Report Title: -

TensorFlow Deliverables

Submitted By: -

Muhammad Talha Saleem

Submitted To: -

Sir Munir Akhtar

SOCO ENGINEERS

Deliverable Number 1

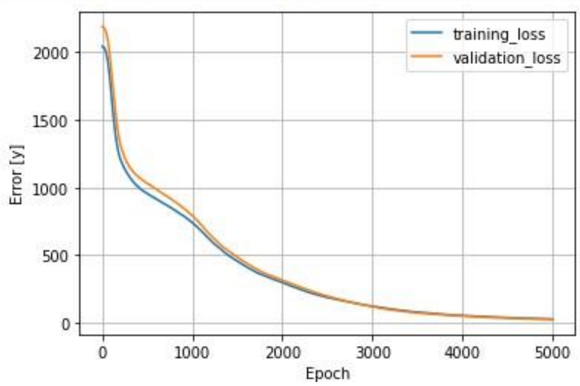
How can we further improve the model prediction, for example:

a. Increase number of neurons, e.g. 128

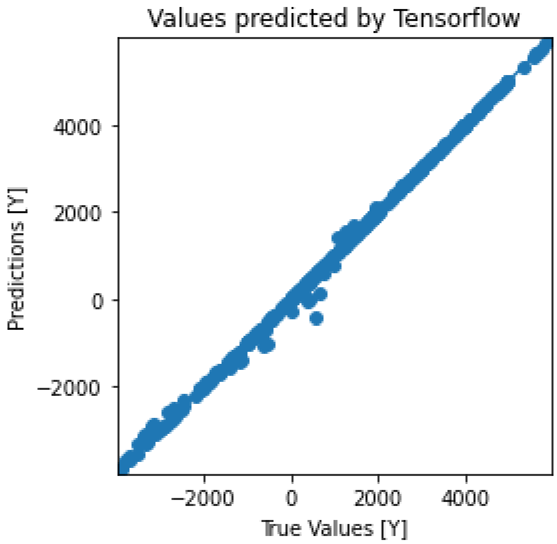
b. Increase number of hidden layers, e.g. use 8 layers instead of 2 hidden layers

c. Increase number of epochs from 5000 to 10000

```
In [9]:  normalizer = preprocessing.Normalization()
         normalizer.adapt(np.array(train_features))
         model = tf.keras.models.Sequential()
         model.add(normalizer)
         model.add(tf.keras.layers.Dense(64, activation="relu"))
         model.add(tf.keras.layers.Dense(64, activation="relu"))
         model.add(tf.keras.layers.Dense(1))
         model.compile(loss="mean_absolute_error",optimizer=tf.keras.optimizers.Adam(0.0001))
         history = model.fit(train_features, train_labels,validation_split=0.2,verbose=0,epochs=5000)
         plot_loss(history)
         test_predictions = model.predict(test_features).flatten()
         plot_prediction_comparison(test_labels, test_predictions)
         test_1p=[[1000,90]] # x1=1000,x2=90 given by user
         print(model.predict(test_1p))
```



This is model which was given to us the model had one keras normalization layer and two hidden layer. The both hidden layers have 64 neurons and activation function used is Relu. In this model we are using Adam Optimizer with learning rate of 0.0001. The epochs which we are using are 5000. At last we are predicting one input which is test_1p = [[1000,90]]. The actual value of this input is 2000 and the predicted value comes out to be 2001.07




```
1/1 [==============================] - 0s 46ms/step
[[2001.0753]]
```

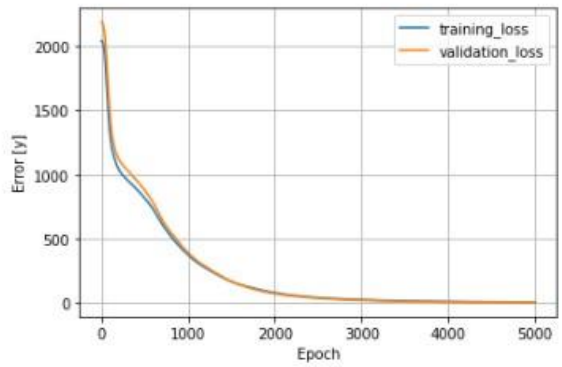
Summary of this model is given as

```
In [15]:  model.summary()

Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_3 (Normalizat  (None, 2)                5
 ion)

 dense_15 (Dense)            (None, 64)                192

 dense_16 (Dense)            (None, 64)                4160

 dense_17 (Dense)            (None, 1)                 65

=================================================================
Total params: 4,422
Trainable params: 4,417
Non-trainable params: 5
_________________________________________________________________
```

