             float64
seats                                 float64
brand_Luxury                             bool
brand_Midrange                           bool
brand_Others                             bool
seller_type_Individual                   bool
seller_type_Trustmark Dealer             bool
dtype: object
```

Now we can proceed with EDA

2.2 Exploratory Data Analysis (EDA)

DataFrame columns:

```
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   name           8128 non-null   object 
 1   year           8128 non-null   int64  
 2   selling_price  8128 non-null   int64  
 3   km_driven      8128 non-null   int64  
 4   fuel           8128 non-null   object 
 5   seller_type    8128 non-null   object 
 6   transmission   8128 non-null   object 
 7   owner          8128 non-null   object 
 8   mileage        7907 non-null   object 
 9   engine         7907 non-null   object 
 10  max_power      7913 non-null   object 
 11  torque         7906 non-null   object 
 12  seats          7907 non-null   float64
```

General Notes about EDA:

```
value_counts(): Frequency counts
outliers: the value that is considerably higher or lower from rest of
the data

Value at 75% is Q3 and value at 25% is Q1 -> Q stands for "quartile"
Outlier are smaller than Q1 - 1.5(Q3-Q1) and bigger than Q3 + 1.5(Q3-
Q1). (Q3-Q1) = IQR
IQR stands for "interquartile range"

We will use describe() method. Describe method includes:
count: number of entries
mean: average of entries
std: standart deviation
```

```
min: minimum entry
25%: first quantile
50%: median or second quantile
75%: third quantile
max: maximum entry
```

```
# Let's see all columns
df.columns
```

```
Index(['year', 'selling_price', 'km_driven', 'fuel', 'transmission', 'owner',
       'mileage', 'engine', 'max_power', 'seats', 'brand_Luxury',
       'brand_Midrange', 'brand_Others', 'seller_type_Individual',
       'seller_type_Trustmark Dealer'],
      dtype='object')
```

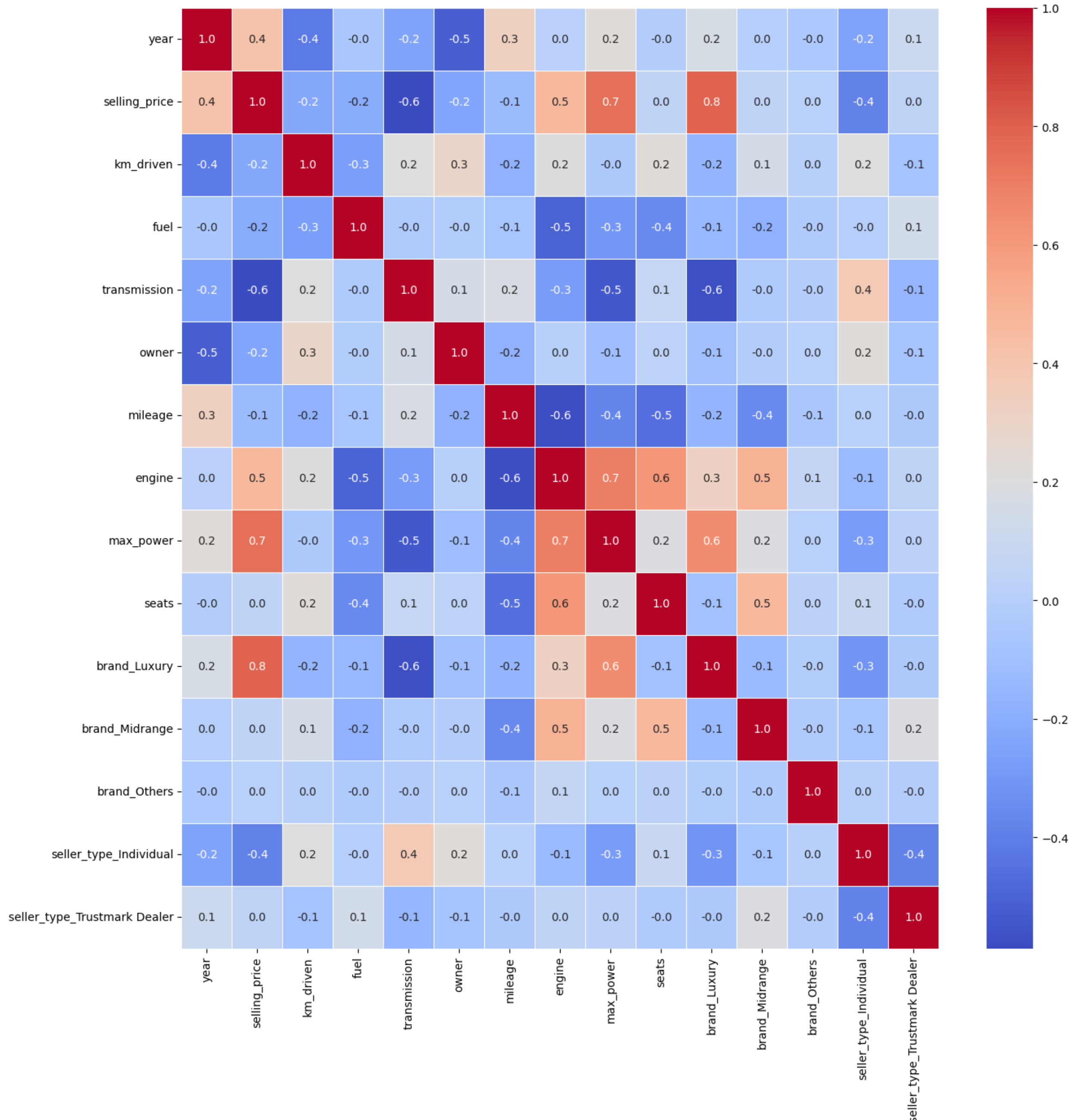
```
# Some basic info about each column
# We see there are null values
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8028 entries, 0 to 8127
Data columns (total 15 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   year                          8028 non-null   int64
 1   selling_price                 8028 non-null   int64
 2   km_driven                     8028 non-null   int64
 3   fuel                          8028 non-null   int32
 4   transmission                  8028 non-null   int32
 5   owner                         8028 non-null   int64
 6   mileage                       7814 non-null   float64
 7   engine                        7814 non-null   float64
 8   max_power                     7820 non-null   float64
 9   seats                         7814 non-null   float64
 10  brand_Luxury                  8028 non-null   bool
 11  brand_Midrange                8028 non-null   bool
 12  brand_Others                  8028 non-null   bool
 13  seller_type_Individual        8028 non-null   bool
 14  seller_type_Trustmark Dealer  8028 non-null   bool
dtypes: bool(5), float64(4), int32(2), int64(4)
memory usage: 666.4 KB
```

Plotting:

```
# # lets drop for now not number columns
# df_only_nums = df[['year', 'selling_price', 'km_driven', 'seats']]
# #only numbers

# previously was keeping categorical as categorical, but now we can
```

```
plot all features
# print(df.corr())

# #correlation map
f, ax = plt.subplots(figsize=(15, 15))
sns.heatmap(df.corr(), annot=True, linewidths=.5, fmt= '.1f',ax=ax,
cmap="coolwarm")

# So the more red color, the more correlation
plt.show()
```



```python
# Observing first 5 data
df.head(5)

