😊 SageMaker Studio for Employee Salary Prediction

Amazon SageMaker Studio is an integrated development environment (IDE) specifically designed for machine learning (ML) workflows. It provides a unified interface for data preparation, model training, tuning, deployment, and management. SageMaker Studio simplifies the process of building, training, and deploying ML models, offering a range of tools that allow data scientists, developers, and ML engineers to collaborate effectively.

Key Features of SageMaker Studio:

1. **End-to-End Machine Learning Development:** SageMaker Studio covers all stages of the machine learning lifecycle, from data processing and experimentation to deployment and monitoring. You can access data, experiment with models, and deploy them within the same environment.
2. **Notebook Integration:** SageMaker Studio provides fully managed Jupyter notebooks, which are auto-scaling and can be shared across team members. These notebooks can easily be connected to various data sources (such as S3) and allow for seamless transition from development to production.
3. **Model Training and Tuning:** SageMaker Studio automates the training and tuning process. It allows users to run experiments, track results, and select the best performing models using SageMaker Experiments. Hyperparameter tuning can be done easily with SageMaker's automated tuning jobs, which efficiently search for the optimal set of hyperparameters.
4. **Collaboration:** Teams can collaborate within SageMaker Studio by sharing notebooks and experiment results. This helps facilitate review and collaboration between data scientists and ML engineers.
5. **Built-in Debugging and Monitoring:** With SageMaker Debugger, you can inspect and debug your models during the training process. It also provides built-in monitoring tools to track the performance of models once deployed.
6. **ML Pipelines:** SageMaker Studio allows you to create automated ML pipelines, making it easier to manage complex ML workflows. These pipelines ensure reproducibility and consistency in your model deployment process.
7. **Model Deployment and Monitoring:** After training, SageMaker Studio simplifies the deployment of models for real-time or batch inference. Deployed models can be monitored for accuracy and performance using built-in tools like SageMaker Model Monitor.
8. **Cost Management:** Since the environment is managed by AWS, you can scale resources on demand. SageMaker Studio provides access to multiple instance types and pricing plans, enabling you to optimize costs based on your workload needs.

Advantages:

- **Scalability:** Easily scale compute resources for training and inference as needed.
- **Managed Environment:** No need to manage infrastructure—AWS takes care of provisioning and scaling.
- **Collaboration-Friendly:** The environment supports team collaboration by sharing notebooks and experiments.
- **Easy Deployment:** It streamlines the transition from experimentation to production.

Use Cases:

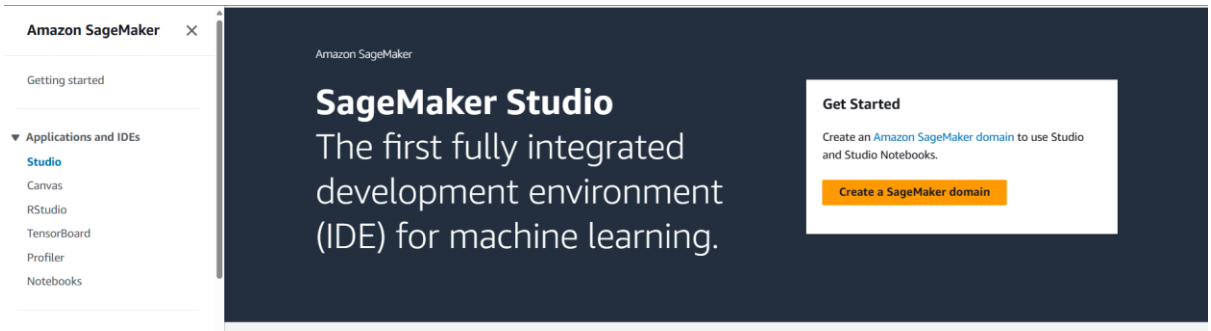
- **Data Preparation:** Use Jupyter notebooks to explore and process datasets.
- **Model Training and Hyperparameter Tuning:** Automate training and tuning processes with a single-click interface.
- **Experiment Tracking:** Use SageMaker Experiments to track and manage your experiments, results, and metrics.
- **Model Deployment and Monitoring:** Deploy models for inference and monitor them in real time for accuracy and performance.

SageMaker Studio is ideal for developers and data scientists who need a comprehensive, scalable, and fully managed platform for end-to-end machine learning development and deployment.

The objective of this lab is to predict the employee salary based on the number of years of experience. The final outcome of this project is to come up with a linear regression model that could accurately predict employee salary based on number of years of experience.

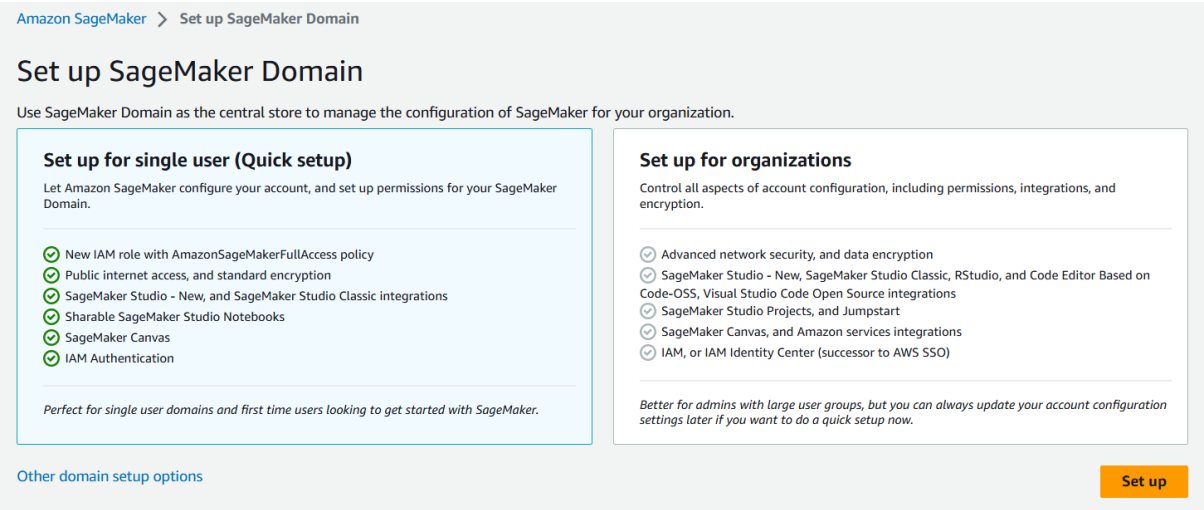
😄 To begin with the Lab:

1. Login to your AWS Console, search for sage maker studio and navigate to it.
2. Here you need to expand applications and IDEs and open Studio. Then click on Create a SageMaker domain.