a. Increase number of neurons, e.g. 128

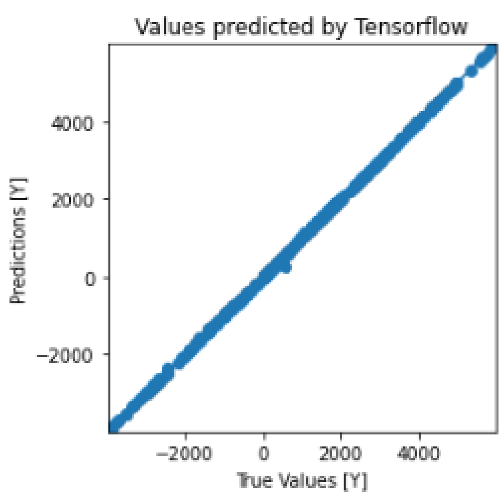
In Part(a) of Deliverable 1 we have increased the number of neurons from 64 to 128 in both the layers.

```
In [10]:  normalizer = preprocessing.Normalization()
          normalizer.adapt(np.array(train_features))
          model = tf.keras.models.Sequential()
          model.add(normalizer)
          model.add(tf.keras.layers.Dense(128, activation="relu"))
          model.add(tf.keras.layers.Dense(128, activation="relu"))
          model.add(tf.keras.layers.Dense(1))
          model.compile(loss="mean_absolute_error",optimizer=tf.keras.optimizers.Adam(0.0001))
          history = model.fit(train_features, train_labels,validation_split=0.2,verbose=0,epochs=5000)
          plot_loss(history)
          test_predictions = model.predict(test_features).flatten()
          plot_prediction_comparison(test_labels, test_predictions)
          test_1p=[[1000,90]] # x1=1000,x2=90 given by user
          print(model.predict(test_1p))
```



A neuron takes a group of weighted inputs, applies an activation function, and returns an output. At First, we increased the number of neurons to 128 from 64, which means we increased the single weighted inputs and as we know by increasing the number of neurons we are increasing the number of more learning units in our system and by that our model will be to learn and predict the values in a more efficient and proper way.

```
12/12 [==============================] - 0s 866us/step
```




```
1/1 [==============================] - 0s 65ms/step
[[1997.9833]]
```

The output value which came from this model is 1998 which is close to the actual value.

The summary of this model is

```
In [11]:  model.summary()

Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_1 (Normalizat  (None, 2)                5
 ion)

 dense_3 (Dense)             (None, 128)               384

 dense_4 (Dense)             (None, 128)               16512

 dense_5 (Dense)             (None, 1)                 129

=================================================================
Total params: 17,030
Trainable params: 17,025
Non-trainable params: 5
_________________________________________________________________
```

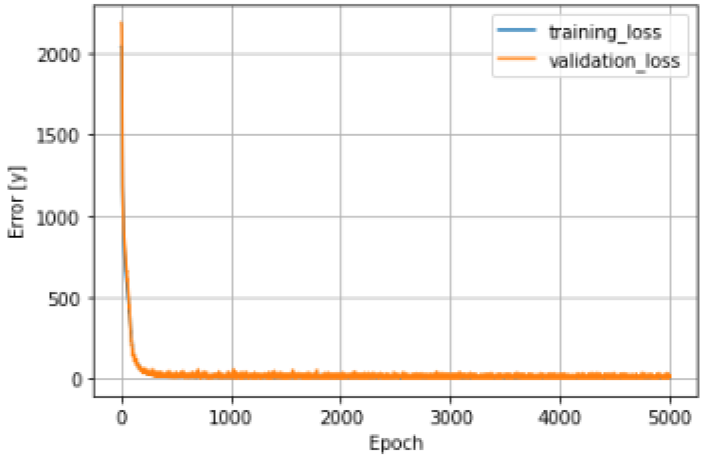
The total parameters in the first Model where there are two layers and 64 neurons in each layer are **4422** but when we increase the hidden layers from 2 to 8. The total trainable parameters increase to **17,025**.

b. Increase number of hidden layers, e.g. use 8 layers instead of 2 hidden layers

In this case we have increased our hidden layers from 2 to 8. The neurons in each layer are 64.

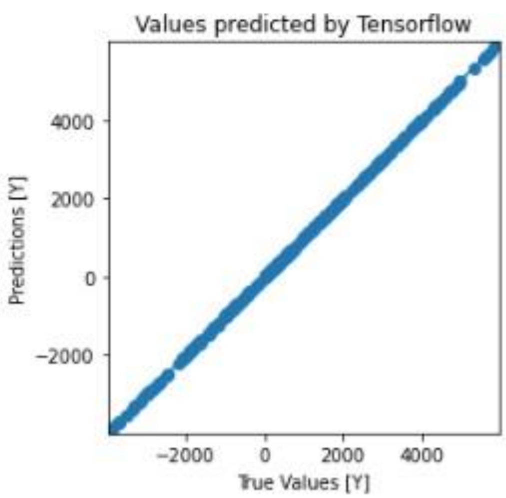
```
In [12]:  # A DNN regression
normalizer = preprocessing.Normalization()
normalizer.adapt(np.array(train_features))
model = tf.keras.models.Sequential()
model.add(normalizer)
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(1))
model.compile(loss="mean_absolute_error",optimizer=tf.keras.optimizers.Adam(0.0001))
history = model.fit(train_features, train_labels,validation_split=0.2,verbose=0,epochs=5000)
plot_loss(history)
test_predictions = model.predict(test_features).flatten()
plot_prediction_comparison(test_labels, test_predictions)
test_1p=[[1000,90]] # x1=1000,x2=90 given by user
print(model.predict(test_1p))
```