# Observing last 5 data
#df.tail()
```

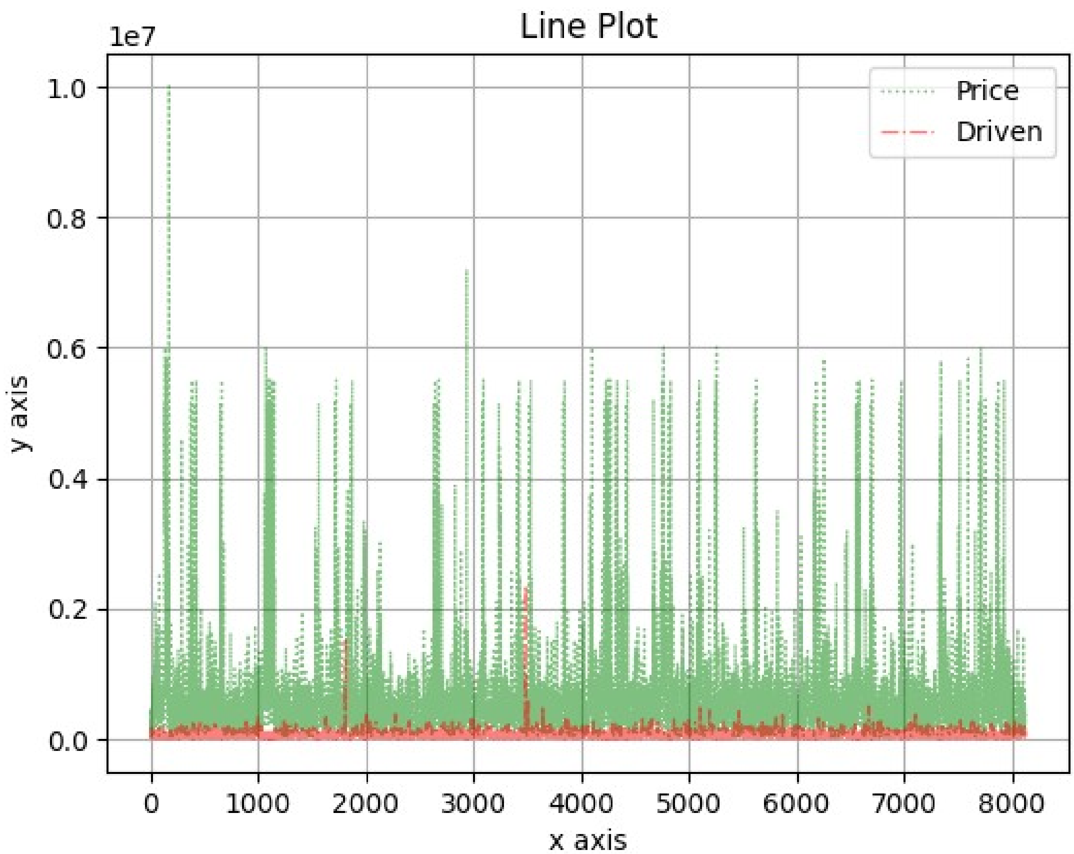
```
   year  selling_price  km_driven  fuel  transmission  owner  mileage
engine  \
0  2014         450000     145500     0             1      1    23.40
1248.0
1  2014         370000     120000     0             1      2    21.14
1498.0
2  2006         158000     140000     1             1      3    17.70
1497.0
3  2010         225000     127000     0             1      1    23.00
1396.0
4  2007         130000     120000     1             1      1    16.10
1298.0

   max_power  seats  brand_Luxury  brand_Midrange  brand_Others  \
0      74.00    5.0         False           False         False
1     103.52    5.0         False            True         False
2      78.00    5.0         False            True         False
3      90.00    5.0         False           False         False
4      88.20    5.0         False           False         False

   seller_type_Individual  seller_type_Trustmark Dealer
0                    True                         False
1                    True                         False
2                    True                         False
3                    True                         False
4                    True                         False
```

Let's try to plot some line, scatter and histogram plots. To choose between, there are some differences in plots:

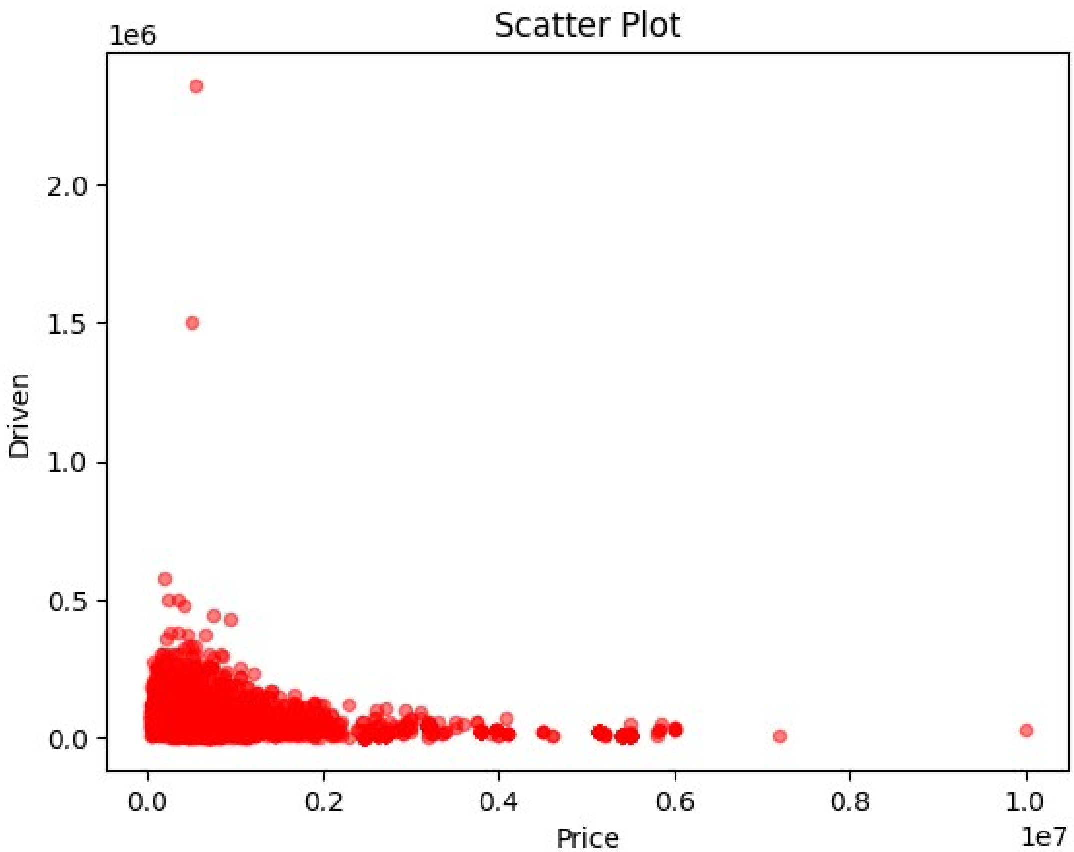
- Line plot is better when x axis is time.
- Box plots: visualize basic statistics like outliers, min/max or quantiles
- Scatter is better when there is correlation between two variables
- Histogram is better when we need to see distribution of numerical data.
- Customization: Colors,labels,thickness of line, title, opacity, grid, figsize, ticks of axis and linestyle