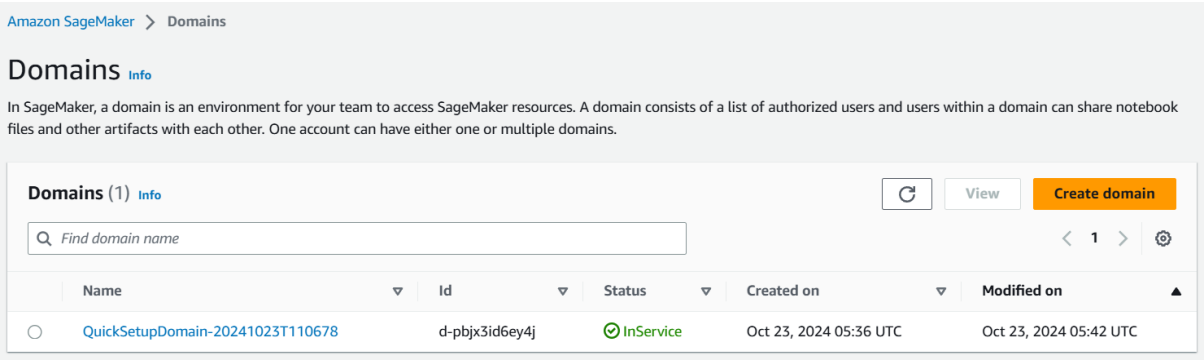


3. Then to set up SageMaker Domain choose Quick set up for single user and click on Set Up.

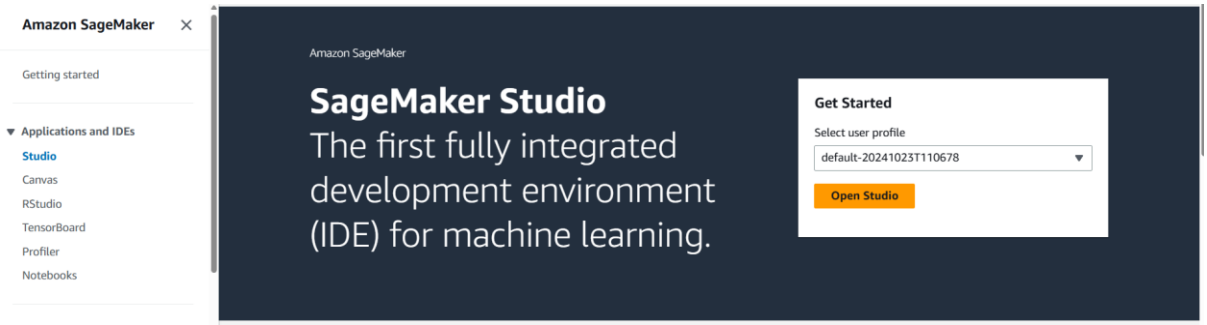
4. Now you need to wait until it gets ready for you to use.



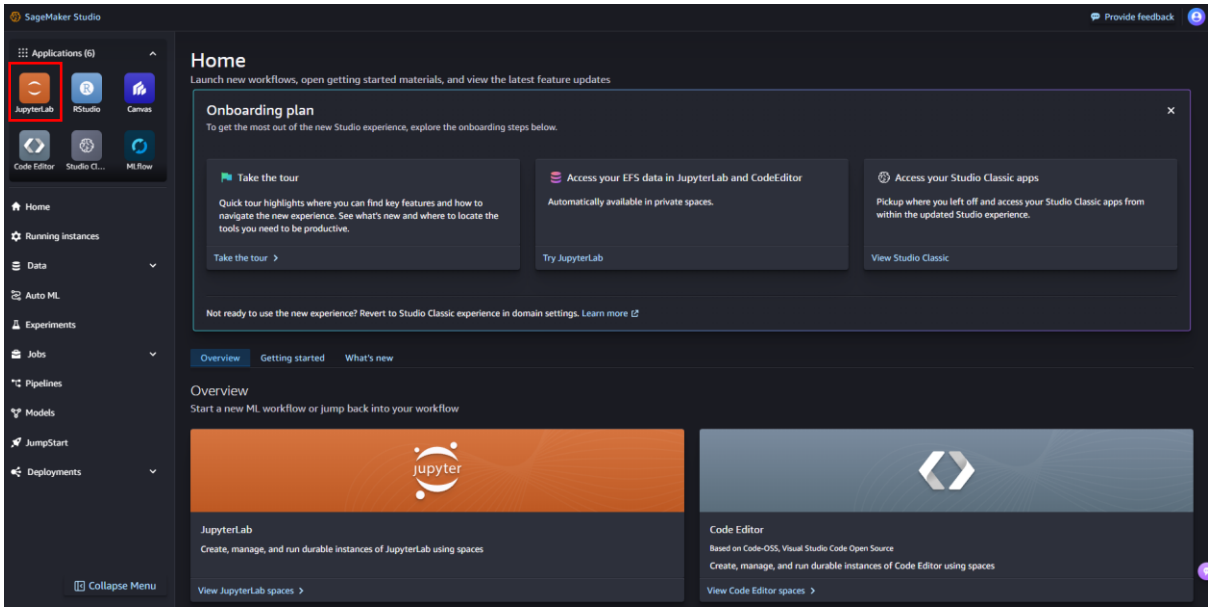
5. Here you can see that our domain is ready and it is in service.



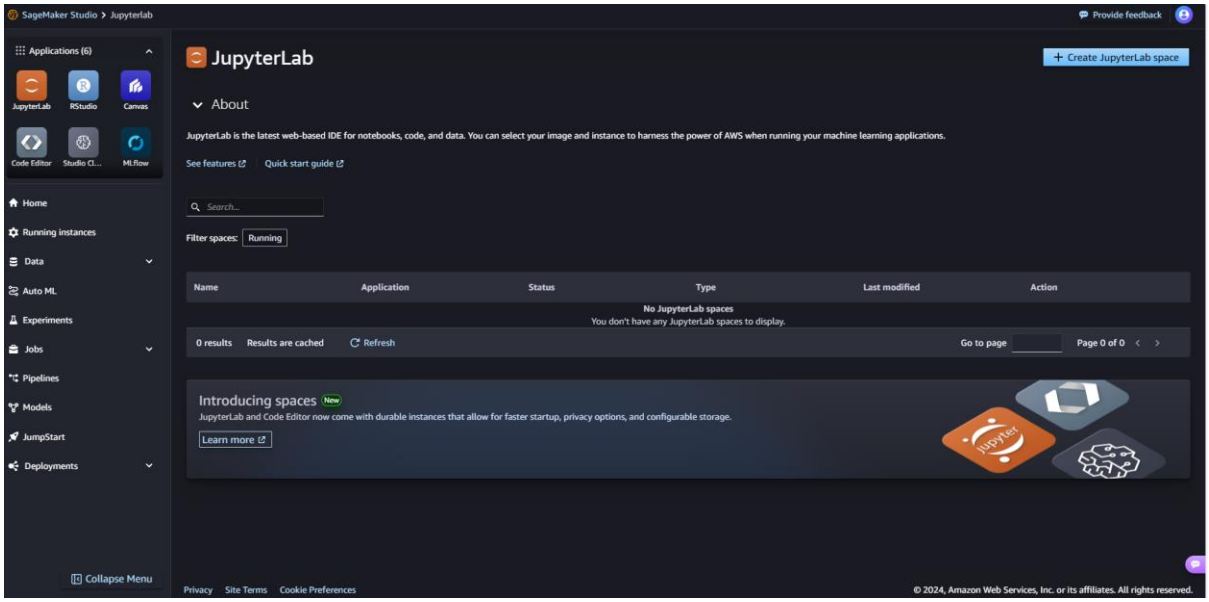
6. Now if you come back to studio, you will see that this time you have the option to open the studio. Click on it and it will direct you to a new page.