Validation and Training Loss:




```
12/12 [==============================] - 0s 993us/step
```

Output:




```
1/1 [==============================] - 0s 94ms/step
[[1978.3245]]
```

Though increasing the number of hidden layers improves the performance but in this case, it is not as effective as increasing the number of neurons or increasing the number of epochs value/iterations over the training data set. The output in this case is **1978.32**.

The model summary is

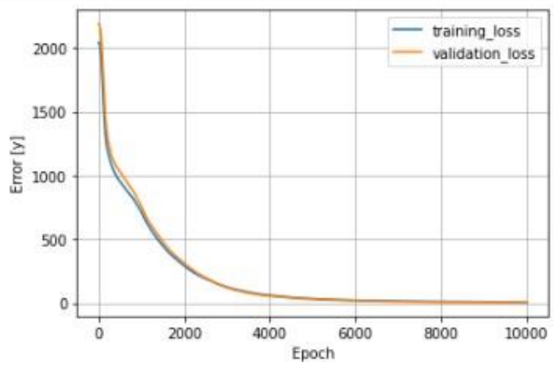
```
In [13]:  model.summary()

Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_2 (Normalizat  (None, 2)                5
 ion)

 dense_6 (Dense)             (None, 64)                192

 dense_7 (Dense)             (None, 64)                4160

 dense_8 (Dense)             (None, 64)                4160

 dense_9 (Dense)             (None, 64)                4160

 dense_10 (Dense)            (None, 64)                4160

 dense_11 (Dense)            (None, 64)                4160

 dense_12 (Dense)            (None, 64)                4160

 dense_13 (Dense)            (None, 64)                4160

 dense_14 (Dense)            (None, 1)                 65

=================================================================
Total params: 29,382
Trainable params: 29,377
Non-trainable params: 5
_________________________________________________________________
```

The total parameters in the first Model where there are two layers and 64 neurons in each layer are **4422** but when we increase the hidden layers from 2 to 8. The total trainable parameters increase to **29,377**.

c. Increase number of epochs from 5000 to 10000

In [14]:

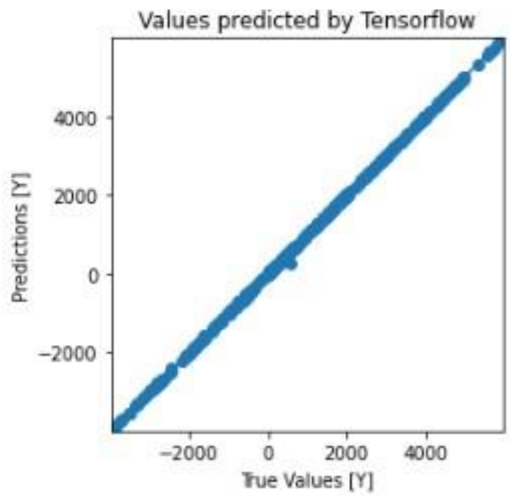
```python
# A DNN regression
normalizer = preprocessing.Normalization()
normalizer.adapt(np.array(train_features))
model = tf.keras.models.Sequential()
model.add(normalizer)
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(1))
model.compile(loss="mean_absolute_error",optimizer=tf.keras.optimizers.Adam(0.0001))
history = model.fit(train_features, train_labels,validation_split=0.2,verbose=0,epochs=10000)
plot_loss(history)
test_predictions = model.predict(test_features).flatten()
plot_prediction_comparison(test_labels, test_predictions)
test_1p=[[1000,90]] # x1=1000,x2=90 given by user
print(model.predict(test_1p))
```



In this case the number of hidden layers are two and each layer have 64 neurons but in this case we have increased our epochs value from 5000 to 10,000.

we saw that it usually improves our model when we increased the number of epochs to 7000 or 8000, after that it starts to overfit and the results/predicted output on the new fetched/given output test data was not as much efficient. Overfitting led this to make it difficult to adapt and predict the output on the new test_data, as it showed great performance on the training data set. as overfitted data performs well on the training data and could not easily adapt to the new test data given to it.