```python
# Line plot
# It might be seen there is no correlation between features,
# but basically I am just exploring type of plots

# Line plot is better when x axis is time
df['selling_price'].plot(kind = 'line', color = 'g',label =
'Price',linewidth=1,alpha = 0.5,grid = True,linestyle = ':')
```

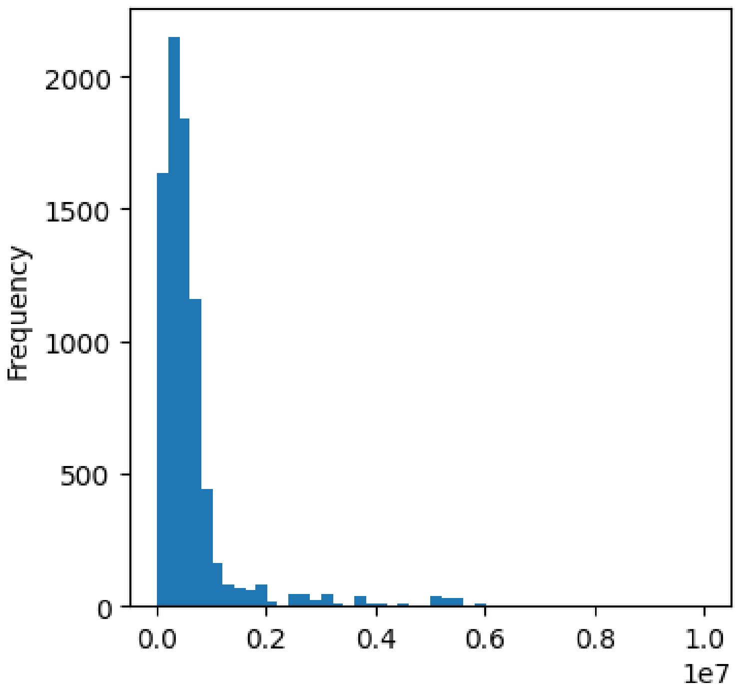
```
df['km_driven'].plot(color = 'r',label = 'Driven',linewidth=1, alpha =
0.5,grid = True,linestyle = '-.')
plt.legend(loc='upper right')     # legend = puts label into plot
plt.xlabel('x axis')              # label = name of label
plt.ylabel('y axis')
plt.title('Line Plot')            # title = title of plot
plt.show()
```




```
# Scatter plot
# Scatter is better when there is correlation between two variables

df.plot(kind='scatter', x='selling_price', y='km_driven',alpha =
0.5,color = 'red')
plt.xlabel('Price')              # label = name of label
plt.ylabel('Driven')
plt.title('Scatter Plot')

Text(0.5, 1.0, 'Scatter Plot')
```

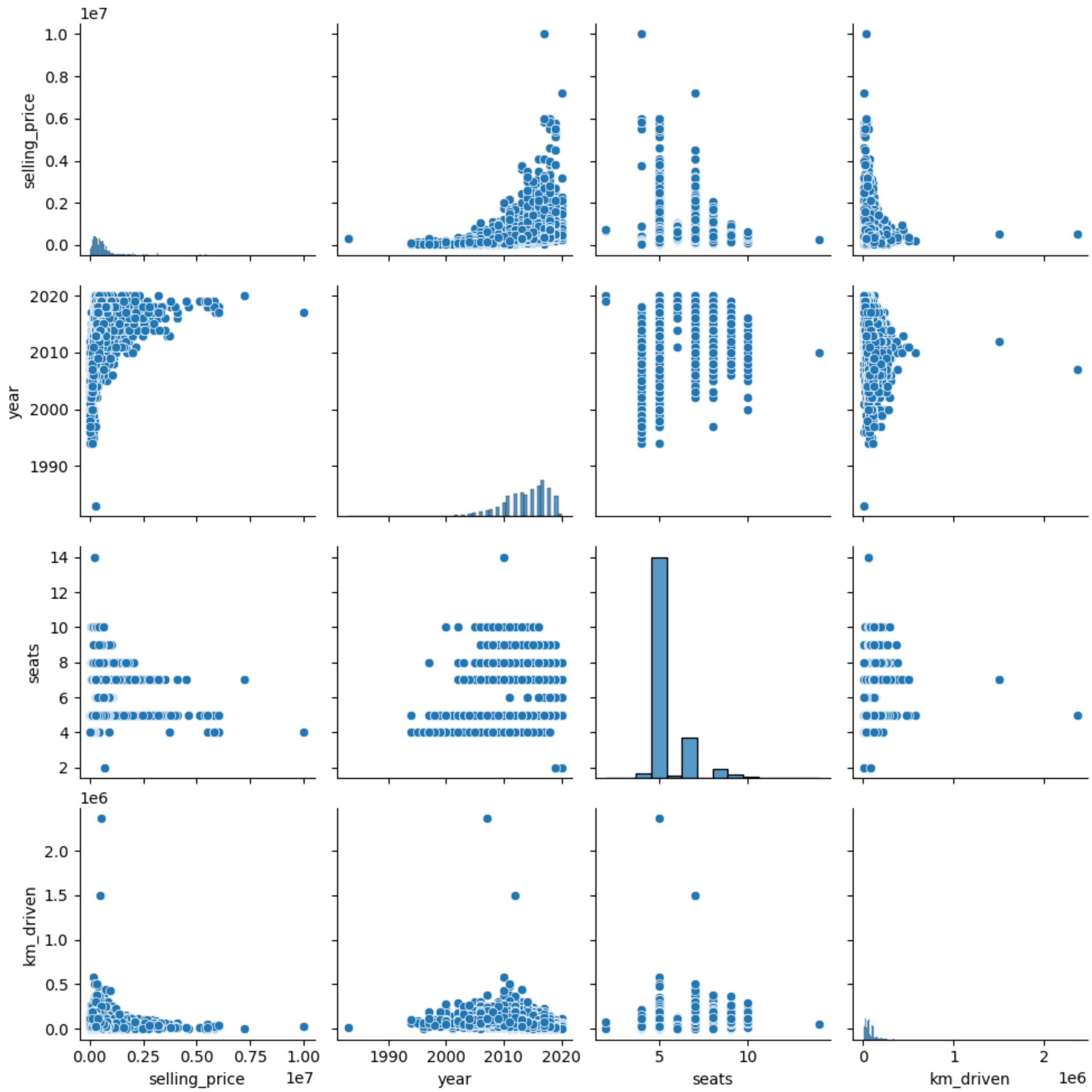



```
# Histogram
# bins = number of bar in figure
# Histogram is better when we need to see distribution of numerical
data.

df['selling_price'].plot(kind = 'hist',bins = 50,figsize = (4,4))
plt.show()
```




```
sns.pairplot(df[['selling_price', 'year', 'seats', 'km_driven']])

# From the result we can see that we need to normalize features
```

```
<seaborn.axisgrid.PairGrid at 0x1a787470940>
```



2.3 Feature Engineering

[08/18/2024] 1st attempt: I think there is no need to create new features, I will try with existing ones (created this section for future use - will require this)

2.4 Feature Selection

```
[08/18/2024] 1st attempt: I am thinking taking all features except the
ones that we need to drop: torque
```

```
[08/20/2024] 2nd attempt: I will choose 5 features that are most
important: max_power, engine, km_driven, mileage, and year
```