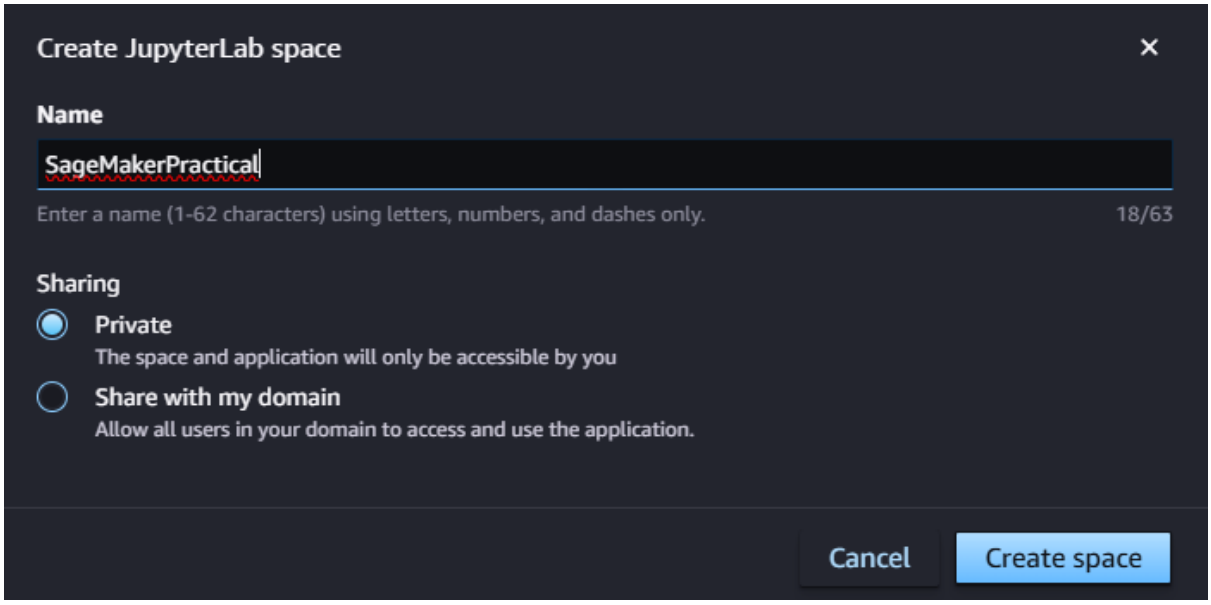
7. Then you will on the home page of this SageMaker Studio home page. You need to click on Jupyter Lab.



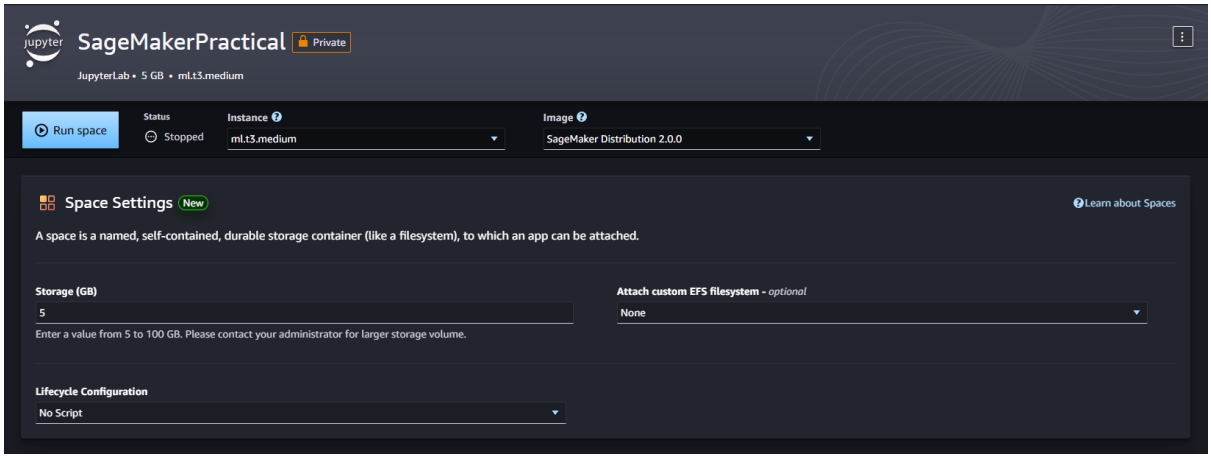
8. You will see that you can now create a Jupyter Lab space. So, click on it.



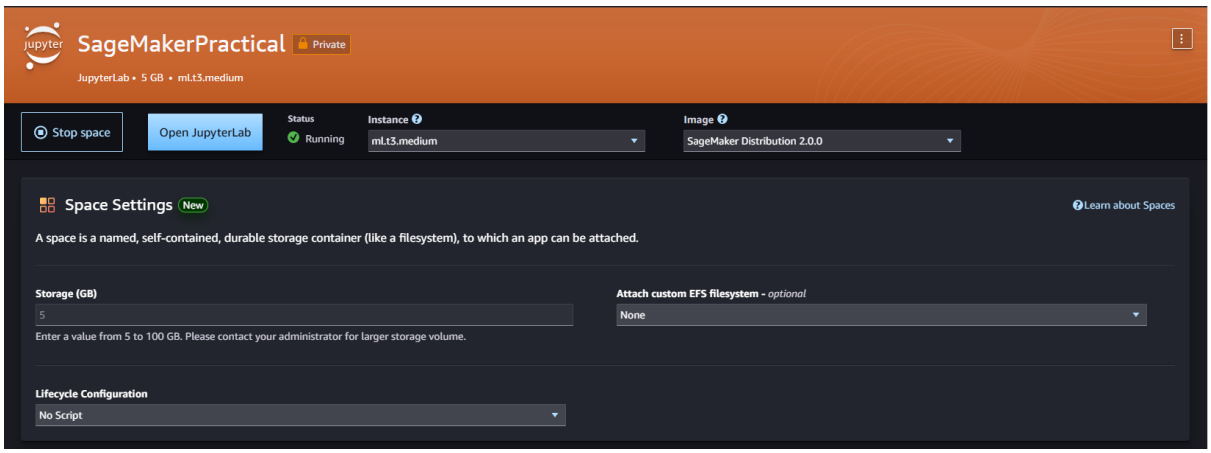
9. Then it will ask you to give it a name and choose Private sharing, click on Create space.