Output:

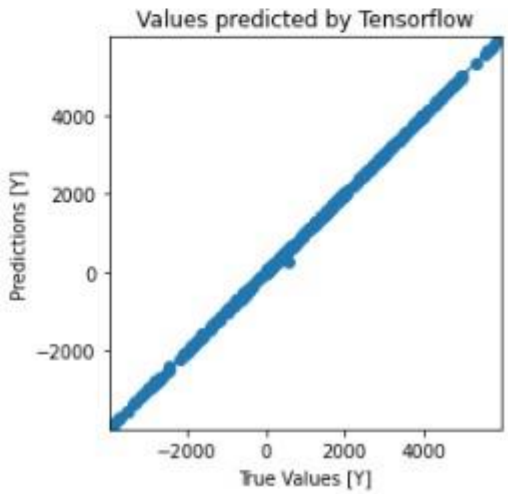



```
1/1 [==============================] - 0s 72ms/step
[[2003.5446]]
```

Model Summary:

In [15]:
```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_3 (Normalizat  (None, 2)                5
 ion)

 dense_15 (Dense)            (None, 64)                192

 dense_16 (Dense)            (None, 64)                4160

 dense_17 (Dense)            (None, 1)                 65

=================================================================
Total params: 4,422
Trainable params: 4,417
Non-trainable params: 5
_________________________________________________________________
```




```
1/1 [==============================] - 0s 72ms/step
[[2003.5446]]
```

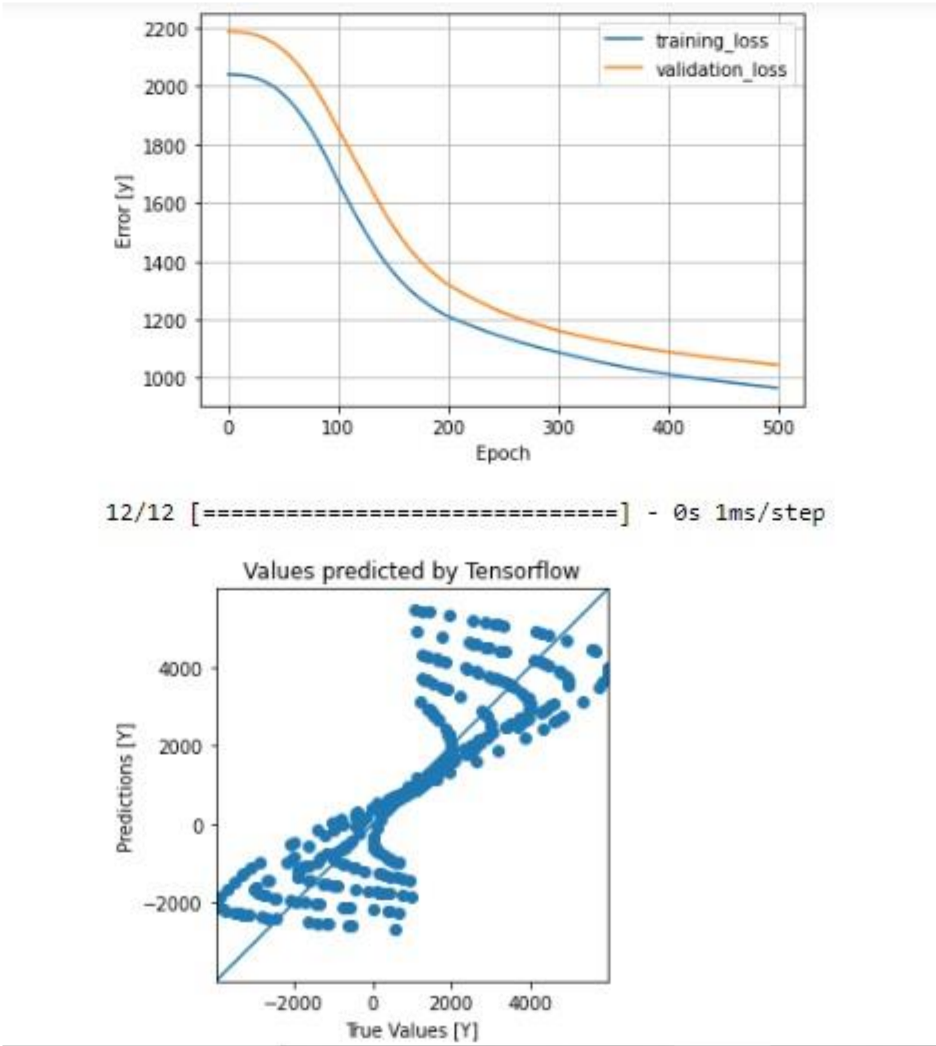
Conclusion:

At last, we would conclude that all of the techniques are improving our model in some way but some have their limitations and increasing the number of neurons had proved to be the best and most optimized modular technique to improve our model and make it adapt to the new dataset and make the good predictions.