```
# Just to remind what are columns
df.columns
```

```
Index(['year', 'selling_price', 'km_driven', 'fuel', 'transmission',
'owner',
       'mileage', 'engine', 'max_power', 'seats', 'brand_Luxury',
       'brand_Midrange', 'brand_Others', 'seller_type_Individual',
       'seller_type_Trustmark Dealer'],
      dtype='object')
```

Outliers

I want to handle them before proceeding to training ['year', 'km_driven', 'mileage', 'engine', 'max_power'] - will chose those only

[08/21/2024] 3rd attempt - trying with outliers before splitting the dataset

```
# To see all outliers

def outlier_count(col, data = df):
    # calculate your 25% quatile and 75% quatile
    q75, q25 = np.percentile(data[col], [75, 25])

    # calculate your inter quatile
    iqr = q75 - q25

    # min_val and max_val
    min_val = q25 - (iqr*1.5)
    max_val = q75 + (iqr*1.5)

    # count number of outliers, which are the data that are less than
min_val or more than max_val calculated above
    outlier_count = len(np.where((data[col] > max_val) | (data[col] <
min_val))[0])

    # calculate the percentage of the outliers
    outlier_percent = round(outlier_count/len(data[col])*100, 2)

    if(outlier_count > 0):
        print("\n"+15*'-' + col + 15*'-'+"\n")
        print('Number of outliers: {}'.format(outlier_count))
        print('Percent of data that is outlier: {}
%'.format(outlier_percent))

# Printing outliers per column
for col in ['year', 'km_driven', 'mileage', 'engine', 'max_power']:
    outlier_count(col)
```

```
---------------year---------------

Number of outliers: 78
Percent of data that is outlier: 0.97%

---------------km_driven---------------

Number of outliers: 168
Percent of data that is outlier: 2.09%
```

Let's not remove them, but cap them to a fixed value (5th or 95th percentile) - reduce impact of extreme values

```
# Capping outliers
def cap_outliers(df, column):
    lower_limit = df[column].quantile(0.05)
    upper_limit = df[column].quantile(0.95)
    df[column] = np.where(df[column] < lower_limit, lower_limit,
df[column])
    df[column] = np.where(df[column] > upper_limit, upper_limit,
df[column])
    return df

# applying for 'year' and 'km_driven' since there are only two
outliers
# for chosen set of features
df = cap_outliers(df, 'year')
df = cap_outliers(df, 'km_driven')

# Printing outliers per column
for col in ['year', 'km_driven', 'mileage', 'engine', 'max_power']:
    outlier_count(col)

# Same approach as in label encoding
df_copy = df.copy()

# shape (m,)
y = df_copy['selling_price']
# df_copy = df_copy.drop(columns=['selling_price'])
print(y.shape)
assert len(y.shape) == 1

# Taking shape (m, n)
X = df_copy[['year', 'km_driven', 'mileage', 'engine', 'max_power']]
print(X.shape)
assert len(X.shape) == 2

(8028,)
(8028, 5)
```

```python
from sklearn.model_selection import train_test_split

# Splitting the dataset, will proceed with processing it
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.13, random_state=42)
```

2.5 Preprocessing

```python
# Let's see what is the train dataset size
X_train.shape
```

```
(6984, 5)
```

```python
# Same for test
X_test.shape
```

```
(1044, 5)
```

NUll values

```python
# Let's observe all null values in training set (did not deal with
them - to avoid data leakage)
X_train.isna().sum()
```

```
year            0
km_driven       0
mileage       187
engine        187
max_power     181
dtype: int64
```

```python
# Same for the testing dataset
X_test.isna().sum()
```

```
year           0
km_driven      0
mileage       27
engine        27
max_power     27
dtype: int64
```

```python
# Removing null values for mileage
print(X_train.mileage.mean(), X_train.mileage.median())

# sns.distplot(X_train, x=X_train.mileage)

# Interchanging nan values with mean - the distribution is normal
X_train.mileage.fillna(X_train.mileage.mean(), inplace=True)
X_test.mileage.fillna(X_train.mileage.mean(), inplace=True)
```

```
19.38204354862439 19.3

C:\Users\eraco\AppData\Local\Temp\ipykernel_31172\2985369338.py:7:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  X_train.mileage.fillna(X_train.mileage.mean(), inplace=True)
C:\Users\eraco\AppData\Local\Temp\ipykernel_31172\2985369338.py:8:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  X_test.mileage.fillna(X_train.mileage.mean(), inplace=True)
```

```
# df_copy.engine.isna().sum() # 214
print(X_train.engine.mean(), X_train.engine.median())

# sns.distplot(X_train, x=X_train['engine'])

# Interchanging nan values with median - the distribution is skewed
X_train.engine.fillna(X_train.engine.mean(), inplace=True)
X_test.engine.fillna(X_train.engine.mean(), inplace=True)
```

```
1463.756068853906 1248.0

C:\Users\eraco\AppData\Local\Temp\ipykernel_31172\2940628221.py:7:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
```

```
df[col].method(value) instead, to perform the operation inplace on the
original object.


  X_train.engine.fillna(X_train.engine.mean(), inplace=True)
C:\Users\eraco\AppData\Local\Temp\ipykernel_31172\2940628221.py:8:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  X_test.engine.fillna(X_train.engine.mean(), inplace=True)
```

```
# df_copy.max_power.isna().sum() # 208
print(X_train.max_power.mean(), X_train.max_power.median())

# sns.distplot(X_train, x=X_train.max_power) # distribution is skewed
a little

# Interchanging nan values with median - the distribution is skewed
X_train.max_power.fillna(X_train.max_power.mean(), inplace=True)
X_test.max_power.fillna(X_train.max_power.mean(), inplace=True)
```

```
91.74543877701014 82.85

C:\Users\eraco\AppData\Local\Temp\ipykernel_31172\3896574348.py:7:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  X_train.max_power.fillna(X_train.max_power.mean(), inplace=True)
C:\Users\eraco\AppData\Local\Temp\ipykernel_31172\3896574348.py:8:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
```

```
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  X_test.max_power.fillna(X_train.max_power.mean(), inplace=True)
```

```python
# # And we want to remove all null values from seats feature - 214
rows
# print(X_train.seats.mean(), X_train.seats.median())

# # sns.distplot(X_train, x=X_train.seats) # distribution is skewed a
little

# # Interchanging nan values with median - the distribution is skewed
# X_train.seats.fillna(X_train.seats.mean(), inplace=True)
# X_test.seats.fillna(X_train.seats.mean(), inplace=True)

# Now verify if everything is fine
X_train.isna().sum()
```

```
year         0
km_driven    0
mileage      0
engine       0
max_power    0
dtype: int64
```

```python
# Same for test set
X_test.isna().sum()
```

```
year         0
km_driven    0
mileage      0
engine       0
max_power    0
dtype: int64
```

```python
# Just to be sure
y_train.isna().sum()
```

```
0
```

```python
# Now we can proceed
y_test.isna().sum()
```

```
0
```

Scaling

```
# Observing what need to be scaled
X_train.head()
```

```
        year  km_driven  mileage  engine  max_power
4419  2016.0    68089.0    19.16  2494.0     157.70
6103  2011.0    81500.0    14.84  2143.0     167.62
7893  2011.0   140000.0    14.40  1598.0     103.60
7427  2016.0   120000.0    13.58  2499.0      72.40
1448  2018.0    30000.0    18.60  1197.0      81.83
```

