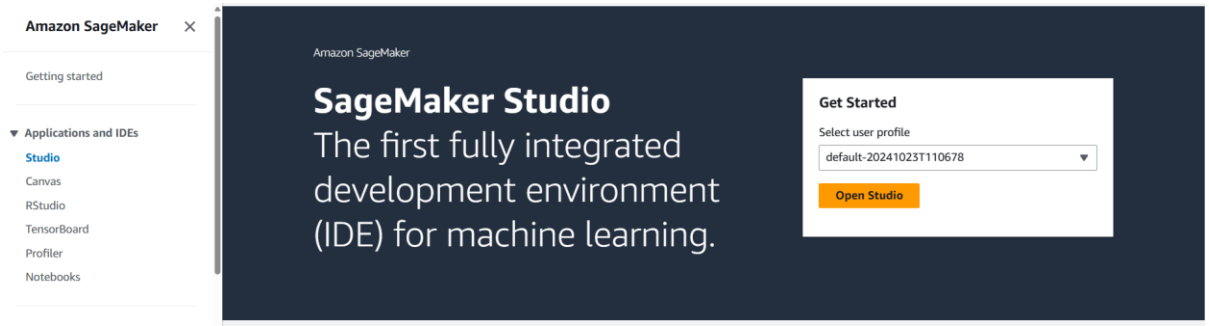
😊 SageMaker Studio for Medical Insurance Premium Prediction

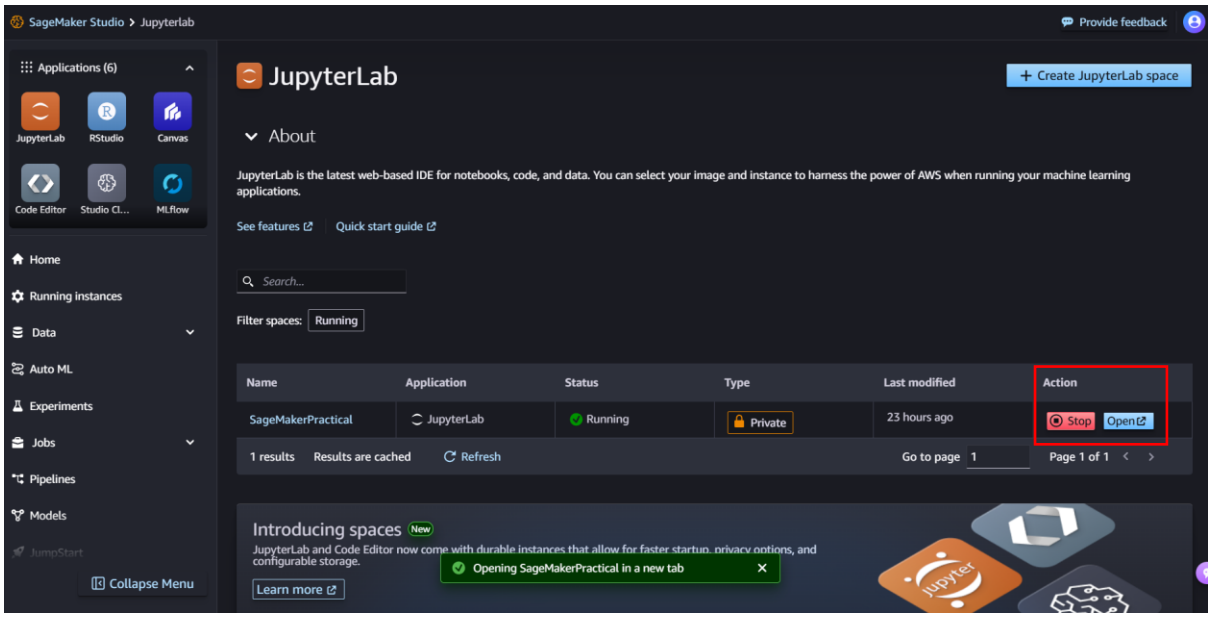
The objective of this case study is to predict the health insurance cost incurred by Individuals based on their age, gender, BMI, number of children, smoking habit and geo-location.

😄 To begin with the Lab:

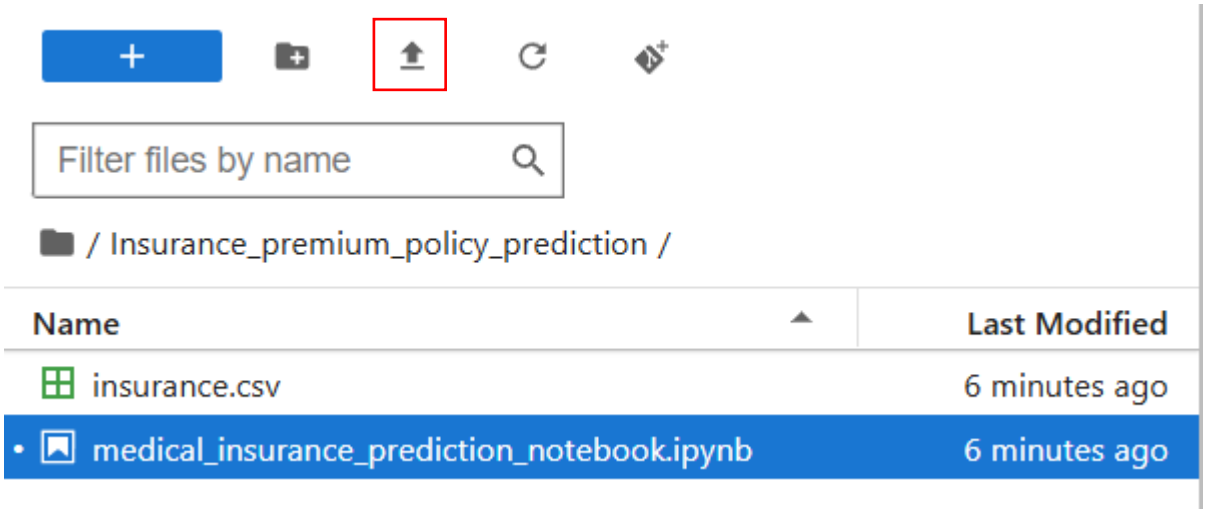
1. Login to your AWS Console and search for SageMaker and go to it.
2. Then open your SageMaker Studio and you will on a new tab.



3. Now click on Jupyter Lab and here you will see your Jupyter Lab Space you need to start it from the action and then click on Open.



4. Once you are inside the Jupyter Lab then you need to click on upload icon and upload these two files as you can see below. After that open medical insurance prediction notebook.