Deliverable Number 2

2. How can we get better results using lesser epochs e.g. 500 instead of 5000? (e.g. Increase layers, change learning rate value)

When we reduced our epochs from 5000 to 500, epochs is basically a hyperparameter that defines the number of times that the learning algorithm will work through the entire training dataset. By reducing the epochs to 500, we saw a considerable change in the training loss, validation loss and the values predicted via this model. The training loss and validation loss both increased tremendously from the previous epochs of 500 tested in the deliverable 1.

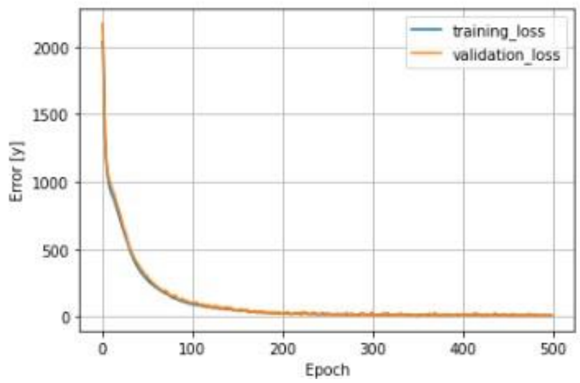


This time to optimize our model and make it improve, we have to keep the epochs to 500 and change the learning rate and the number of layers in our model of the neural network following are the steps we take to optimize our model and what result achieved the best and improved optimize model.

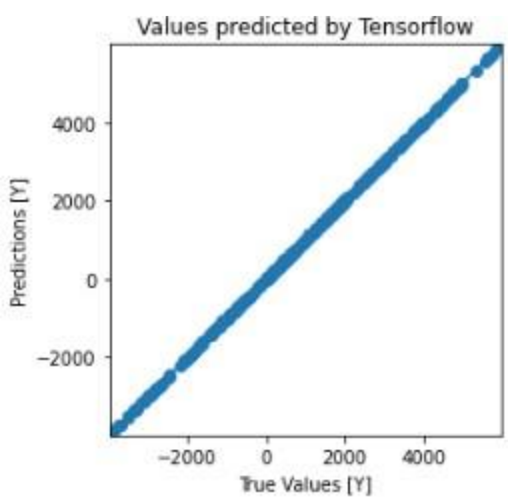
We used variety of combination to get the best optimized model result and for that we changed the learning_rate and number of layers and from that we checked the combination between the learning_rate and the number of layers, we implemented all these combination in the jupyter notebook staring from the learning_rate = 0.001 and the number of layers = 2 and the best result we got from checking all these combination was when we put the learning_rate = 0.04 and number of layers = 2,the output they gave was 2004.3 which is almost near to actual value of 2000 and the error is minimal in this case as compared to the other combinations of the learning_rate and the number of layers in our model of the neural network. Below is the pictorial representation which depicts our result of 2004.3 and the respective training and validation losses.

Epochs = 500, Learning Rate = 0.004, Hidden Layers = 2

```
In [45]:  normalizer = preprocessing.Normalization()
          normalizer.adapt(np.array(train_features))
          model = tf.keras.models.Sequential()
          model.add(normalizer)
          model.add(tf.keras.layers.Dense(64, activation="relu"))
          model.add(tf.keras.layers.Dense(64, activation="relu"))
          model.add(tf.keras.layers.Dense(1))
          model.compile(loss="mean_absolute_error",optimizer=tf.keras.optimizers.Adam(0.004))
          history = model.fit(train_features, train_labels,validation_split=0.2,verbose=0,epochs=500)
          plot_loss(history)
          test_predictions = model.predict(test_features).flatten()
          plot_prediction_comparison(test_labels, test_predictions)
          test_1p=[[1000,90]] # x1=1000,x2=90 given by user
          print(model.predict(test_1p))
```



Output:




```
1/1 [==============================] - 0s 73ms/step
[[2004.3031]]
```

Deliverable Number 3

In deliverable number 3 we removed keras normalization layer and we created two functions when for normalization and the other for denormalization. The screenshot of both function is attached.

```
In [48]:  def new_normalization(to_normalize_dataset):

              #print(dataset)

              new_dataset = ( 2 * (to_normalize_dataset - dataset.min()) / (dataset.max() - dataset.min()) ) - 1

              return new_dataset
```

```
In [49]:  def de_normalization (new_dataset):

              old_dataset = ( (new_dataset + 1 ) * (dataset.max() - dataset.min()) ) / 2 + dataset.min()

              return old_dataset
```

As mentioned in task we normalized our data between -1 to +1. A new variable 'normalized_dataset' will be used in the code now and keras normalization layer will be

```
In [50]:  normalized_dataset = new_normalization(dataset)
```

```
In [51]:  normalized_dataset.describe()
```

Out[51]:

	x1	x2	y
count	1805.000000	1.805000e+03	1805.000000
mean	0.000000	-7.873050e-18	-0.000037
std	0.707303	5.791122e-01	0.468456
min	-1.000000	-1.000000e+00	-1.000000
25%	-0.500000	-5.000000e-01	-0.335400
50%	0.000000	0.000000e+00	0.000000
75%	0.500000	5.000000e-01	0.335400
max	1.000000	1.000000e+00	1.000000

romoved and after the prediction the the dataset will be converted to actual values.