```
# We need to scale all numerics whose difference is large
from sklearn.preprocessing import StandardScaler

# After observing above, we can proceed with the following columns
col_names = ['year', 'km_driven', 'mileage', 'engine', 'max_power']

# Defining Scaler
sc = StandardScaler()

# Scaling is performed
X_train[col_names] = sc.fit_transform(X_train[col_names])
X_test[col_names] = sc.transform(X_test[col_names])

# Let's see if its fine
X_train.head()
```

```
          year  km_driven   mileage    engine  max_power
4419  0.559461   0.028728 -0.056110  2.064080   1.885161
6103 -0.806911   0.364115 -1.147758  1.360856   2.168701
7893 -0.806911   1.827105 -1.258944  0.268956   0.338836
7427  0.559461   1.326937 -1.466156  2.074098  -0.552945
1448  1.106010  -0.923816 -0.197620 -0.534442  -0.283410
```

```
# Same for test set
X_test.head()
```

```
          year  km_driven   mileage    engine  max_power
5948 -0.806911  -0.798774 -1.107327 -0.191846  -0.049889
6039  0.559461   1.076853  0.072766  0.236900   0.984805
3069  0.286187  -0.173565  0.285031 -0.933136  -0.706149
6531  0.012912   0.326602 -0.094014 -0.005522   0.477462
322   1.106010  -0.948824  1.166941 -0.556481  -0.508070
```

```
# Same for selling price, we want to do np.log transformation
# y_train = np.log(y_train)
# y_train

# We dont need to log the actual target set - so leaving it
# y_test = np.log(y_test)
```

```
X_train = X_train.to_numpy()
y_train = y_train.to_numpy()

X_test = X_test.to_numpy()
y_test = y_test.to_numpy()
```

3. Modeling

[8/19/2024] first attempt - at first iteration, I proceeded with RandomForest, using cross_validation+gridsearch found the

```
best params: {'bootstrap': True, 'max_depth': None, 'n_estimators':
15}
mse error: -0.04769509278891849
(y_test is scaled into np.log - otherwise error is very huge - because
y_test and y_preds are huge numbers)
```

Also looked for grid.best_estimator_.feature_importances_:

```
0     year  0.464258
7     max_power  0.396566
6     engine     0.053404
1     km_driven  0.026052
5     mileage    0.025340
8     seats 0.008029
10    brand_Midrange   0.007669
4     owner 0.006844
9     brand_Luxury     0.006078
2     fuel  0.002882
12    seller_type_Individual     0.001563
3     transmission     0.001192
11    brand_Others     0.000072
13    seller_type_Trustmark Dealer     0.000052
```

Configuring MlFlow workplace

```
mlflow_url = 'https://mlflow.ml.brain.cs.ait.ac.th/'
mlflow.set_tracking_uri(mlflow_url)

os.environ["MLFLOW_TRACKING_USERNAME"] = "admin"
os.environ["MLFLOW_TRACKING_PASSWORD"] = "password"

os.environ["LOGNAME"] = "st125457-ulugbek"

mlflow.set_experiment(experiment_name="st125457-ulugbek-experiment")

import logging
```

```
# Setting only logging warning messages, later it would make output
neat
logging.getLogger("mlflow").setLevel(logging.WARNING)

class LinearRegression(object):

    #in this class, we add cross validation as well for some spicy
code....
    kfold = KFold(n_splits=3)

    def __init__(self, regularization=None, lr=0.001, method='batch',
momentum=None, isXavier=False, num_epochs=50, batch_size=32, cv=kfold,
l=''):
        self.lr         = lr
        self.num_epochs = num_epochs
        self.batch_size = batch_size
        self.method     = method
        self.cv         = cv
        self.regularization = regularization

        # For plotting in mlflow
        self.kfold_epoch_mse = []

        # Added xavier initialization flag
        self.isXavier = isXavier

        # Added momentum
        self.momentum = momentum
        self.prev_step = 0

        # For choosing best model upon training
        self.last_r2_score = 0

    def mse(self, ytrue, ypred):
        # return ((ypred - ytrue) ** 2).sum() / ytrue.shape[0]
        return np.mean((ytrue - ypred) ** 2)

    def score(self, y_target, yhat):
        ss_res = np.sum((y_target - yhat) ** 2)
        ss_tot = np.sum((y_target - np.mean(y_target)) ** 2)
        return 1 - ss_res / ss_tot

    def xavier_initialize(self, m):
        # The pseudocode provided solution is below
        # But I dont think it is the correct way to do it
        # Usually it uses:
        # # lower = -np.sqrt(6) / np.sqrt(n_in + n_out)
        # # upper = np.sqrt(6) / np.sqrt(n_in + n_out)
```

```
        # But will proceed with the pseudocode
        lower, upper = -1.0 / np.sqrt(m), 1.0 / np.sqrt(m)

        # to get the same results
        np.random.seed(52)

        # numbers = np.random.rand(m)
        numbers = np.random.uniform(lower, upper, m)

        return lower + numbers * (upper - lower)
        # lower = -np.sqrt(6) / np.sqrt(m + 1)  # 1 for output
dimension
        # upper = np.sqrt(6) / np.sqrt(m + 1)
        # return np.random.uniform(lower, upper, m)

    def fit(self, X_train, y_train):
        if isinstance(X_train, pd.DataFrame):
            X_train = X_train.to_numpy()
        if isinstance(y_train, pd.Series):
            y_train = y_train.to_numpy()

        #create a list of kfold scores
        self.kfold_scores = list()

        #reset val loss
        self.val_loss_old = np.infty

        #kfold.split in the sklearn.....
        #5 splits
        for fold, (train_idx, val_idx) in
enumerate(self.cv.split(X_train)):
            X_cross_train = X_train[train_idx]
            y_cross_train = y_train[train_idx]
            X_cross_val   = X_train[val_idx]
            y_cross_val   = y_train[val_idx]

            if self.isXavier:
                self.theta =
self.xavier_initialize(X_cross_train.shape[1])
            else:
                self.theta = np.zeros(X_cross_train.shape[1])

            #one epoch will exhaust the WHOLE training set
            with mlflow.start_run(run_name=f"Fold-{fold}",
nested=True):

                params = {"method": self.method, "lr": self.lr, "reg":
type(self).__name__, "xavier_initialization": self.isXavier}
                mlflow.log_params(params=params)
```

```
                for epoch in range(self.num_epochs):
                    # self.learning_rate_decay(epoch)

                    #with replacement or no replacement
                    #with replacement means just randomize
                    #with no replacement means 0:50, 51:100,
101:150, ......300:323
                    #shuffle your index
                    perm =
np.random.permutation(X_cross_train.shape[0])

                    X_cross_train = X_cross_train[perm]
                    y_cross_train = y_cross_train[perm]

                    if self.method == 'sgd':
                        for batch_idx in
range(X_cross_train.shape[0]):
                            X_method_train =
X_cross_train[batch_idx].reshape(1, -1) #(11,) ==> (1, 11) ==> (m, n)
                            y_method_train = y_cross_train[batch_idx]
                            train_loss = self._train(X_method_train,
y_method_train)
                    elif self.method == 'mini':
                        for batch_idx in range(0,
X_cross_train.shape[0], self.batch_size):
                            X_method_train =
X_cross_train[batch_idx:batch_idx+self.batch_size, :]
                            y_method_train =
y_cross_train[batch_idx:batch_idx+self.batch_size]
                            train_loss = self._train(X_method_train,
y_method_train)
                    else:
                        X_method_train = X_cross_train
                        y_method_train = y_cross_train
                        train_loss = self._train(X_method_train,
y_method_train)