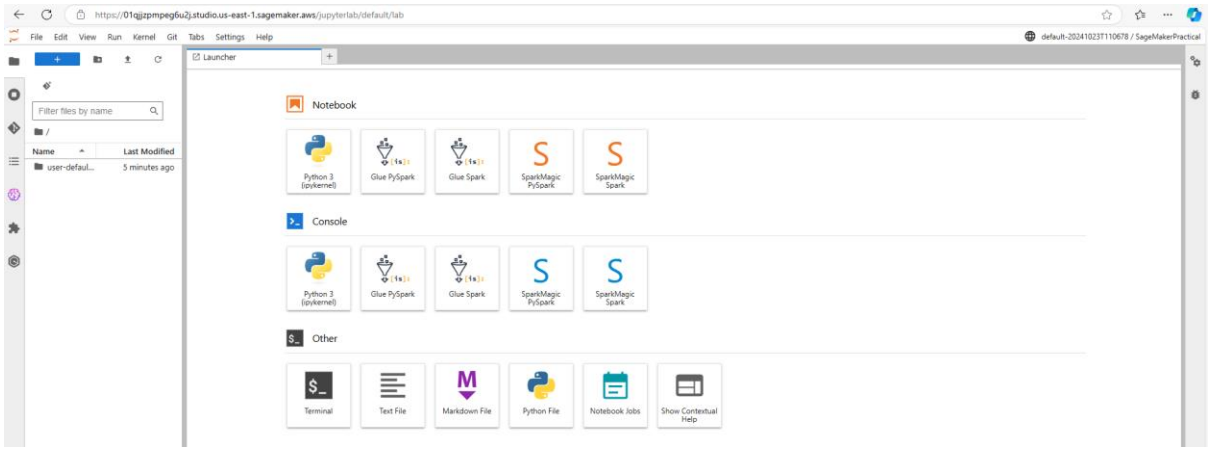
10. After that you need to click on Run space button, keep the instance and image to default settings.



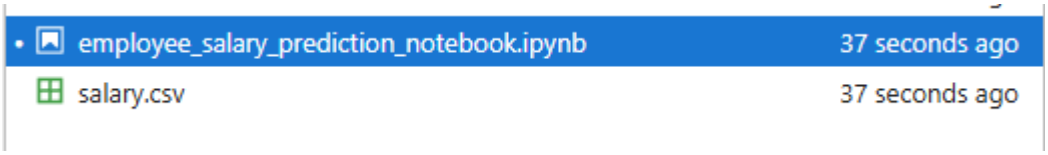
11. Once your space is created successfully then click on Open Jupyter Lab.



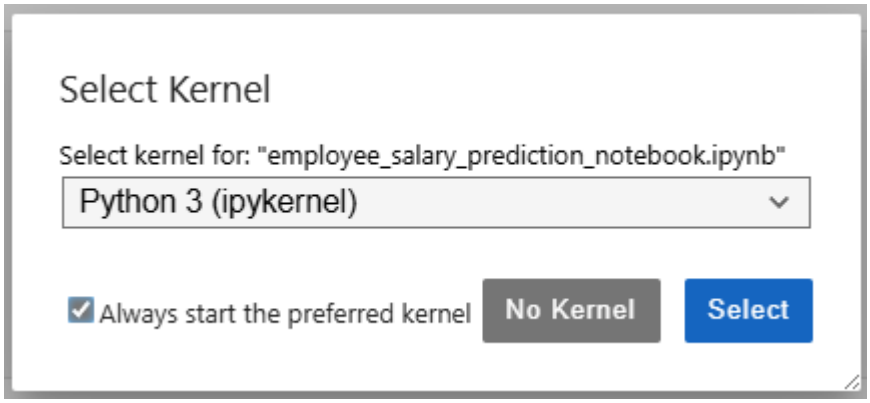
12. Below you can see that you are on the homepage of Jupyter Lab hosted on AWS.



13. Now you need to click on the upload symbol to upload the files. Below you can see the two files that you need to upload for this lab. First the employee salary prediction python notebook and the salary CSV file.



14. Once you have uploaded the files then you need to open the employee salary prediction notebook and choose the Python 3 kernel and click on select.



15. First, we are going to import the necessary libraries and datasets as you can below in the code and run the code.

TASK #1: UNDERSTAND THE PROBLEM STATEMENT

- The objective of this case study is to predict the employee salary based on the number of years of experience.
- In simple linear regression, we predict the value of one variable Y based on another variable X.
- X is called the independent variable and Y is called the dependant variable.
- Why simple? Because it examines relationship between two variables only.
- Why linear? when the independent variable increases (or decreases), the dependent variable increases (or decreases) in a linear fashion.

TASK #2: IMPORT LIBRARIES AND DATASETS

```
[3]: # install seaborn library
!pip install seaborn
!pip install tensorflow
import tensorflow as tf
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

16. After that in the next step we want pandas to read my CSV file, so we ran the command shown below in line number 4 and then we displayed the entries from our dataset.