5. In the task 1 you can read it and understand the problem statement for this lab.

TASK #1: UNDERSTAND THE PROBLEM STATEMENT

- Aim of the problem is to find the health insurance cost incured by Individuals based on thier age, gender, BMI, number of children, smoking habit and geo-location.
- Features available are:
  - sex: insurance contractor gender, female, male
  - bmi: Body mass index (ideally 18.5 to 24.9)
  - children: Number of children covered by health insurance / Number of dependents
  - smoker: smoking habits
  - region: the beneficiary's residential area in the US, northeast, southeast, southwest, northwest.
  - charges: Individual medical costs billed by health insurance

Data Source:https://www.kaggle.com/mirichoi0218/insurance

6. Then in the task 2 we import the necessary libraries and data sets.

TASK #2: IMPORT LIBRARIES AND DATASETS

TYPES OF AVAILABLE SAGEMAKER IMAGES

- Data Science [datascience-1.0]: Data Science is a Conda image with the most commonly used Python packages and libraries, such as NumPy and SciKit Learn.
- Base Python [python-3.6]
- MXNet (optimized for CPU) [mxnet-1.6-cpu-py36]
- MXNet (optimized for GPU) [mxnet-1.6-gpu-py36]
- PyTorch (optimized for CPU) [pytorch-1.4-cpu-py36]
- PyTorch (optimized for GPU) [pytorch-1.4-gpu-py36]
- TensorFlow (optimized for CPU) [tensorflow-1.15-cpu-py36]
- TensorFlow (optimized for GPU) [tensorflow-1.15-gpu-py36]
- TensorFlow 2 (optimized for CPU) [tensorflow-2.1-cpu-py36]
- TensorFlow 2 (optimized for GPU) [tensorflow-2.1-gpu-py36]

[2]:
```
!pip install seaborn
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

7. After we make use of pandas to read our CSV file and then we displayed the first and last 5 values of our data frame.

MINI CHALLENGE

- Read the CSV file "insurance.csv" using pandas
- Visualize the first and last 5 rows

[3]:
```
# read the csv file
insurance_df = pd.read_csv('insurance.csv')
```

[4]:
```
insurance_df.head()
```

[4]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

[5]:
```
insurance_df.tail()
```

[5]:

	age	sex	bmi	children	smoker	region	charges
1333	50	male	30.97	3	no	northwest	10600.5483
1334	18	female	31.92	0	no	northeast	2205.9808
1335	18	female	36.85	0	no	southeast	1629.8335
1336	21	female	25.80	0	no	southwest	2007.9450
1337	61	female	29.07	0	yes	northwest	29141.3603

8. To start with our Task 3, we will use seaborn to check for any null values in our data set, below you can see that there is no null value in our data set.

TASK #3: PERFORM EXPLORATORY DATA ANALYSIS:

[6]:
```
# check if there are any Null values
sns.heatmap(insurance_df.isnull(), yticklabels = False, cbar = False, cmap="Blues")
```

[6]: <Axes: >

[7]:
```
# check if there are any Null values
insurance_df.isnull().sum()
```

[7]:
```
age         0
sex         0
bmi         0
children    0
smoker      0
region      0
charges     0
dtype: int64
```

9. After that we check for the data frame information and you can see that is has total 1338 entries 7 columns and what are the data types this data frame has.

[8]:
```
# Check the dataframe info

insurance_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   age       1338 non-null   int64  
 1   sex       1338 non-null   object 
 2   bmi       1338 non-null   float64
 3   children  1338 non-null   int64  
 4   smoker    1338 non-null   object 
 5   region    1338 non-null   object 
 6   charges   1338 non-null   float64
dtypes: float64(2), int64(2), object(3)
memory usage: 73.3+ KB
```

10. So, the command we used groups the data by region and calculates the average (mean) for only the numeric information, like the charges and body mass index (BMI). Basically, we are saying that "Group my data by region and find the average of the numeric values like charges and BMI for each region."

[10]:
```
# Grouping by region to see any relationship between region and charges
# Seems like south east region has the highest charges and body mass index
df_region = insurance_df.groupby(by='region').mean(numeric_only=True)
df_region
```

[10]:

region	age	bmi	children	charges
northeast	39.268519	29.173503	1.046296	13406.384516
northwest	39.196923	29.199785	1.147692	12417.575374
southeast	38.939560	33.355989	1.049451	14735.411438
southwest	39.455385	30.596615	1.141538	12346.937377

11. After that we check for the unique values in the ‘sex’ column and we only have male and female. Then we convert the categorical variable to numerical and then display the values from our data set.

[11]:
```
# Check unique values in the 'sex' column
insurance_df['sex'].unique()
```

[11]:
```
array(['female', 'male'], dtype=object)
```

[12]:
```
# convert categorical variable to numerical

insurance_df['sex'] = insurance_df['sex'].apply(lambda x: 0 if x == 'female' else 1)
```

[13]:
```
insurance_df.head()
```

[13]:

	age	sex	bmi	children	smoker	region	charges
0	19	0	27.900	0	yes	southwest	16884.92400
1	18	1	33.770	1	no	southeast	1725.55230
2	28	1	33.000	3	no	southeast	4449.46200
3	33	1	22.705	0	no	northwest	21984.47061
4	32	1	28.880	0	no	northwest	3866.85520

12. Again, we did the same thing with the ‘smoker’ column. We check for the unique values in it and then we converted the categorical variable to numerical as you can see below.

[14]:
```
# Check the unique values in the 'smoker' column
insurance_df['smoker'].unique()
```

[14]:
```
array(['yes', 'no'], dtype=object)
```

[15]:
```
# Convert categorical variable to numerical