                    # Appending metrics
                    self.kfold_epoch_mse.append(train_loss)
                    mlflow.log_metric(key="train_loss",
value=train_loss, step=epoch)

                    yhat_val = self.predict(X_cross_val)
                    val_loss_new = self.mse(y_cross_val, yhat_val)
                    mlflow.log_metric(key="val_loss",
value=val_loss_new, step=epoch)

                    val_r2_score = self.score(y_cross_val, yhat_val)
                    mlflow.log_metric(key="val_r2_score",
value=val_r2_score, step=epoch)
```

```
                    #early stopping - modified, because it was
stopping too early
                    # if np.abs(val_loss_new - self.val_loss_old) <
1e-6:
                    if np.allclose(val_loss_new, self.val_loss_old):
                        break

                    self.val_loss_old = val_loss_new

                self.last_r2_score = val_r2_score
                self.kfold_scores.append(val_loss_new)
                print(f"Fold {fold}: {val_loss_new}")


    def learning_rate_decay(self, epoch):
        self.lr = self.lr * (0.95 ** (epoch // 10))

    def _train(self, X, y):
        yhat = self.predict(X)
        # print("PREDICTION INSIDE TRAIN: ", yhat.reshape(1, -1))
        m    = X.shape[0]
        grad = (1/m) * X.T @ (yhat - y)

        if self.regularization:
            grad += self.regularization.derivation(self.theta)

        # Momentum implementation
        if self.momentum and 0 <= self.momentum < 1:
            step = self.lr * grad
            self.theta = self.theta - step + self.momentum *
self.prev_step
            self.prev_step = step
        else:
            if self.momentum and self.momentum >= 1:
                print("The value of momentum is more than allowed [0,
1], switching to version without momentum")
            self.theta = self.theta - self.lr * grad
            self.prev_step = 0
        return self.mse(y, yhat)

    def predict(self, X, to_transform=False):
        if isinstance(X, pd.DataFrame):
            X = X.to_numpy()
        return X @ self.theta  #===>(m, n) @ (n, )

    def _coef(self):
        return self.theta[1:]  #remind that theta is (w0, w1, w2, w3,
w4.....wn)
                               #w0 is the bias or the intercept
```

```
                              #w1....wn are the weights /
coefficients / theta
    def _bias(self):
        return self.theta[0]

    def plot_feature_importance(self, feature_names=None):
        if not hasattr(self, 'theta'):
            raise ValueError("Model coefficients are not available.
Fit the model first.")

        # Coefficients
        coefficients = self._coef()
        importance = np.abs(coefficients)

        # Assign default names if feature_names are not provided
        if feature_names is None:
            feature_names = [f"Feature {i}" for i in range(1,
len(coefficients) + 1)]

        # Sort features by importance
        mask = np.argsort(importance)[::-1]
        sorted_importance = importance[mask]
        sorted_feature_names = np.array(feature_names)[mask]

        # Plotting
        plt.figure(figsize=(10, 6))
        plt.barh(sorted_feature_names, sorted_importance,
color='skyblue')
        plt.xlabel('Coefficient Magnitude (Absolute)')
        plt.title('Feature Importance based on Coefficients')
        plt.gca().invert_yaxis()  # To display the most important
feature at the top
        plt.show()

        # Test
        # sorted_idx = rf.feature_importances_.argsort()
        # plt.barh(X.columns[sorted_idx],
rf.feature_importances_[sorted_idx])
        # plt.xlabel("Random Forest Feature Importance")

class LassoPenalty:

    def __init__(self, l):
        self.l = l # lambda value

    def __call__(self, theta): #__call__ allows us to call class as
method
        return self.l * np.sum(np.abs(theta))

    def derivation(self, theta):
```

```
        return self.l * np.sign(theta)

class RidgePenalty:

    def __init__(self, l):
        self.l = l

    def __call__(self, theta): #__call__ allows us to call class as
method
        return self.l * np.sum(np.square(theta))

    def derivation(self, theta):
        return self.l * 2 * theta

class ElasticPenalty:

    def __init__(self, l = 0.1, l_ratio = 0.5):
        self.l = l
        self.l_ratio = l_ratio

    def __call__(self, theta):  #__call__ allows us to call class as
method
        l1_contribution = self.l_ratio * self.l *
np.sum(np.abs(theta))
        l2_contribution = (1 - self.l_ratio) * self.l * 0.5 *
np.sum(np.square(theta))
        return (l1_contribution + l2_contribution)

    def derivation(self, theta):
        l1_derivation = self.l * self.l_ratio * np.sign(theta)
        l2_derivation = self.l * (1 - self.l_ratio) * theta
        return (l1_derivation + l2_derivation)

class Lasso(LinearRegression):

    def __init__(self, method, lr, l, momentum, isXavier):
        self.regularization = LassoPenalty(l)
        super().__init__(self.regularization, lr, method, momentum,
isXavier)

class Ridge(LinearRegression):

    def __init__(self, method, lr, l, momentum, isXavier):
        self.regularization = RidgePenalty(l)
        super().__init__(self.regularization, lr, method, momentum,
isXavier)

class ElasticNet(LinearRegression):
```

```
    def __init__(self, method, lr, l, momentum, isXavier,
l_ratio=0.5):
        self.regularization = ElasticPenalty(l, l_ratio)
        super().__init__(self.regularization, lr, method, momentum,
isXavier)
```

Polynomial regression for task2

```
from sklearn.preprocessing import PolynomialFeatures


class PolynomialRegression(LinearRegression):

    def __init__(self, method, lr, l, momentum, isXavier, degree=2):
        self.degree = degree
        self.poly = PolynomialFeatures(degree=degree)
        # Using Ridge as regularization
        self.regularization = RidgePenalty(l)
        super().__init__(self.regularization, lr, method, momentum,
isXavier)

    def fit(self, X_train, y_train):
        # Transform the input data to polynomial features
        X_poly = self.poly.fit_transform(X_train)

        # Use the base class's fit method to train the model
        super().fit(X_poly, y_train)

    def predict(self, X, to_transform=False):
        X_poly = X
        # Transform the input data to polynomial features before
making predictions
        if to_transform:
            X_poly = self.poly.transform(X)
        return super().predict(X_poly)

y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.reshape(-1,
1)).flatten()
y_train_scaled
```

```
array([ 1.74438242,  0.9132295 , -0.49333698, ...,  0.14601142,
        5.77227733,  0.17797884])
```

```
import sys


def str_to_class(classname):
    return getattr(sys.modules[__name__], classname)
```

```
def run_experiment_cross_validation(X_train, y_train):