```
[4]: # read the csv file
salary_df = pd.read_csv('salary.csv')
```

```
[5]: salary_df
```

[5]:

	YearsExperience	Salary
0	1.1	39343
1	1.3	46205
2	1.5	37731
3	2.0	43525
4	2.2	39891
5	2.9	56642
6	3.0	60150

17. Then as a part of mini challenge we used the head and tail methods to display the first and last 7 rows of our data frame and then we found out the maximum salary in our data set.

MINI CHALLENGE

- Use head and tail methods to print the first and last 7 rows of the dataframe
- Try to find the maximum salary value in the dataframe

[6]:
```
salary_df.head(7)
```

[6]:

	YearsExperience	Salary
0	1.1	39343
1	1.3	46205
2	1.5	37731
3	2.0	43525
4	2.2	39891
5	2.9	56642
6	3.0	60150

[7]:
```
salary_df.tail(7)
```

[7]:

	YearsExperience	Salary
28	10.3	122391
29	10.5	121872
30	11.2	127345
31	11.5	126756
32	12.3	128765
33	12.9	135675
34	13.5	139465

[8]:
```
salary_df['Salary'].max()
```

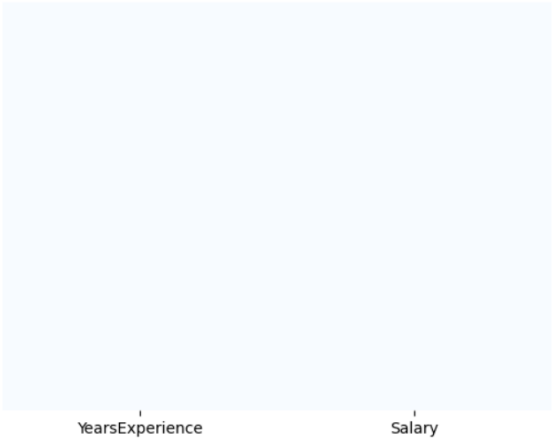
[8]:
```
139465
```

18. As a part of our Task 3 we are going to perform EDA (Exploratory Data Analysis) and Visualization on our dataset.
19. So, the first command we used here is to check if any null values exist in our data set and here you can see that there se none null value. If there was any null value in our data then in the graph you will be able to see the dots.

TASK #3: PERFORM EXPLORATORY DATA ANALYSIS AND VISUALIZATION

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

```
[9]: <Axes: >
```



20. Then we check for the information in our data set you can see that our data set contains only 35 entries and there are two columns and what data type they are.

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

      salary_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 35 entries, 0 to 34
Data columns (total 2 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   YearsExperience  35 non-null     float64
 1   Salary           35 non-null     int64  
dtypes: float64(1), int64(1)
memory usage: 692.0 bytes
```

21. Here we used describe to get the statistical summary of our data frame as you can see below. So, this command give us the mean, standard deviation, max and min value.

[11]:
```
# Statistical summary of the dataframe

salary_df.describe()
```

[11]:

	YearsExperience	Salary
count	35.000000	35.000000
mean	6.308571	83945.600000
std	3.618610	32162.673003
min	1.100000	37731.000000
25%	3.450000	57019.000000
50%	5.300000	81363.000000
75%	9.250000	113223.500000
max	13.500000	139465.000000

22. Below we found out what is the maximum and minimum salaries of the employees.

MINI CHALLENGE

- What are the number of years of experience corresponding to employees with minimum and maximim salaries?

[12]:
```
max = salary_df[ salary_df['Salary'] == salary_df['Salary'].max()]
max
```

[12]:

	YearsExperience	Salary
34	13.5	139465

[13]:
```
min = salary_df[ salary_df['Salary'] == salary_df['Salary'].min()]
min
```

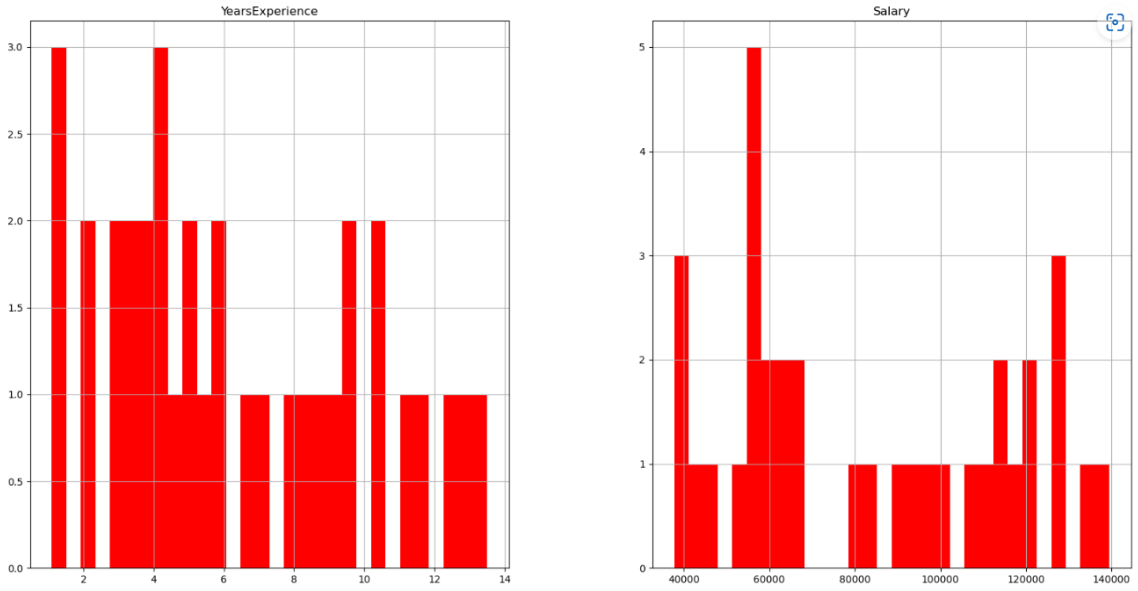
[13]:

	YearsExperience	Salary
2	1.5	37731

23. Using the below command we plotted a histogram for our data frame as you can see below.

```
[14]: salary_df.hist(bins = 30, figsize = (20,10), color = 'r')
```