insurance_df['smoker'] = insurance_df['smoker'].apply(lambda x: 0 if x == 'no' else 1)
```

[16]:
```
insurance_df.head()
```

[16]:

	age	sex	bmi	children	smoker	region	charges
0	19	0	27.900	0	1	southwest	16884.92400
1	18	1	33.770	1	0	southeast	1725.55230
2	28	1	33.000	3	0	southeast	4449.46200
3	33	1	22.705	0	0	northwest	21984.47061
4	32	1	28.880	0	0	northwest	3866.85520

13. Then we check the unique values in region column and this command transforms the 'region' column, which contains text (categorical data), into numeric values. This is important for many machine learning models, which can't handle text directly.

[17]:
```
# Check unique values in 'region' column
insurance_df['region'].unique()
```

[17]:
```
array(['southwest', 'southeast', 'northwest', 'northeast'], dtype=object)
```

[18]:
```
region_dummies = pd.get_dummies(insurance_df['region'], drop_first = True)
```

[19]:
```
region_dummies
```

[19]:

	northwest	southeast	southwest
0	False	False	True
1	False	True	False
2	False	True	False
3	True	False	False
4	True	False	False

14. The first command merges the existing insurance dataset with a new set of columns that represent the different regions in a simplified way (these new columns are created earlier and are called "region dummies"). The second command removes the original "region" column from the dataset, since it has now been replaced by these new region-related columns.

[20]:
```
insurance_df = pd.concat([insurance_df, region_dummies], axis = 1)
```

[21]:
```
insurance_df.head()
```

[21]:

	age	sex	bmi	children	smoker	region	charges	northwest	southeast	southwest
0	19	0	27.900	0	1	southwest	16884.92400	False	False	True
1	18	1	33.770	1	0	southeast	1725.55230	False	True	False
2	28	1	33.000	3	0	southeast	4449.46200	False	True	False
3	33	1	22.705	0	0	northwest	21984.47061	True	False	False
4	32	1	28.880	0	0	northwest	3866.85520	True	False	False

[22]:
```
# Let's drop the original 'region' column
insurance_df.drop(['region'], axis = 1, inplace = True)
```

[23]:
```
insurance_df.head()
```

[23]:

	age	sex	bmi	children	smoker	charges	northwest	southeast	southwest
0	19	0	27.900	0	1	16884.92400	False	False	True
1	18	1	33.770	1	0	1725.55230	False	True	False
2	28	1	33.000	3	0	4449.46200	False	True	False
3	33	1	22.705	0	0	21984.47061	True	False	False
4	32	1	28.880	0	0	3866.85520	True	False	False

15. Here we have the mean and Standard deviation of our dataset.

MINI CHALLENGE

- Calculate the mean and standard deviation of the age, charges and bmi

[24]:
```
insurance_df.describe()
```

[24]:

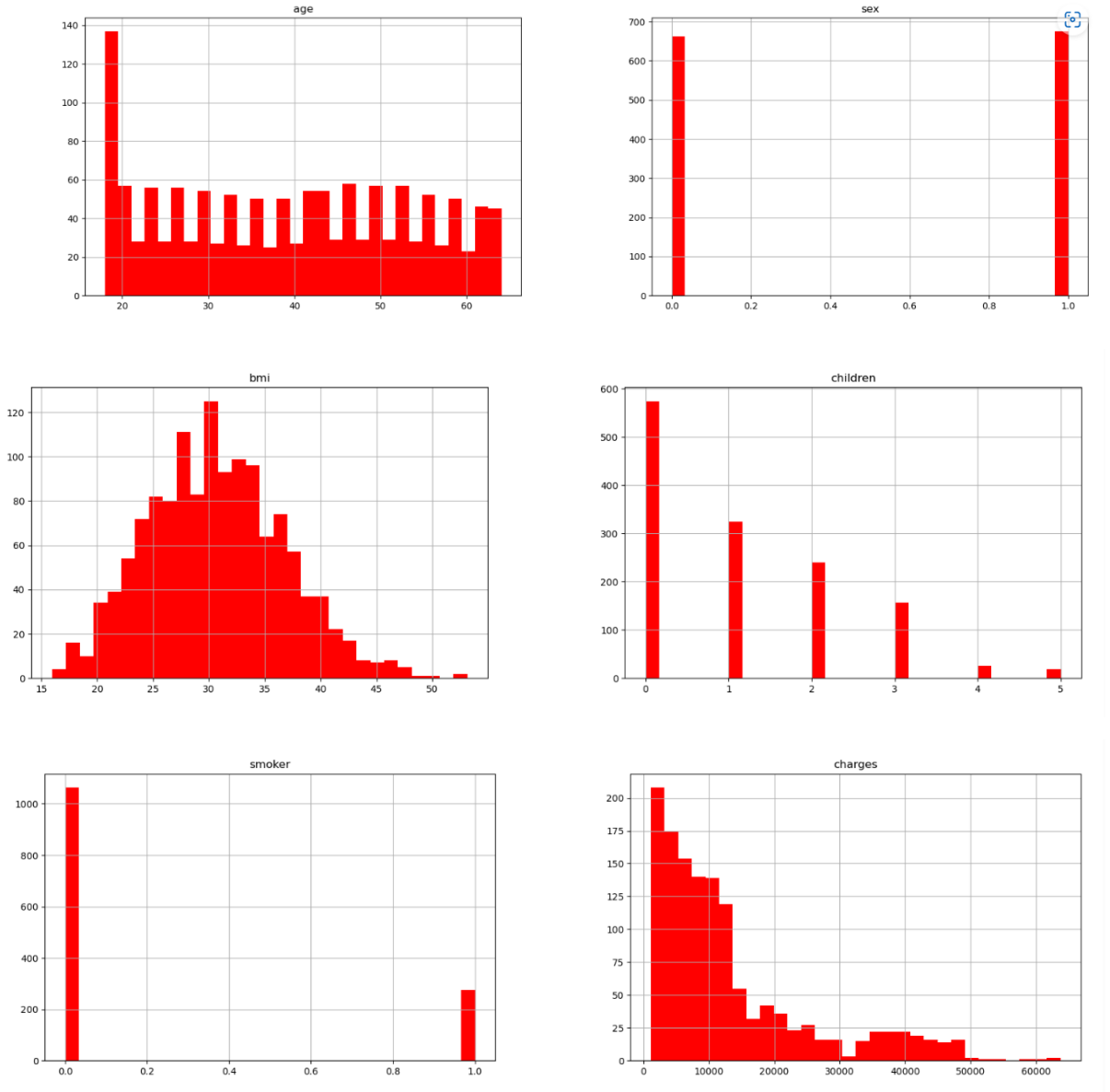
	age	sex	bmi	children	smoker	charges
count	1338.000000	1338.000000	1338.000000	1338.000000	1338.000000	1338.000000
mean	39.207025	0.505232	30.663397	1.094918	0.204783	13270.422265
std	14.049960	0.500160	6.098187	1.205493	0.403694	12110.011237
min	18.000000	0.000000	15.960000	0.000000	0.000000	1121.873900
25%	27.000000	0.000000	26.296250	0.000000	0.000000	4740.287150
50%	39.000000	1.000000	30.400000	1.000000	0.000000	9382.033000
75%	51.000000	1.000000	34.693750	2.000000	0.000000	16639.912515
max	64.000000	1.000000	53.130000	5.000000	1.000000	63770.428010

16. In our task 4 we are going to visualize our data set. This command creates histograms for the columns **"age," "sex," "bmi" (body mass index), "children," "smoker," and "charges"** from the **insurance dataset**. Each histogram will show the distribution of values in these columns. The histograms will have 30 bins (which determine the width of each bar in the graph), and the graphs will be displayed in a large grid with red-colored bars, sized at 20x20 inches.

TASK #4: VISUALIZE DATASET

```
[25]: insurance_df[['age', 'sex', 'bmi', 'children', 'smoker', 'charges']].hist(bins = 30, figsize = (20,20), color = 'r')
```