    # Define parameter grid
    learning_rates = [0.01, 0.001, 0.0001]
    momentum_values = [None, 0.9]
    initializations = [False, True]
    methods = ['sgd', 'mini', 'batch']

    models = ['LinearRegression', 'Lasso', 'Ridge',
'PolynomialRegression']

    trained_models = []

    r2_scores = []

    mlflow_params = []
    for model_name in models:
            for lr in learning_rates:
                for momentum in momentum_values:
                    for method in methods:
                        for is_xavier in initializations:
                            mlflow_prm =
f"{model_name}_lr{lr}_momentum{momentum}_method{method}_xavier{is_xavi
er}"
                            mlflow_params.append(mlflow_prm)

                            with mlflow.start_run(run_name=mlflow_prm,
nested=True):
                                # Performing custom cross-validation
                                print(model_name)
                                params = {"method": method, "lr": lr,
"momentum": momentum, "isXavier": is_xavier, "l": 0.1}

                                type_of_regression =
str_to_class(model_name)
                                model = type_of_regression(**params)

                                model.fit(X_train, y_train)

                                trained_models.append(model)

                                # Metrics
                                r2_scores.append(model.last_r2_score)

                                for i, mse_score in
enumerate(model.kfold_epoch_mse):

mlflow.log_metric(key="train_kfold_epochs_mse", value=mse_score,
step=i)
```

```
                                for i, kfold_mse in
enumerate(model.kfold_scores):

mlflow.log_metric(key="train_kfold_mse", value=kfold_mse, step=i)


mlflow.log_metric(key='train_r2_score', value=model.last_r2_score)

                                print(f"{model_name}: MSE =
{model.kfold_scores[-1]}, R2_score = {r2_scores[-1]}")
                                signature =
mlflow.models.infer_signature(X_train, model.predict(X_train,
to_transform=True))
                                mlflow.sklearn.log_model(model,
artifact_path='model', signature=signature)

    # returning Best model based on r2_score
    index = r2_scores.index(min(r2_scores))
    return trained_models[index], mlflow_params[index]


def run_experiment_test_best_model(model, mlflow_prms):
    mlflow.start_run(run_name=mlflow_prms, nested=True)
    yhat = model.predict(X_test, to_transform=True)
    print("YHAT: ", yhat)

    predict = y_scaler.inverse_transform(yhat.reshape(-1,
1)).flatten()

    score = model.score(predict, y_test)
    print("Test R2_Score: ", score)
    mlflow.log_metric(key="test_r2_score", value=score)

    mse  = model.mse(predict, y_test)
    print("Test MSE: ", mse)
    mlflow.log_metric(key="test_mse", value=mse)

    # kfold_metrics_mse = {f"train_kfold_mse_step_{i}": score for i,
score in enumerate(model.kfold_scores)}
    # mlflow.log_metric(kfold_metrics_mse)

    for i, mse_score in enumerate(model.kfold_epoch_mse):
        mlflow.log_metric(key="train_kfold_epochs_mse",
value=mse_score, step=i)

    for i, kfold_mse in enumerate(model.kfold_scores):
        mlflow.log_metric(key="train_kfold_mse", value=kfold_mse,
step=i)

    signature = mlflow.models.infer_signature(X_train,
model.predict(X_train, to_transform=True))
```

```
    mlflow.sklearn.log_model(model, artifact_path='model',
signature=signature)

    print("ENDING")
    mlflow.end_run()

# Run the experiment
model, mlflow_prms = run_experiment_cross_validation(X_train,
y_train_scaled)
run_experiment_test_best_model(model, mlflow_prms)
```

Some info about model training process

Below is the trained models logs with epoch 50 which took me 4 hours to run, I found best model which is PolynomialRegression based on r2_score, will rerun it below with all required parameters:

```
All logs would be found in 'cross_validation_logs.log' file because
the output is huge.
```

Best model

So the best model based on r2_score is PolynomialRegression_lr0.0001_momentumNone_methodsgd_xavierTrue (according to logs and mlflow training runs) including such parameters. It got r2_score 0.84 which is really close to 1. Note: I used Ridge regularization for Polynomial Regression, and I think it performed better, but will test both with Ridge regularization and pure regularization based on the parameters I found.

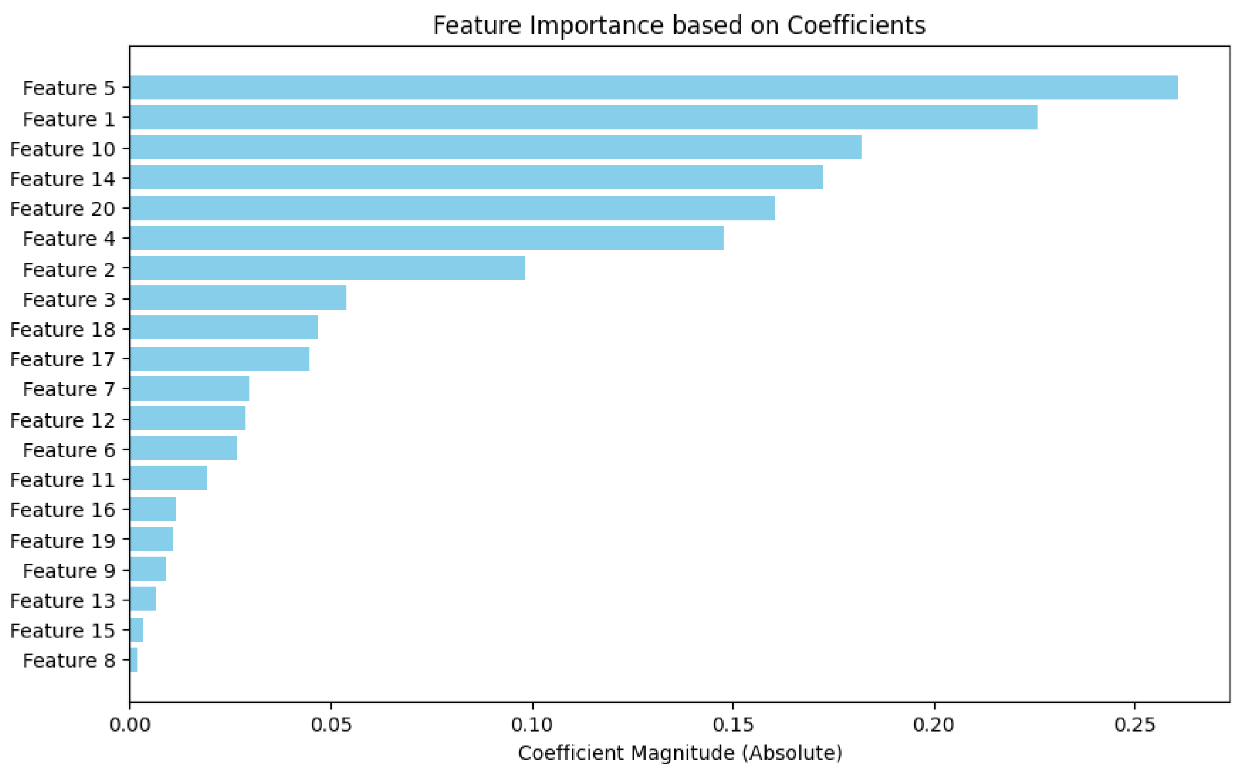
Will test it on testing set, training it again.