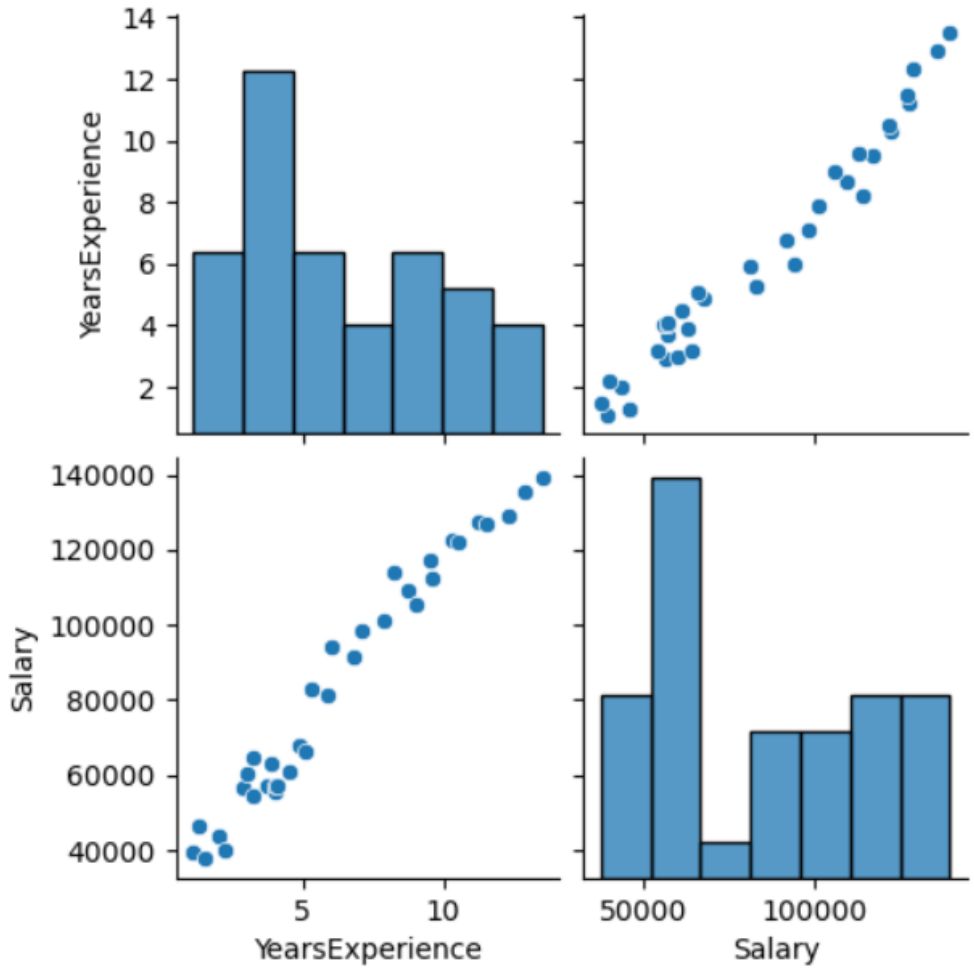
```
[14]: array([[<Axes: title={'center': 'YearsExperience'}>,
              <Axes: title={'center': 'Salary'}>]], dtype=object)
```



24. Then we used seaborn to create a pair plot for our data frame. Below you can see that we have the pots ready.

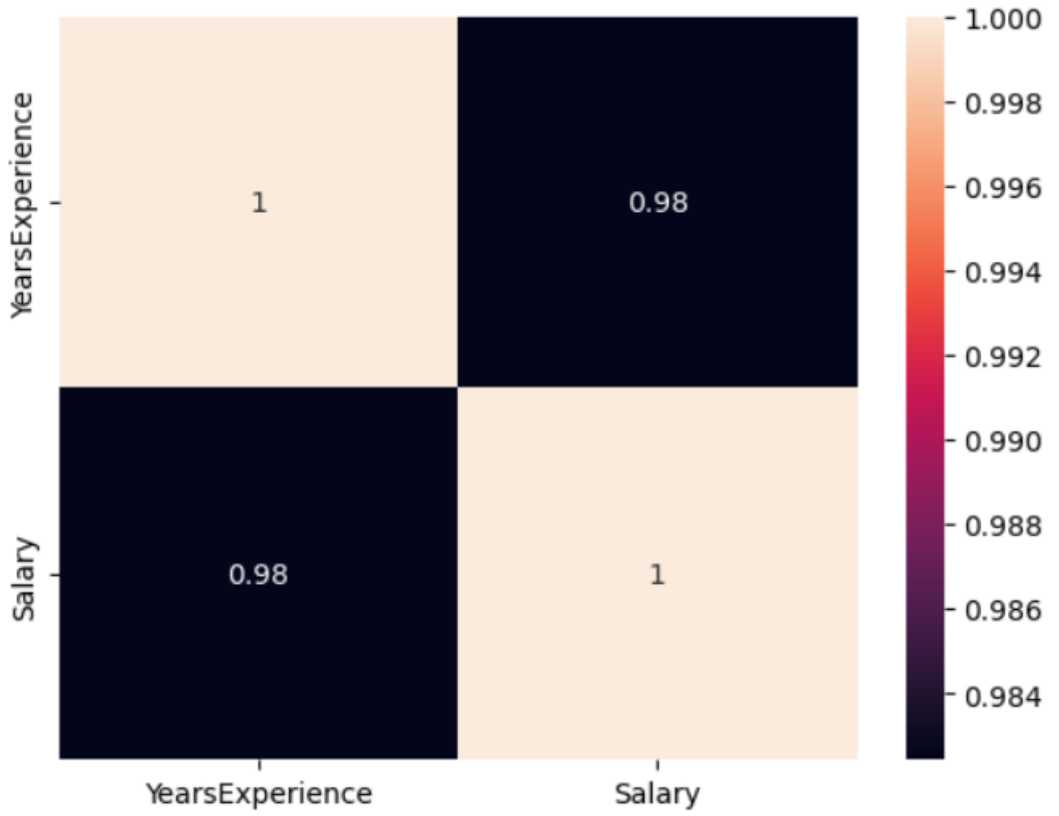
[15]:
```
# plot pairplot

sns.pairplot(salary_df)
```

[15]:
```
<seaborn.axisgrid.PairGrid at 0x7f2f112d9b90>
```



25. So, the code below is creating a **visual chart** (heatmap) and corelation matrix that shows how strongly the different variables (columns) in our dataset are related to each other. For example, it is telling us if two factors (like experience and salary) tend to go up together, are unrelated, or if one goes up while the other goes down.

[16]:
```
corr_matrix = salary_df.corr()
sns.heatmap(corr_matrix, annot = True)
plt.show()
```



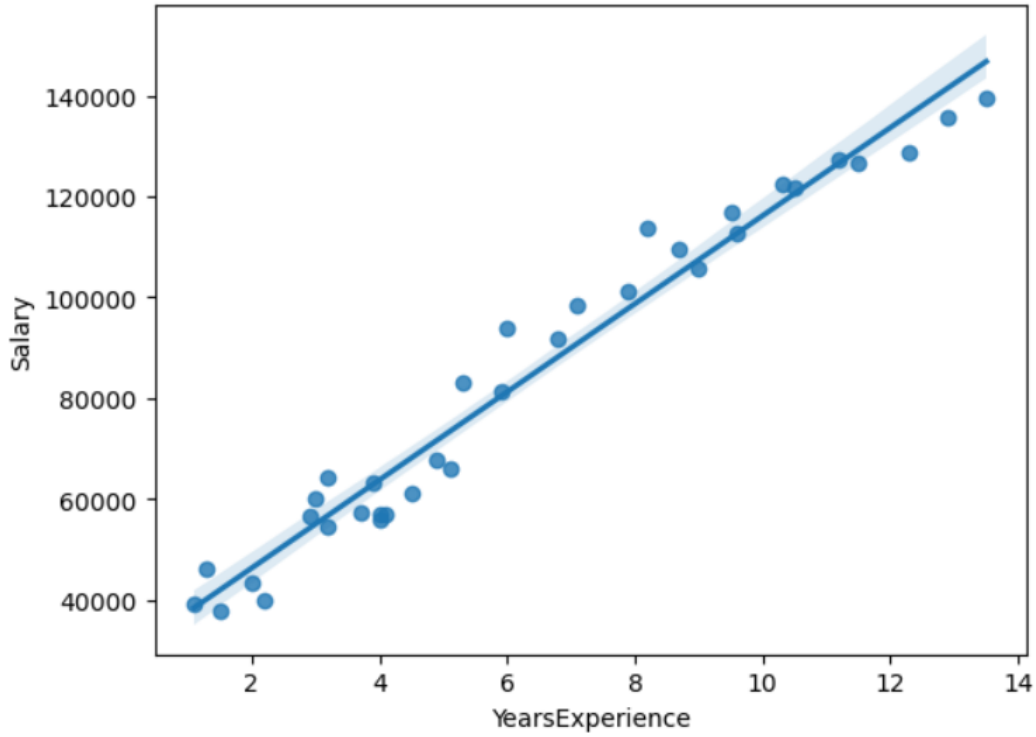
26. Here we are using regplot in Seaborn to obtain a straight line fit between salary and year of experience of the employee from our data frame.