```
[25]: array([[<Axes: title={'center': 'age'}>, <Axes: title={'center': 'sex'}>],
       [<Axes: title={'center': 'bmi'}>,
        <Axes: title={'center': 'children'}>],
       [<Axes: title={'center': 'smoker'}>,
        <Axes: title={'center': 'charges'}>]], dtype=object)
```



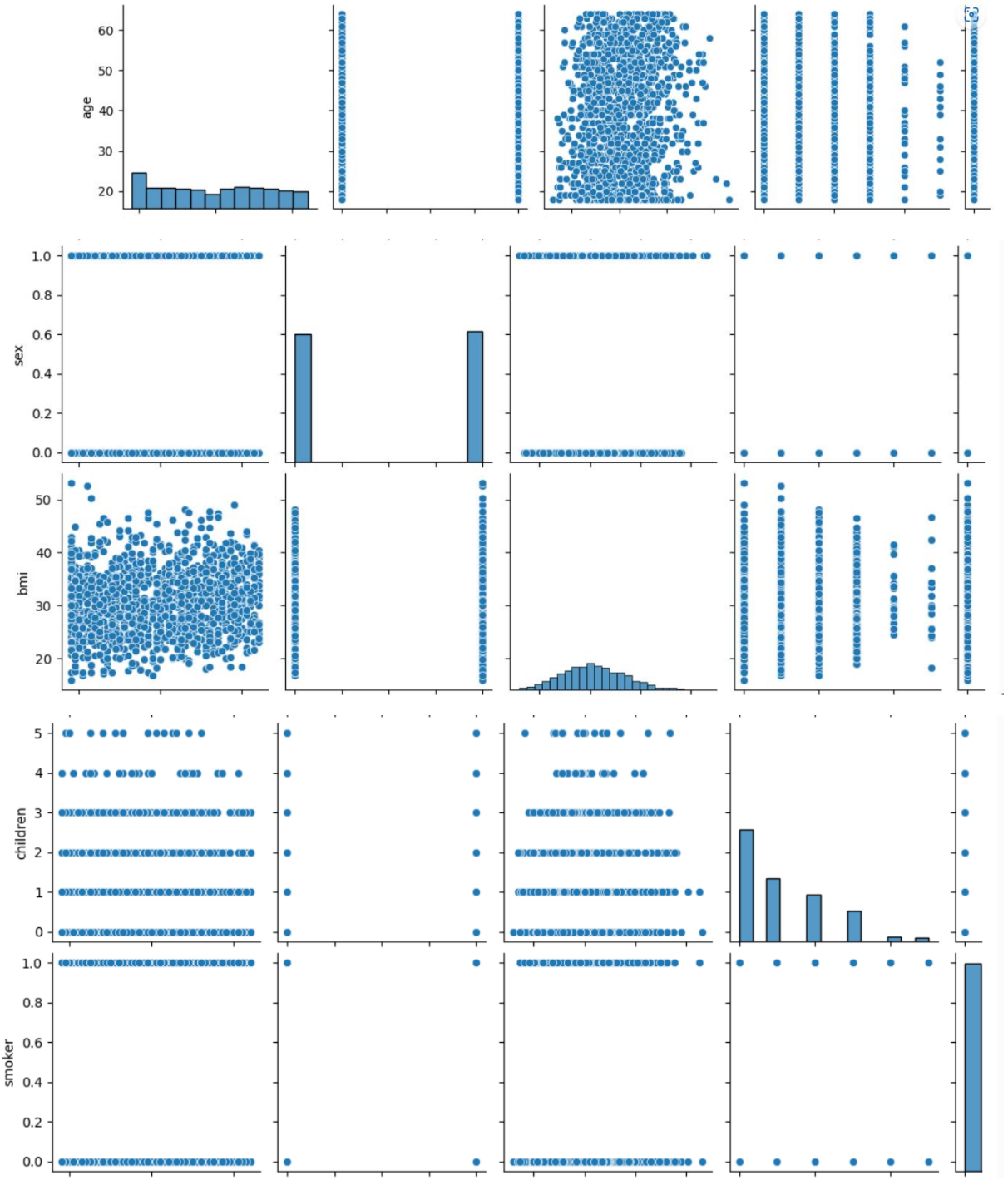
17. This command generates a pairplot using Seaborn to visualize relationships between pairs of variables in the insurance dataset. A pairplot displays scatter plots for each pair of numerical variables and histograms (or density plots) for the individual variables

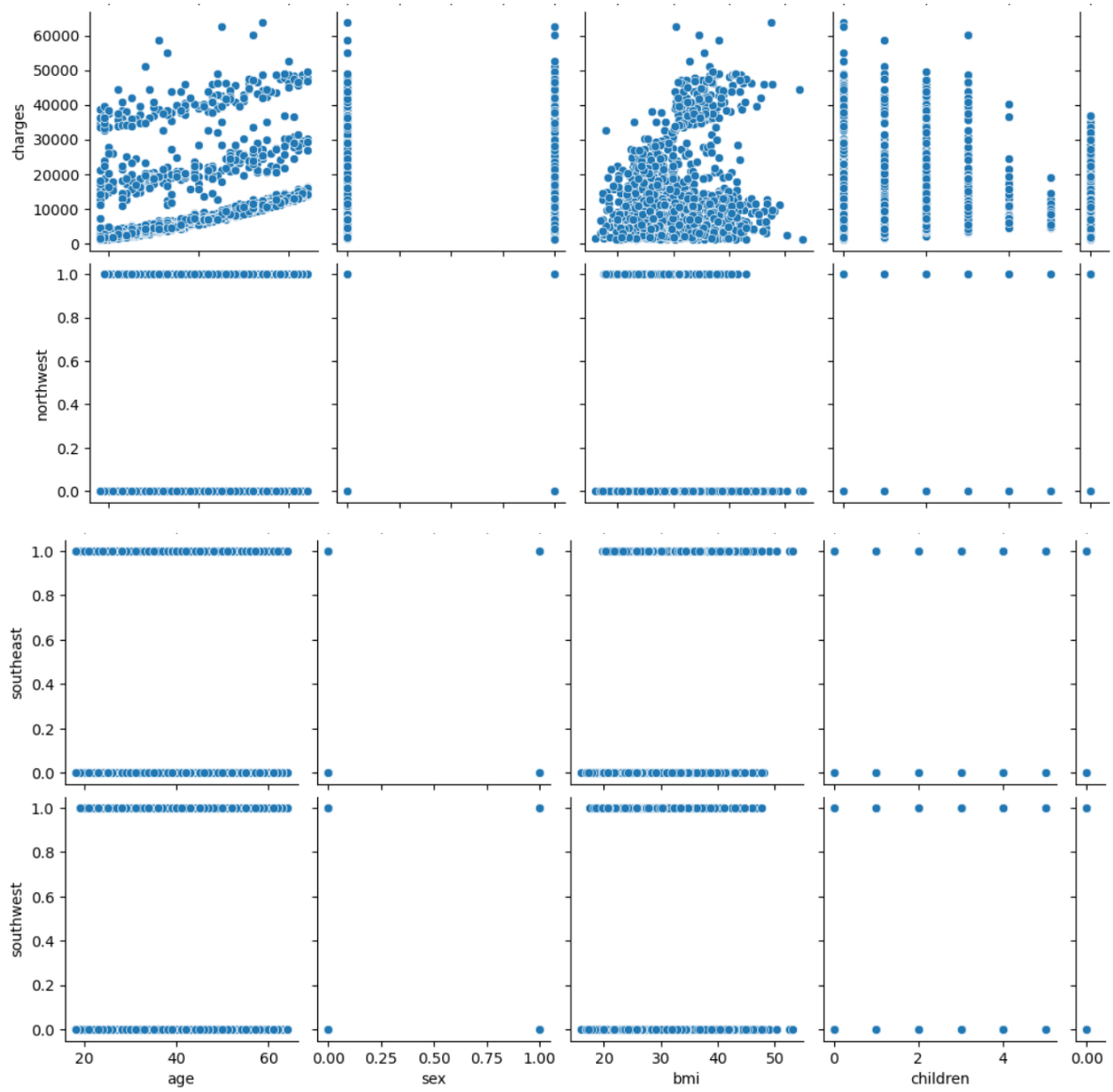
along the diagonal. This helps in understanding correlations and distributions between the features in the dataset.