```
# PolynomialRegression_lr0.0001_momentumNone_methodsgd_xavierTrue

model_name = 'PolynomialRegression'
lr = 0.0001
momentum = None
method = 'sgd'
is_xavier = True

mlflow_prm =
f"Best_model_{model_name}_lr{lr}_momentum{momentum}_method{method}_xav
ier{is_xavier}"


with mlflow.start_run(run_name=mlflow_prm, nested=True):

    params = {"method": method, "lr": lr, "momentum": momentum,
"isXavier": is_xavier, "l": 0.1}
```

```
    type_of_regression = str_to_class(model_name)
    model = type_of_regression(**params)

    model.fit(X_train, y_train_scaled)

    print("Train_r2_score: ", model.last_r2_score)
    mlflow.log_metric(key='train_r2_score', value=model.last_r2_score)

    yhat = model.predict(X_test, to_transform=True)
    print("YHAT: ", yhat)

    predict = y_scaler.inverse_transform(yhat.reshape(-1,
1)).flatten()

    score = model.score(predict, y_test)
    print("Test R2_Score: ", score)
    mlflow.log_metric(key="test_r2_score", value=score)

    mse  = model.mse(predict, y_test)
    print("Test MSE: ", mse)
    mlflow.log_metric(key="test_mse", value=mse)

    for i, mse_score in enumerate(model.kfold_epoch_mse):
        mlflow.log_metric(key="train_kfold_epochs_mse",
value=mse_score, step=i)

    for i, kfold_mse in enumerate(model.kfold_scores):
        mlflow.log_metric(key="train_kfold_mse", value=kfold_mse,
step=i)

    signature = mlflow.models.infer_signature(X_train,
model.predict(X_train, to_transform=True))
    mlflow.sklearn.log_model(model, artifact_path='model',
signature=signature)
```

```
Fold 0: 0.14683410899757604
Fold 1: 0.15637831070545702
Fold 2: 0.15516829127654766
Train_r2_score:  0.8425119675725541
YHAT:  [-0.39698406  0.24175563 -0.39826621 ... -0.33011989 -
0.44589894
 -0.49793023]
Test R2_Score:  0.7959747276746449
Test MSE:  156900205360.3569
```

```
# Plotting features -> I cant plot with feature names since there are
only 5 features are used, but polynomial regression will expand those
features to be 21 features, that is why will show all features as
feature[1-20]
model.plot_feature_importance()
```



Let's test this model for other scores as well

```
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.metrics import mean_absolute_error
import numpy as np

preds = model.predict(X_test, to_transform=True)
yhat = y_scaler.inverse_transform(preds.reshape(-1, 1)).flatten()
print("PREDICTED DATA: ", yhat[:10])
print("ACTUAL DATA: ", list(y_test[:10]))

pred_y = yhat

# Doing just MSE is not okay, since our predictions are huge numbers
and there are outliers, the mse is not accurate
print("MSE: ", mean_squared_error(y_test, pred_y))

# RMSE performs better, but still not so good
print("RMSE: ", np.sqrt(mean_squared_error(y_test, pred_y)))

# So from this it is clear that squaring is really bad idea, since we
get huge numbers as per selling_price
# Therefore better to proceed with other metrics like below:

# MAE
print("MAE: ", mean_absolute_error(y_test, pred_y))
# Performance of MAE is good - 71856. It means we can use this model
```

```
for deploying.

# Percentage error seems fine, we got 15% error
percentage_error = np.abs((pred_y - y_test) / y_test) * 100
mean_percentage_error = np.mean(percentage_error)
print("Mean Percentage Error:", mean_percentage_error)

# We can also do cosine similarity to check whether the predictions is
fine or not
# Reshape the vectors to be 2D arrays for cosine_similarity function
y_test_reshaped = y_test.reshape(1, -1)
pred_y_reshaped = pred_y.reshape(1, -1)

# Cosine similarity predictions
cos_sim = cosine_similarity([y_test], [pred_y])
print("Cosine Similarity:", cos_sim[0][0])

PREDICTED DATA:  [325352.43767865 824876.39628472 324349.72966635
604089.3096096
 567639.05412669 226084.32727238 381765.0732805  492923.38179561
 343738.78894688 380949.12801269]
ACTUAL DATA:  [225000, 900000, 320000, 650000, 520000, 170000, 280000,
500000, 170000, 335000]
MSE:  156900205360.3569
RMSE:  396106.3056306437
MAE:  187608.2103977607
Mean Percentage Error: 43.63912321035114
Cosine Similarity: 0.9388036545362345

# Saving the model

import joblib

# save the model to disk
filename =
'polynomialRegression_lr0_0001_momentumNone_methodsgd_xavierTrue.pkl'
joblib.dump(model, filename)

['polynomialRegression_lr0_0001_momentumNone_methodsgd_xavierTrue.pkl'
]
```

Task 2.3: Report - Conclusion

So regarding the findings, let's break it down into parts:

1. Integrating the discussed concepts (Xavier initialization, momentum, plot_feature_importances, r2_score) was fun, because I tried to do all code from my perspective as I read the documentation/research paper. I go with r2_score first, it was pretty straightforward, but I had interest to check how it was implemented in scikit-learn

- found new objectives because they also add 'sample_weight' which is multiplied to both SS_res and SS_tot to normalize the output. Then I go with Xavier Initialization which is actually different then the pseudocode provided (also wrote in comments the widely used version: the one with sqrt(6)), reading the paper helped me to understand why it is so important and the idea. Then, I go with momentum which is straightforward as well, the pseudocode was fine. Plotting feature importances was a bit challenging until I understood the feature importance selection process. Also noticed that early stopping is very rough, tried with more appropriate one which was making my further training in cross validation to run eternally. Also added random seed for getting same results for xavier initialization.

2. For training model side, I have configured MLflow with AIT deployed MLflow one. Then, for testing purposes, decided to run the initial code block for LinearRegression. In that process, I understood something is wrong - getting huge mse + the kfold values are also huge. Then, after several hours analyzes, I found out that we need to scale our training set (same issue I had at assignment 1 - doing same thing again). Cross validation code part was straightforward, though it was a bit long to run it. I also found out that we need to use Polynomial Regression, so I decided to extend Linear Regression for that case which in result looks good. Also added some more features inside Linear Regression class for different purposes including for plotting graphs in MLflow, doing learning decay, and etc. The process was fun, but training the cross validation loop was time consuming making me run it several times.
3. I made small mistake in choosing best model part which I decided to put inside the training loop (used min for r2_score, which was corrected after some time to max), that is why performing double job - training the model with best params again to get the same result (got the same result).
4. Cosine similarity is doing good, and the predictions are close enough, so I would proceed with this model - Polynomial Regression+Ridge

Interesting facts:

1. I have trained 1175 models (runs) in mlflow for this assignment... image.png
2. Best Model graphics+metrics: image-2.png
3. Registered the model (I don't know why): image-3.png

Final Table: Top-5 according to model training logs in descending order based on r2_score (removed mse score from table because we dont need it): @upd: all top scores are reached by polynomial regression, the table will also include other models as well | model_name | method | momentum | isXavier | learning_rate | r2_score | |----------|----------|----------|----------|----------|----------| | Polynomial Regression | sgd | None | True | 0.0001 | 0.842 | | Polynomial Regression | mini | 0.9 | True | 0.01 | 0.839 | | Polynomial Regression | sgd | 0.9 | True | 0.001 | 0.839 | | Linear Regression | sgd | 0.9 | Flase | 0.001 | 0.657 | | Lasso | sgd | None | True | 0.001 | 0.653 |