MINI CHALLENGE

- Use regplot in Seaborn to obtain a straight line fit between "salary" and "years of experience"

[18]:
```
sns.regplot(x = 'YearsExperience', y = 'Salary', data = salary_df)
```

[18]:
```
<Axes: xlabel='YearsExperience', ylabel='Salary'>
```



27. Now in our Task 4 we will do the training and testing on our Data set. First, we defined x as the years of experience and y as the salary column from our data frame as you can see below. Also remember that x is our independent variable and y is our dependent variable.
28. Then if we enter x and run the cell then you will get all the information about years of experience column and if we enter y and run the cell then you will get all the information about salary column.

TASK #4: CREATE TRAINING AND TESTING DATASET

[19]:
```
X = salary_df[['YearsExperience']]
y = salary_df[['Salary']]
```

[20]:
```
X
```

[20]:

	YearsExperience
0	1.1
1	1.3
2	1.5
3	2.0
4	2.2
5	2.9

[21]:
```
y
```

[21]:

	Salary
0	39343
1	46205
2	37731
3	43525
4	39891
5	56642
6	60150
7	54445
8	64445
9	57189

29. After that we checked the shape of our data and ran the x shape and y shape command. As you can see below, we got 35 rows and 1 column in both of them.

```
[22]: X.shape

[22]: (35, 1)

[23]: y.shape

[23]: (35, 1)
```

30. Here we converted our x and y variables to float 32.

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

31. Then we viewed our x column and you can see that is of float 32 data type now.

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

```
[ 5.3],
[ 5.9],
[ 6. ],
[ 6.8],
[ 7.1],
[ 7.9],
[ 8.2],
[ 8.7],
[ 9. ],
[ 9.5],
[ 9.6],
[10.3],
[10.5],
[11.2],
[11.5],
[12.3],
[12.9],
[13.5]], dtype=float32)
```

32. After that we slit our data into test and train. So, what will happen is that our data will be trained first and then it will be tested.
    This code is splitting your dataset into two groups:

- **Training set**: This group (80% of the data) is used to train your model, so it can learn from the data.
- **Testing set**: This group (20% of the data) is used to check how well the model performs on new, unseen data.

```
[26]: # split the data into test and train sets
      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)
```

33. The change you will notice after slitting your data is that you data has now been shuffled.

```
[27]: X_test

[27]: array([[3. ],
             [3.2],
             [1.1],
             [4. ],
             [8.2],
             [8.7],
             [6. ]], dtype=float32)

[28]: X_train

[28]: array([[ 2. ],
             [ 7.1],
             [11.5],
             [ 3.9],
             [ 5.9],
             [ 4.5],
             [12.3],
             [ 7.9],
             [ 9. ],
```

34. So, in our Task number 5 we will train a linear regression model using SK-Learn (Scikit Learn) library.
35. In this code we are setting up a linear regression model, which tries to find a straight line that best fits the data (i.e., explains the relationship between the input and output). It then trains the model using the training data so that it can make predictions about the output values based on new input data.

TASK #5: TRAIN A LINEAR REGRESSION MODEL IN SK-LEARN (NOTE THAT SAGEMAKER BUILT-IN ALGORITHMS ARE NOT USED HERE)

[30]:
```
# using linear regression model
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, accuracy_score

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

[30]:
```
LinearRegression
LinearRegression()
```

36. In the line number 31, by running this code we are checking the accuracy of our x test and y test. You can see that we are able to get the 87% accuracy on our data.
37. In the line number 32, this code is printing out two key parameters of your trained linear regression model, the **coefficients** (m) tell you how strongly each feature (input variable) influences the target variable. The **intercept** (b) is the value your model predicts when all input variables are 0.

Note: For example, in the equation of a line y=mx+b, the coef_ is m (slope), and the intercept_ is b (the point where the line crosses the y-axis).

[31]:
```
regresssion_model_sklearn_accuracy = regresssion_model_sklearn.score(X_test, y_test)
regresssion_model_sklearn_accuracy
```

[31]:
```
0.8768443879629642
```

[32]:
```
print('Linear Model Coefficient (m): ', regresssion_model_sklearn.coef_)
print('Linear Model Coefficient (b): ', regresssion_model_sklearn.intercept_)
```

```
Linear Model Coefficient (m):  [[8751.173]]
Linear Model Coefficient (b):  [27436.148]
```

38. Then we set the fit intercept to False and we can see that our coefficient b is forced to be zero (0).

MINI CHALLENGE

- Retrain the model while setting the fit_intercept = False, what do you notice?

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

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

```
[33]: LinearRegression
      LinearRegression(fit_intercept=False)
```

```
[34]: print('Linear Model Coefficient (m): ', regresssion_model_sklearn.coef_)
      print('Linear Model Coefficient (b): ', regresssion_model_sklearn.intercept_)

      Linear Model Coefficient (m):  [[11906.033]]
      Linear Model Coefficient (b):  0.0
```

39. Once you have done the mini challenge and observe the changes then change the values back to true.
40. In our Task number 6 we are going to evaluate our trained model performance. So, here we are going to apply the predict method on our trained model and feed it in my x test which is my testing data and that will generate y predict. Below you can see the y prediction.

TASK #6: EVALUATE TRAINED MODEL PERFORMANCE (NOTE THAT SAGEMAKER BUILT-IN ALGORITHMS ARE NOT USED HERE)

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

```
[42]: y_predict
```

```
[42]: array([[ 53689.668],
             [ 55439.902],
             [ 37062.438],
             [ 62440.84 ],
             [ 99195.766],
             [103571.35 ],
             [ 79943.19 ]], dtype=float32)
```

41. This code creates a graph that shows:
    - Gray dots: The actual data points (the number of years of experience and the corresponding salary).
    - Red line: The prediction made by the linear regression model, which tries to fit a straight line through the data to represent the relationship between experience and salary.

    This visualization helps you see how well the regression line fits the actual data.