```
[26]: # plot pairplot

      sns.pairplot(insurance_df)
```

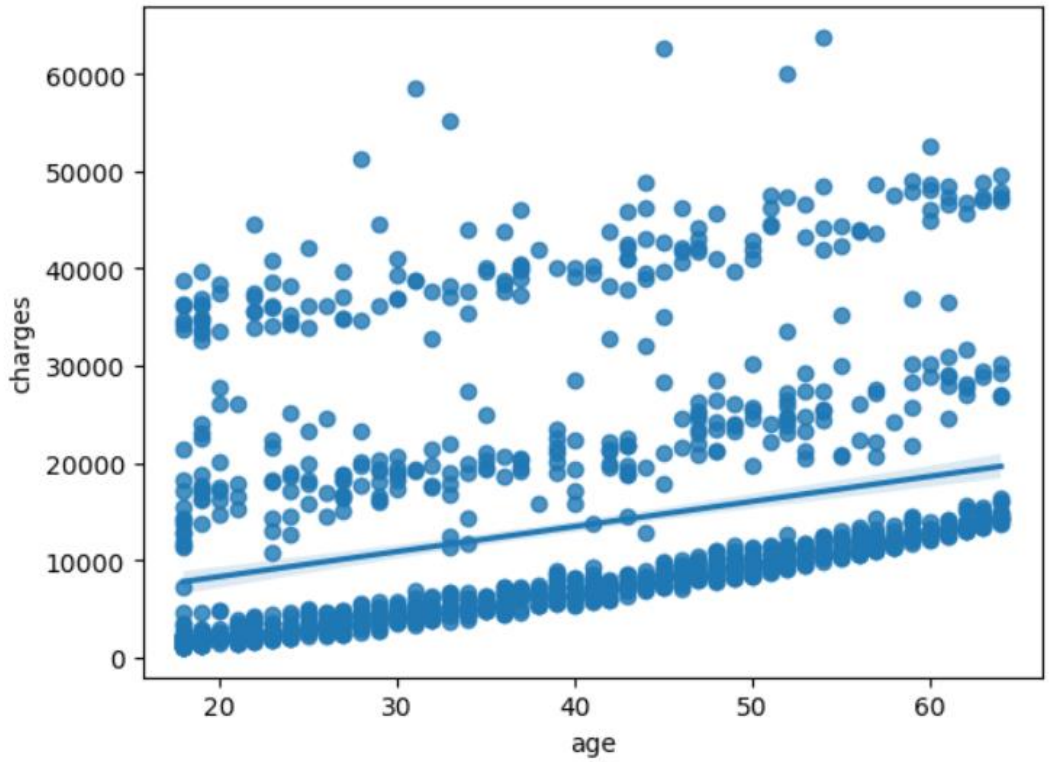
```
[26]: <seaborn.axisgrid.PairGrid at 0x7f15d4dec310>
```





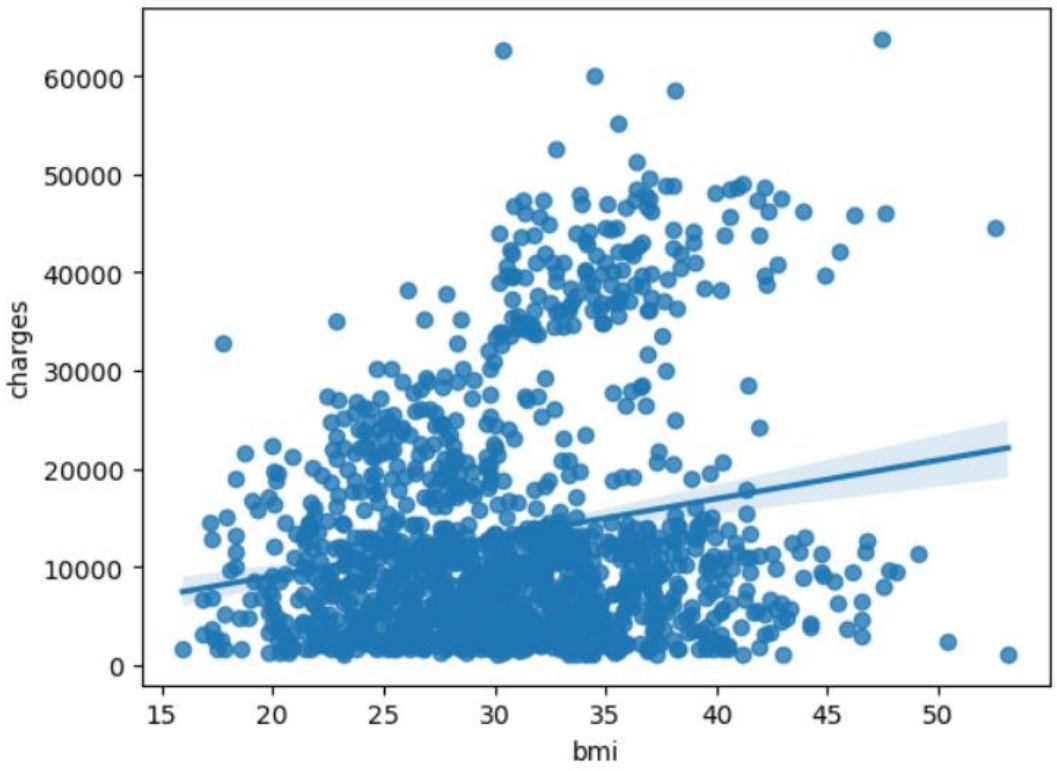
18. The below command creates a regression plot (regplot) using Seaborn to visualize the relationship between the "age" and "charges" columns in the insurance dataset. The plot will display individual data points as well as a trend line that shows the linear relationship between age and insurance charges. The plt.show() command ensures that the plot is displayed.

[27]:
```
sns.regplot(x = 'age', y = 'charges', data = insurance_df)
plt.show()
```



19. Then we create a regression plot for BMI and charges.

[28]:
```
sns.regplot(x = 'bmi', y = 'charges', data = insurance_df)
plt.show()
```



20. Here first we draw the correlation matrix and then plot the heat map.

MINI CHALLENGE

- Calculate and plot the correlation matrix
- Which feature has the most positive correlation with charges?

[30]:
```
corr = insurance_df.corr()
corr
```

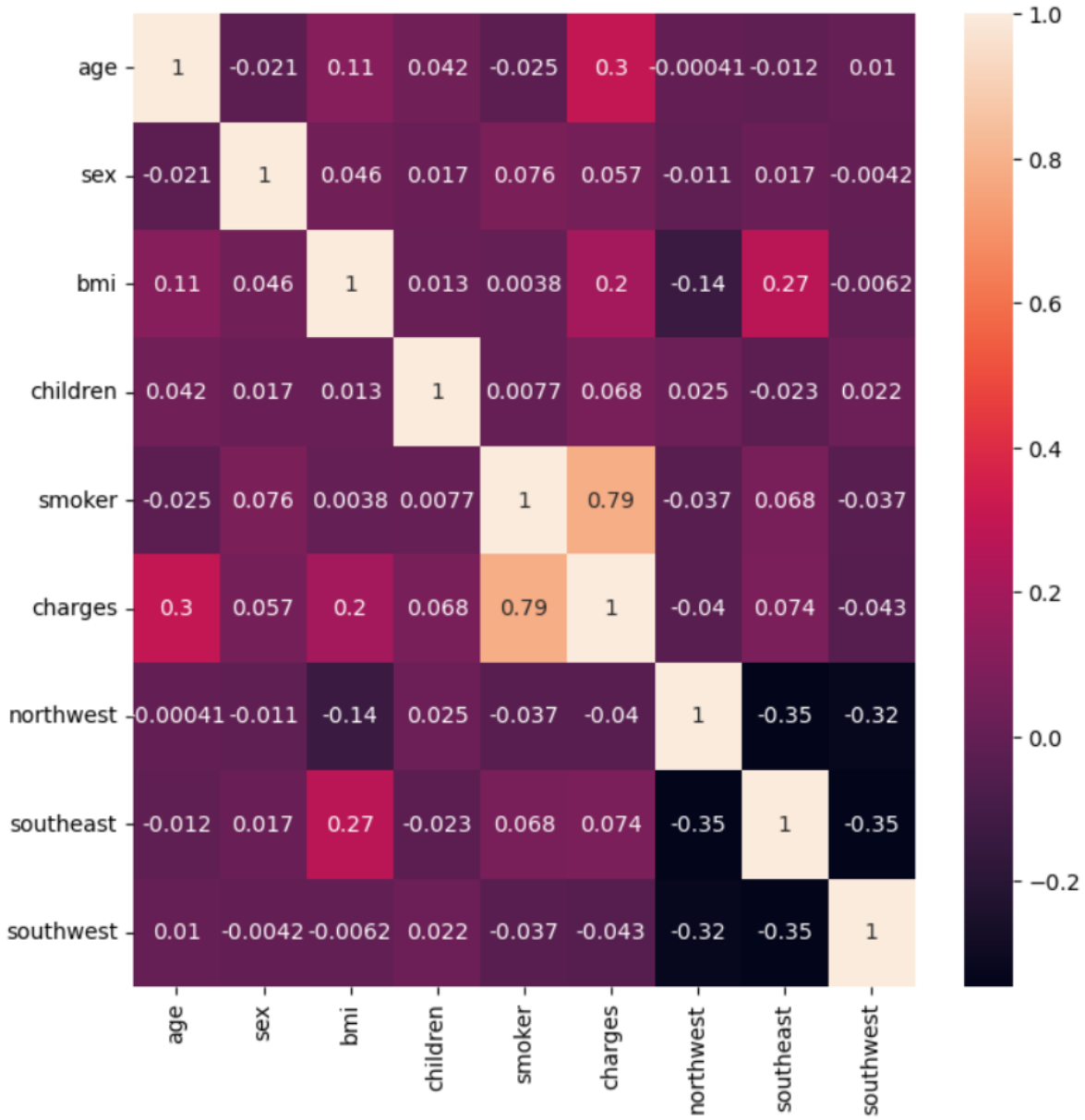
[30]:

	age	sex	bmi	children	smoker	charges	northwest	southeast	southwest
age	1.000000	-0.020856	0.109272	0.042469	-0.025019	0.299008	-0.000407	-0.011642	0.010016
sex	-0.020856	1.000000	0.046371	0.017163	0.076185	0.057292	-0.011156	0.017117	-0.004184
bmi	0.109272	0.046371	1.000000	0.012759	0.003750	0.198341	-0.135996	0.270025	-0.006205
children	0.042469	0.017163	0.012759	1.000000	0.007673	0.067998	0.024806	-0.023066	0.021914
smoker	-0.025019	0.076185	0.003750	0.007673	1.000000	0.787251	-0.036945	0.068498	-0.036945
charges	0.299008	0.057292	0.198341	0.067998	0.787251	1.000000	-0.039905	0.073982	-0.043210
northwest	-0.000407	-0.011156	-0.135996	0.024806	-0.036945	-0.039905	1.000000	-0.346265	-0.320829
southeast	-0.011642	0.017117	0.270025	-0.023066	0.068498	0.073982	-0.346265	1.000000	-0.346265
southwest	0.010016	-0.004184	-0.006205	0.021914	-0.036945	-0.043210	-0.320829	-0.346265	1.000000

[32]:
```
# smoker and age have positive correlations with charges