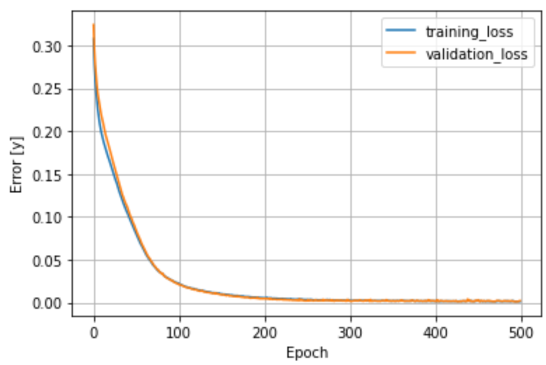
normalized_dataset is also described by using describe function. normalized_dataset is now zero mean centered and the minimum value is -1 and the maximum value is +1.

The train and test dataset is prepared on the base of normalized_dataset.

In [54]:
```
# Prepare train and test data
train_dataset = normalized_dataset.sample(frac=0.8, random_state=0)
test_dataset = normalized_dataset.drop(train_dataset.index)
train_features = train_dataset.copy()
test_features = test_dataset.copy()
train_labels = train_features.pop(objectiveFunction)
test_labels = test_features.pop(objectiveFunction)
```

Now, the same model was created but this time we did not use the normalization layer from keras. We trained our model on the features which were already normalized. The output given by the model will also be normalized but we will use our denormalization function to change it to actual value.

In [56]:
```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(1))
model.compile(loss="mean_absolute_error",optimizer=tf.keras.optimizers.Adam(0.0001))
history = model.fit(train_features, train_labels,validation_split=0.2,verbose=0,epochs=500)
plot_loss(history)
test_predictions = model.predict(test_features)
```




```
12/12 [==============================] - 0s 1ms/step
```

For the input part we cannot use the test_1p = [[1000,90]] and get the correct answers because now our model is built on the normalized dataset. The solution to this problem is this we normalize the input part also.

```
In [57]: test_1p=[[1000,90]]
```

```
In [58]: #Converting to pandas Dataframe
         normalized_test_1p = pd.DataFrame (test_1p, columns = [column_names[0],column_names[1]] )
```

```
In [60]: #Normalizing test_1p
         normalized_test_1p = new_normalization(normalized_test_1p)
```

```
In [62]: x1 = normalized_test_1p.iloc[0]['x1']
         x2 = normalized_test_1p.iloc[0]['x2']
         test_1p = [[x1 , x2]]
```

```
In [65]: test_1p
Out[65]: [[-1.0, -0.5]]
```

Now the test_1p point is normalized we can use this input in our model to get good results.

```
In [66]: print(model.predict(test_1p))
         prediction = model.predict(test_1p)

         1/1 [==============================] - 0s 260ms/step
         [[0.19807297]]
         1/1 [==============================] - 0s 30ms/step
```

We saved the result of model.predict(test_1p) in a variable prediction. This result is normalized and we want to de normalize it.

```
In [67]: un_normalized_output_predicted = pd.DataFrame (prediction, columns = [column_names[2]] )
         output_prdicted = de_normalization (un_normalized_output_predicted)
```

```
In [69]: output_prdicted['y']
Out[69]: 0    1990.364552
         Name: y, dtype: float64
```

The result is 1990.364 which is close to the actual value of 2000. In our view this model has performed well.

Question: Any other additional tricks with supported python code to get normalization done

Yes, Normalization can be done in different ways. One of the technique which we have used is using a Standard Scalar from sklearn.preprocessing.