```
[43]: plt.scatter(X_train, y_train, color = 'gray')
      plt.plot(X_train, regresssion_model_sklearn.predict(X_train), color = 'red')
      plt.ylabel('Salary')
      plt.xlabel('Number of Years of Experience')
      plt.title('Salary vs. Years of Experience')
```

```
[43]: Text(0.5, 1.0, 'Salary vs. Years of Experience')
```

MINI CHALLENGE

- Use the trained model, obtain the salary corresponding to eployees who have years of experience = 5

```
[44]: num_years_exp = [[5]]
```

```
[46]: salary = regresssion_model_sklearn.predict(num_years_exp)
      salary
```

```
[46]: array([[71192.01269531]])
```

42. In our Task number 7 we are going to 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

[48]:
```
# 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
role = sagemaker.get_execution_role()
print(role)
```

```
arn:aws:iam::878893308172:role/service-role/AmazonSageMaker-ExecutionRole-20241023T110678
```

43. Then we move ahead and check the shape of our data as you can see below.

[49]:
```
X_train.shape
```

[49]:
```
(28, 1)
```

[50]:
```
y_train = y_train[:,0]
```

[51]:
```
y_train.shape
```

[51]:
```
(28,)
```

44. 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.

[52]:
```
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)
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
```

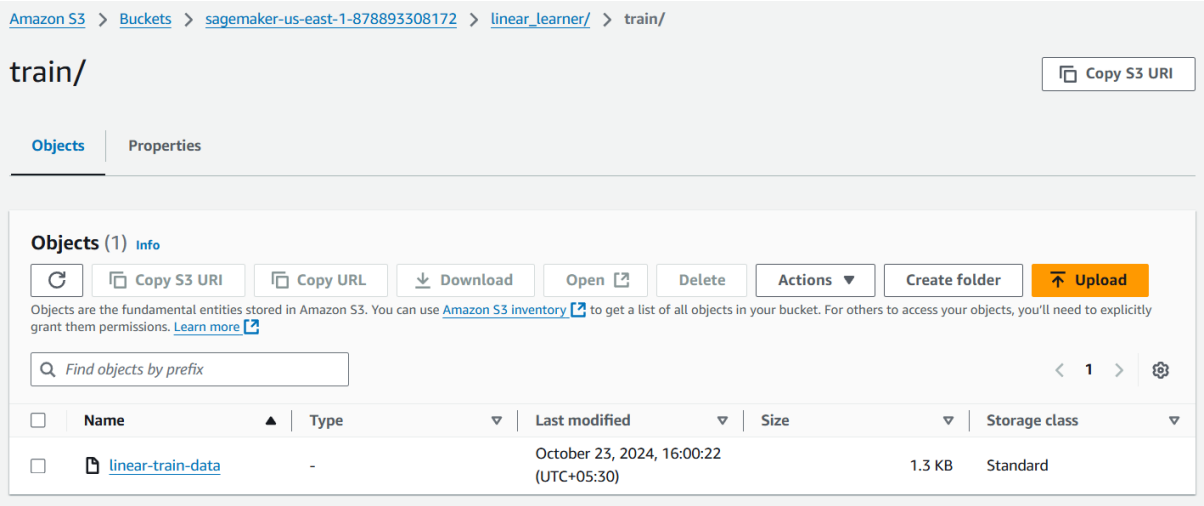
[52]:
```
0
```

45. 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.

```
[53]: 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
```

46. Also, if you go to the S3 bucket, here you will see your linear train data.



47. After that we look for the shape of our data.

```
[54]: X_test.shape

[54]: (7, 1)

[55]: y_test.shape

[55]: (7, 1)
```

```
[56]: # Make sure that the target label is a vector
      y_test = y_test[:,0]

[57]: y_test.shape

[57]: (7,)

[58]: y_test

[58]: array([ 60150.,  64445.,  39343.,  55794., 113812., 109431.,  93940.],
            dtype=float32)
```

48. This code creates a location in our **S3 bucket** where the output (trained model and other results) from SageMaker's **Linear Learner** training job will be stored. It then prints the path of that location for your reference.

```
[61]: # 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
```

49. 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.

```
[62]: # 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.
```

50. 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.

[63]:
```
# 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
# sagemaker_session = sagemaker.Session()

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
# Train 32 different versions of the model and will get the best out of them (built-in parameters optimization!)

linear.set_hyperparameters(feature_dim = 1,
                           predictor_type = 'regressor',
                           mini_batch_size = 5,
                           epochs = 5,
                           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
```

```
2024-10-23 10:47:33 Completed - Training job completed
Training seconds: 150
Billable seconds: 150
```

51. Now we are going to deploy and test our trained linear model in our task number 8.
52. 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

[64]:
```
# 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-2024-10-23-10-53-00-942
INFO:sagemaker:Creating endpoint-config with name linear-learner-2024-10-23-10-53-00-942
INFO:sagemaker:Creating endpoint with name linear-learner-2024-10-23-10-53-00-942
--------!
```

53. 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.

[43]:
```
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 typ

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

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

54. We're using the deployed model to predict outcomes (like salaries) 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.

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

      result = linear_regressor.predict(X_test)
```

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

```
[45]: {'predictions': [{'score': 72078.765625},
        {'score': 72078.765625},
        {'score': 79669.265625},
        {'score': 129352.4765625},
        {'score': 123832.1171875},
        {'score': 64488.28125},
        {'score': 53447.5625}]}
```

```
[46]: # 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']])
```

```
[47]: predictions
```

```
[47]: array([ 72078.765625 ,  72078.765625 ,  79669.265625 , 129352.4765625,
             123832.1171875,  64488.28125  ,  53447.5625   ])
```

```
[48]: predictions.shape
```

```
[48]: (7,)
```

55. This code creates a visual representation to compare actual salaries (shown as gray dots) with predicted salaries (shown as a red line). The scatter plot helps us to see how well the model's predictions match the actual data points, making it easier to assess the model's performance. The line indicates the trend that the model predicts based on the input data (years of experience).

[49]:
```
# VISUALIZE TEST SET RESULTS
plt.scatter(X_test, y_test, color = 'gray')
plt.plot(X_test, predictions, color = 'red')
plt.xlabel('Years of Experience (Testing Dataset)')
plt.ylabel('salary')
plt.title('Salary vs. Years of Experience')
```

[49]:
```
Text(0.5, 1.0, 'Salary vs. Years of Experience')
```

56. In the end run this command to delete your endpoint or some unnecessary charges will occur.

[59]:
```
# Delete the end-point

linear_regressor.delete_endpoint()
```