plt.figure(figsize = (8,8))
sns.heatmap(corr, annot = True)
```

[32]: <Axes: >



21. Now in our Task 5 we create training and testing dataset. First, we display all the columns from our dataset. Then we choose x variable to display our all the input columns and y to display only the output column which is our charges column.

TASK #5: CREATE TRAINING AND TESTING DATASET

[33]:
```
insurance_df.columns
```

[33]:
```
Index(['age', 'sex', 'bmi', 'children', 'smoker', 'charges', 'northwest',
       'southeast', 'southwest'],
      dtype='object')
```

[34]:
```
X = insurance_df.drop(columns =['charges'])
y = insurance_df['charges']
```

[35]:
```
X
```

[35]:

	age	sex	bmi	children	smoker	northwest	southeast	southwest
0	19	0	27.900	0	1	False	False	True
1	18	1	33.770	1	0	False	True	False
2	28	1	33.000	3	0	False	True	False
3	33	1	22.705	0	0	True	False	False
4	32	1	28.880	0	0	True	False	False
...	...	...	...	...	...	...	...	...

22. Here you can see that the y variable displaying the output column which is the charges column. Then we displayed the shape of our x variable and y variable.

```
[36]: y
```

```
[36]: 0       16884.92400
      1        1725.55230
      2        4449.46200
      3       21984.47061
      4        3866.85520
                  ...
      1333    10600.54830
      1334     2205.98080
      1335     1629.83350
      1336     2007.94500
      1337    29141.36030
      Name: charges, Length: 1338, dtype: float64
```

```
[37]: X.shape
```

```
[37]: (1338, 8)
```

```
[38]: y.shape
```

```
[38]: (1338,)
```

23. Then we converted the data type to float 32 for both the variables and we used scikit learn library to train and split the data into two parts. Then the scaling process helps normalize the data, which can enhance the performance and convergence speed of machine learning algorithms.

```
[39]: X = np.array(X).astype('float32')
      y = np.array(y).astype('float32')

[40]: y = y.reshape(-1,1)

[41]: # Only take the numerical variables and scale them
      X

[41]: array([[19.  ,  0.  , 27.9 , ...,  0.  ,  0.  ,  1.  ],
             [18.  ,  1.  , 33.77, ...,  0.  ,  1.  ,  0.  ],
             [28.  ,  1.  , 33.  , ...,  0.  ,  1.  ,  0.  ],
             ...,
             [18.  ,  0.  , 36.85, ...,  0.  ,  1.  ,  0.  ],
             [21.  ,  0.  , 25.8 , ...,  0.  ,  0.  ,  1.  ],
             [61.  ,  0.  , 29.07, ...,  1.  ,  0.  ,  0.  ]], dtype=float32)

[42]: from sklearn.model_selection import train_test_split

      X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=42)

[43]: #scaling the data before feeding the model
      from sklearn.preprocessing import StandardScaler, MinMaxScaler

      scaler_x = StandardScaler()
      X_train = scaler_x.fit_transform(X_train)
      X_test = scaler_x.transform(X_test)

      scaler_y = StandardScaler()
      y_train = scaler_y.fit_transform(y_train)
      y_test = scaler_y.transform(y_test)
```

24. After that we split the data into 20% testing and 80% training so what our data gets shuffled.

MINI CHALLENGE

- Split the data into 20% Testing and 80% Training
- Double check that the split was successful by getting the shape of both the training and testing datasets

```
[44]: from sklearn.model_selection import train_test_split
      X_train, X_test, y_test, y_test = train_test_split(X, y, test_size = 0.2)

[45]: X_train.shape

[45]: (1070, 8)