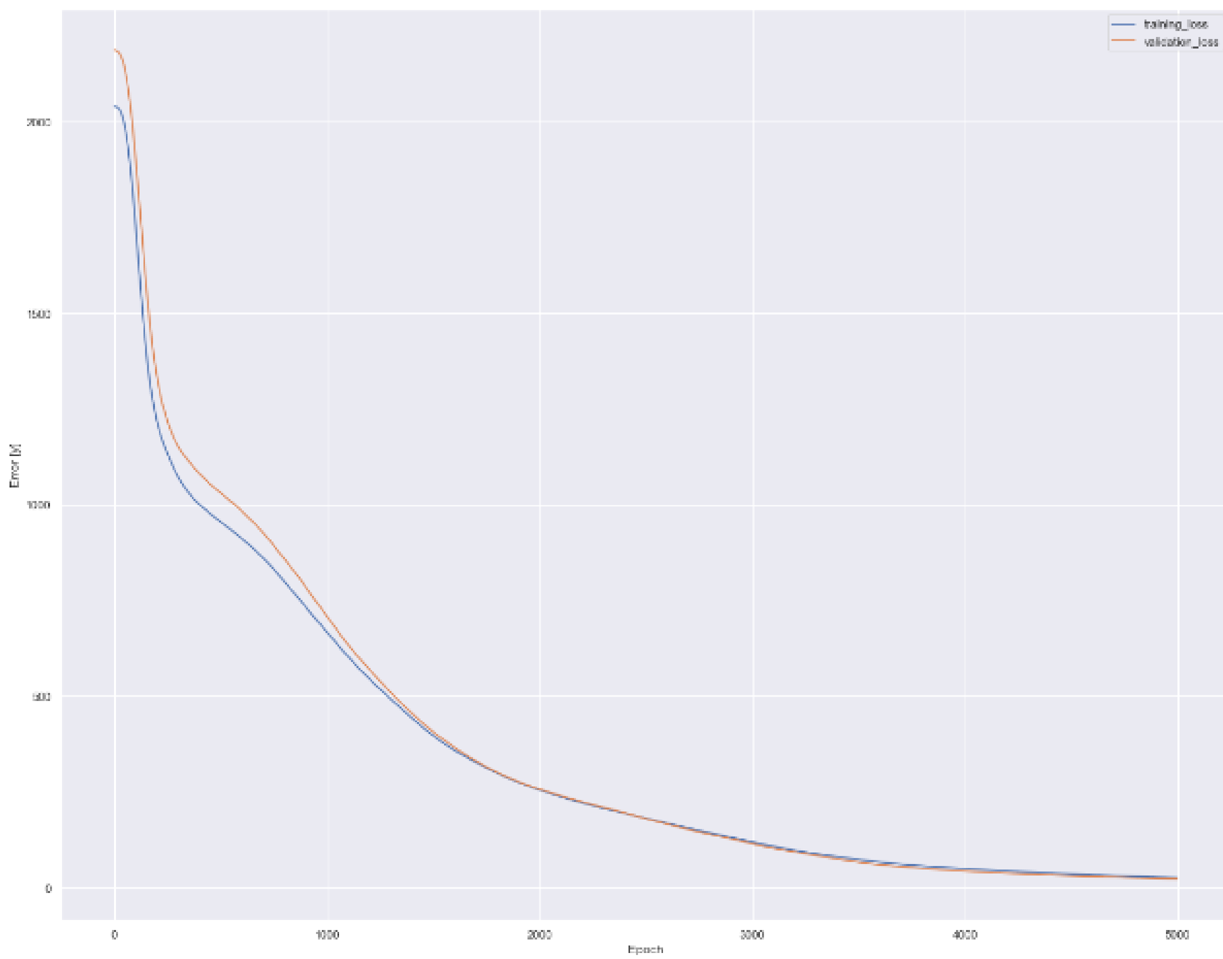
First from sklearn.preprocessing we imported StandardScaler and then created a object of StandardScaler. We then fit our scalar according to train features and after fitting we scaled our train and test features.

```
In [245]:  from sklearn.preprocessing import StandardScaler

           scaler = StandardScaler()
           scaler.fit(train_features)

           train_features_scaled = scaler.transform(train_features)
           test_features_scaled = scaler.transform(test_features)
```

After that we created our model without keras normalization layer. We used train_features_scaled in our model.fit and in model.predict we used our test_features_scaled.

```
In [248]:  model = tf.keras.models.Sequential()
           model.add(tf.keras.layers.Dense(64, activation="relu"))
           model.add(tf.keras.layers.Dense(64, activation="relu"))
           model.add(tf.keras.layers.Dense(1))
           model.compile(loss="mean_absolute_error",optimizer=tf.keras.optimizers.Adam(0.0001))
           history = model.fit(train_features_scaled, train_labels,validation_split=0.2,verbose=0,epochs=5000)
           plot_loss(history)
           test_predictions = model.predict(test_features_scaled)
           plot_prediction_comparison(test_labels, test_predictions)
```

Training and Validation Loss




```
12/12 [==============================] - 0s 836us/step
```

For the testing of our model as in the above case we have trained our model on the scaled dataset so we have to also scale our input to get correct results. We have scaled our input by the scalar.