[46]: X_test.shape

[46]: (268, 8)
```

25. Now in our task 6 we train and test a linear regression model in scikit learn library. In the first command imported some necessary tools from scikit learn library and then we performed the fit or train on it.

TASK #6: TRAIN AND TEST A LINEAR REGRESSION MODEL IN SK-LEARN (NOTE THAT SAGEMAKER BUILT-IN ALGORITHMS ARE NOT USED HERE)

[47]:
```
# using linear regression model
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, accuracy_score

regresssion_model_sklearn = LinearRegression()
regresssion_model_sklearn.fit(X_train, y_train)
```

[47]:
```
LinearRegression
LinearRegression()
```

26. Then we check for the accuracy of the model and used the y predict on the model. After that we check for the samples and we got 268 total samples in our data.

[48]:
```
regresssion_model_sklearn_accuracy = regresssion_model_sklearn.score(X_test, y_test)
regresssion_model_sklearn_accuracy
```

[48]:
```
-1.1889013935103607
```

[49]:
```
y_predict = regresssion_model_sklearn.predict(X_test)
```

[50]:
```
y_predict_orig = scaler_y.inverse_transform(y_predict)
y_test_orig = scaler_y.inverse_transform(y_test)
```

[51]:
```
k = X_test.shape[1]
n = len(X_test)
n
```

[51]:
```
268
```

27. The below code snippet calculates evaluation metrics for the regression model's performance. **RMSE** provides an indication of the average distance between predicted and actual values, with lower values indicating better model performance. **MSE** is the average of the squared differences between predicted and actual values, also indicating model performance, where lower values are preferable.

[52]:
```
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
from math import sqrt

RMSE = float(format(np.sqrt(mean_squared_error(y_test_orig, y_predict_orig)),'.3f'))
MSE = mean_squared_error(y_test_orig, y_predict_orig)
```

28. Here we calculated the mean absolute error, R2 and adjusted R2.

MINI CHALLENGE

- calculate the mean absolute error, R2 and adjusted R2

[55]:
```
MAE = mean_absolute_error(y_test_orig, y_predict_orig)
r2 = r2_score(y_test_orig, y_predict_orig)
adj_r2 = 1-(1-r2) * (n-1)/(n-k-1)
```

[56]:
```
print('RMSE =',RMSE, '\nMSE =',MSE, '\nMAE =',MAE, '\nR2 =', r2, '\nAdjusted R2 =', adj_r2)
```

```
RMSE = 203825824.0
MSE = 4.1544965e+16
MAE = 150216530.0
R2 = -1.1889014147979244
Adjusted R2 = -1.2565122693090576
```

29. In our task 7 we train a linear learner model using the SageMaker. This code sets up a SageMaker session to work with machine learning models. It identifies the S3 bucket where data will be stored and retrieves the IAM role (permissions) for SageMaker to use AWS services on your behalf.

TASK #7: TRAIN A LINEAR LEARNER MODEL USING SAGEMAKER

[57]:
```
# Boto3 is the Amazon Web Services (AWS) Software Development Kit (SDK) for Python
# Boto3 allows Python developer to write software that makes use of services like Amazon S3 and Amazon EC2

import sagemaker
import boto3
from sagemaker import Session

# Let's create a Sagemaker session
sagemaker_session = sagemaker.Session()
bucket = Session().default_bucket()
prefix = 'linear_learner' # prefix is the subfolder within the bucket.

# Let's get the execution role for the notebook instance.
# This is the IAM role that you created when you created your notebook instance. You pass the role to the training job.
# Note that AWS Identity and Access Management (IAM) role that Amazon SageMaker can assume to perform tasks on your behalf (for example, reading tr
role = sagemaker.get_execution_role()
print(role)
```

```
sagemaker.config INFO - Not applying SDK defaults from location: /etc/xdg/sagemaker/config.yaml
sagemaker.config INFO - Not applying SDK defaults from location: /home/sagemaker-user/.config/sagemaker/config.yaml
arn:aws:iam::878893308172:role/service-role/AmazonSageMaker-ExecutionRole-20241023T110678
```

30. Here we checked shape for x train and y train.

[58]:
```
X_train.shape
```

[58]:
```
(1070, 8)
```

[59]:
```
y_train.shape
```

[59]:
```
(1070, 1)
```

31. This code takes our training data (stored in NumPy arrays). Converts it into a special format (RecordIO) that AWS SageMaker's Linear Learner algorithm requires. Stores this converted data in a temporary memory buffer. The seek(0) part ensures that when you later read the data from this buffer, you start from the very beginning.

[61]:
```
import io # The io module allows for dealing with various types of I/O (text I/O, binary I/O and raw I/O).
import numpy as np
import sagemaker.amazon.common as smac # sagemaker common libary

# Code below converts the data in numpy array format to RecordIO format
# This is the format required by Sagemaker Linear Learner

buf = io.BytesIO() # create an in-memory byte array (buf is a buffer I will be writing to)
smac.write_numpy_to_dense_tensor(buf, X_train, y_train.reshape(-1))
buf.seek(0)
# When you write to in-memory byte arrays, it increments 1 every time you write to it
# Let's reset that back to zero
```

[61]:
```
0
```

32. This code uploads the training data (which is stored in memory in RecordIO format) to an S3 bucket. The file is saved in a folder named train under a path like linear_learner/train/linear-train-data. Finally, it prints out the S3 URL where the data is now stored. This is a key step in preparing your data for training in SageMaker, as the model training job will need to access this data in S3.

[62]:
```
import os

# Code to upload RecordIO data to S3

# Key refers to the name of the file
key = 'linear-train-data'

# The following code uploads the data in record-io format to S3 bucket to be accessed later for training
boto3.resource('s3').Bucket(bucket).Object(os.path.join(prefix, 'train', key)).upload_fileobj(buf)

# Let's print out the training data location in s3
s3_train_data = 's3://{}/{}/train/{}'.format(bucket, prefix, key)
print('uploaded training data location: {}'.format(s3_train_data))
```

```
uploaded training data location: s3://sagemaker-us-east-1-878893308172/linear_learner/train/linear-train-data
```

[63]:
```
# create an output placeholder in S3 bucket to store the linear learner output

output_location = 's3://{}/{}/output'.format(bucket, prefix)
print('Training artifacts will be uploaded to: {}'.format(output_location))
```

```
Training artifacts will be uploaded to: s3://sagemaker-us-east-1-878893308172/linear_learner/output
```

33. The below code gets the path to a pre-built SageMaker environment (container image) that has everything needed to train a model using SageMaker's Linear Learner algorithm. You don't have to worry about specifying the region or setting up the algorithm yourself, it's all handled automatically by the get_image_uri function.

[64]:
```
# This code is used to get the training container of sagemaker built-in algorithms
# all we have to do is to specify the name of the algorithm, that we want to use

# Let's obtain a reference to the linearLearner container image
# Note that all regression models are named estimators
# You don't have to specify (hardcode) the region, get_image_uri will get the current region name using boto3.Session


from sagemaker.amazon.amazon_estimator import get_image_uri

container = get_image_uri(boto3.Session().region_name, 'linear-learner')
```

```
The method get_image_uri has been renamed in sagemaker>=2.
See: https://sagemaker.readthedocs.io/en/stable/v2.html for details.
```

34. This code creates a machine learning model using Amazon SageMaker's Linear Learner. You specify the type of instance, number of instances, and where the model should store its output. Then, you define parameters for training, like how the model should handle the data (batch size, number of iterations) and the type of model you want (regression). Finally, the model is trained using data stored in an S3 bucket.

```
[*]: # We have pass in the container, the type of instance that we would like to use for training
     # output path and sagemaker session into the Estimator.
     # We can also specify how many instances we would like to use for training

     linear = sagemaker.estimator.Estimator(container,
                                            role,
                                            train_instance_count = 1,
                                            train_instance_type = 'ml.c4.xlarge',
                                            output_path = output_location,
                                            sagemaker_session = sagemaker_session)


     # We can tune parameters like the number of features that we are passing in, type of predictor like 'regressor' or 'classifier', mini batch size, e
     # Train 32 different versions of the model and will get the best out of them (built-in parameters optimization!)

     linear.set_hyperparameters(feature_dim = 8,
                                predictor_type = 'regressor',
                                mini_batch_size = 100,
                                epochs = 100,
                                num_models = 32,
                                loss = 'absolute_loss')

     # Now we are ready to pass in the training data from S3 to train the linear learner model

     linear.fit({'train': s3_train_data})

     # Let's see the progress using cloudwatch logs
```

```
2024-10-24 08:08:45 Completed - Training job completed
Training seconds: 155
Billable seconds: 155
```

35. In our task 8 we are going to deploy and test the trained linear learner model.
36. Using the code below our model that we trained earlier is now being deployed on a server (endpoint) in the cloud. This server is where we'll send new data to get predictions (like predicting salaries based on years of experience). SageMaker handles all the infrastructure for us, including setting up the server and managing the model.

TASK #8: DEPLOY AND TEST THE TRAINED LINEAR LEARNER MODEL

```
[66]: # Deploying the model to perform inference

      linear_regressor = linear.deploy(initial_instance_count = 1,
                                       instance_type = 'ml.m4.xlarge')
```

```
INFO:sagemaker:Creating model with name: linear-learner-2024-10-24-08-14-18-056
INFO:sagemaker:Creating endpoint-config with name linear-learner-2024-10-24-08-14-18-056
INFO:sagemaker:Creating endpoint with name linear-learner-2024-10-24-08-14-18-056
----------!
```

37. The model we deployed expects data in a specific format (CSV). This code sets up how to convert your input data into CSV when you send it to the model and how to handle the output data when you receive predictions. It essentially prepares your model to correctly read the input data and format the output predictions.

[67]:
```
from sagemaker.serializers import CSVSerializer
from sagemaker.deserializers import JSONDeserializer

# Content type overrides the data that will be passed to the deployed model, since the deployed model expects data in text/csv format.

# Serializer accepts a single argument, the input data, and returns a sequence of bytes in the specified content type

# Deserializer accepts two arguments, the result data and the response content type, and return a sequence of bytes in the specified content type.

# Reference: https://sagemaker.readthedocs.io/en/stable/predictors.html

linear_regressor.content_type = 'text/csv'
linear_regressor.serializer = CSVSerializer()
linear_regressor.deserializer = JSONDeserializer()
```

38. We're using the deployed model to predict outcomes based on the test data. The model sends back predictions in a structured format (JSON). This code extracts the actual predicted values and puts them into a format (NumPy array) that we can easily work with. Finally, we check how many predictions we received to ensure everything is working correctly.

[68]:
```
# making prediction on the test data

result = linear_regressor.predict(X_test)
```

[69]:
```
result # results are in Json format
```

[69]:
```
{'predictions': [{'score': -0.3003460764884949},
  {'score': -0.23428753018379211},
  {'score': -0.3684966564178467},
  {'score': -0.3058070242404938},
  {'score': -0.28367722034454346},
  {'score': -0.36490440368652344},
  {'score': -0.2372516393661499},
  {'score': -0.32775694131851196},
  {'score': -0.28212040662765503},
  {'score': -0.414196640253067},
  {'score': -0.23722906410694122},
  {'score': -0.2688663601875305},
  {'score': -0.35465121269226074},
  {'score': -0.2708197236061096},
```

[70]:
```
# Since the result is in json format, we access the scores by iterating through the scores in the predictions

predictions = np.array([r['score'] for r in result['predictions']])
```

[71]:
```
predictions
```

[71]:
```
array([-0.30034608, -0.23428753, -0.36849666, -0.30580702, -0.28367722,
       -0.3649044 , -0.23725164, -0.32775694, -0.28212041, -0.41419664,
       -0.23722906, -0.26886636, -0.35465121, -0.27081972, -0.34636879,
       -0.34131783, -0.3005752 , -0.31892779, -0.24776249, -0.33790106,
       -0.25991297, -0.3526355 , -0.27111754, -0.40955544, -0.36833012,
       -0.32939512, -0.29254285, -0.33499959, -0.26110482, -0.37650409,
       -0.34780285, -0.27295944, -0.31854618, -0.30255479, -0.43415761,
       -0.30321124, -0.32818055, -0.29911101, -0.37659261, -0.39680427,
       -0.26977777, -0.32862362, -0.43093204, -0.3169542 , -0.30541959,
       -0.46136257, -0.37217882, -0.32364228, -0.30558598, -0.36309317,
       -0.29798129, -0.35031283, -0.27148688, -0.34218556, -0.32538891,
       -0.31904131, -0.28388831, -0.28240177, -0.33849511, -0.24229819,
       -0.37542155, -0.38333315, -0.36433214, -0.35176793, -0.37496933,
       -0.24732679, -0.24360028, -0.3180249 , -0.28143737, -0.28832638,
```

39. The code evaluates a regression model's performance by reversing the scaling of predicted and actual values. It reshapes the predictions and test targets into a suitable 2D format and transforms them back to their original scale. It calculates several metrics: RMSE measures the average prediction error, MSE represents the average squared differences, and MAE indicates the average absolute difference. R-squared assesses how well the model explains variability, while Adjusted R-squared adjusts for the number of predictors. Finally, it prints RMSE, MSE, MAE, R-squared, and Adjusted R-squared, providing insights into the model's effectiveness in predictions.

[74]:
```
y_predict_orig = scaler_y.inverse_transform(predictions.reshape(-1, 1))
y_test_orig = scaler_y.inverse_transform(y_test.reshape(-1, 1))
```

[75]:
```
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
from math import sqrt

RMSE = float(format(np.sqrt(mean_squared_error(y_test_orig, y_predict_orig)),'.3f'))
MSE = mean_squared_error(y_test_orig, y_predict_orig)
MAE = mean_absolute_error(y_test_orig, y_predict_orig)
r2 = r2_score(y_test_orig, y_predict_orig)
adj_r2 = 1-(1-r2)*(n-1)/(n-k-1)

print('RMSE =',RMSE, '\nMSE =',MSE, '\nMAE =',MAE, '\nR2 =', r2, '\nAdjusted R2 =', adj_r2)
```

```
RMSE = 203828541.109
MSE = 4.154607417076408e+16
MAE = 150220452.4786469
R2 = -1.188959944005323
Adjusted R2 = -1.2565726063684215
```

40. Once you are done then you need to delete the endpoint.

[76]:
```
# Delete the end-point

linear_regressor.delete_endpoint()
```

```
INFO:sagemaker:Deleting endpoint configuration with name: linear-learner-2024-10-24-08-14-18-056
INFO:sagemaker:Deleting endpoint with name: linear-learner-2024-10-24-08-14-18-056
```