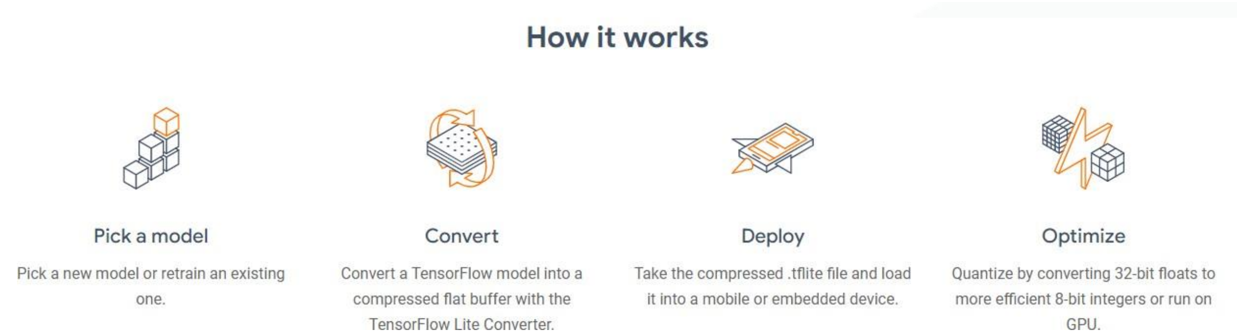
```
In [249]:  column_names = list(dataset.columns)
           test_1p = [[1000,90]]
           test_1p = pd.DataFrame (test_1p, columns = [column_names[0],column_names[1]] )

           test_1p = scaler.transform(test_1p)

           print(model.predict(test_1p))

           1/1 [==============================] - 0s 15ms/step
           [[1998.2666]]
```

The output value in this case is 1998.266. There are many techniques in which normalization can be done. There our many scalars available for example MinMax Scalar from sklearn.

Deliverable Number 4

How can we save/convert this model to tensorflow LITE format?

In this deliverable we changed our tensorflow model to tensorflow LITE (tflite) model. TensorFlow Lite is a mobile library for deploying models on mobile, microcontrollers and other edge devices.



At first, we our creating a TensorFlow model we are not using keras Normalization layer we our using StandardScalar from sklearn.preprocessing.

```
In [250]:  from sklearn.model_selection import train_test_split
           from sklearn.preprocessing import StandardScaler
```

```
In [251]:  X = dataset.drop('y', axis = 1)
           Y = dataset['y']
```

We also imported train_test_split from sklearn we used train_test_split to split our dataset this time.

```
In [252]:  X_train , X_test , Y_train , Y_test = train_test_split(X,Y,test_size = 0.2, random_state = 0)
```

After that we created a scalar object of StandardScaler and we fit this scalar according to X_train and after that scaled our train and test features.

```
In [253]:  scaler = StandardScaler()
           scaler.fit(X_train)
           X_trained_scaled = scaler.transform(X_train)
           X_test_scaled = scaler.transform(X_test)
```

After the creation of train and test features we created our model without normalization layer. We created our model on the base of X_trained_Scaled.

```
In [254]:
#Creating New Model where we have removed Normalization Layer.
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(1))
model.compile(loss="mean_absolute_error",optimizer=tf.keras.optimizers.Adam(0.002))

history = model.fit(X_trained_scaled, Y_train,validation_split=0.2,epochs = 5000)
```

The Summary of TensorFlow Model is given as

```
In [173]:
model.summary()

Model: "sequential_21"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_111 (Dense)           (None, 64)                192

 dense_112 (Dense)           (None, 64)                4160

 dense_113 (Dense)           (None, 1)                 65

=================================================================
Total params: 4,417
Trainable params: 4,417
Non-trainable params: 0
_________________________________________________________________
```

We tested this model by test_1p. we first scaled this input and then predicted value of model on the behalf of scaled input.

```
In [255]:
column_names = list(dataset.columns)
test_1p = [[1000,90]]
test_1p = pd.DataFrame (test_1p, columns = [column_names[0],column_names[1]] )

test_1p = scaler.transform(test_1p)

print(model.predict(test_1p))

1/1 [==============================] - 0s 61ms/step
[[2001.9921]]
```

The Output Value is 2001.9921

Converting TensorFlow Model to TFLITE

To convert a TensorFlow model to a tflite model the first step is to define a converter and then call a convert function.

```
In [256]:  converte = tf.lite.TFLiteConverter.from_keras_model(model)
           tflite_model = converte.convert()

           INFO:tensorflow:Assets written to: C:\Users\TALHAS~1\AppData\Local\Temp\tmpdkucfo40\assets

           INFO:tensorflow:Assets written to: C:\Users\TALHAS~1\AppData\Local\Temp\tmpdkucfo40\assets
```

In order to evaluate tflite model we have to setup interpreter. The interpreter will give us the input and output details.

```
In [257]:  interpreter = tf.lite.Interpreter(model_content = tflite_model)
           interpreter.allocate_tensors()
           input_details = interpreter.get_input_details()
           output_details = interpreter.get_output_details()

In [258]:  input_details

Out[258]: [{'name': 'serving_default_dense_144_input:0',
            'index': 0,
            'shape': array([1, 2]),
            'shape_signature': array([-1,  2]),
            'dtype': numpy.float32,
            'quantization': (0.0, 0),
            'quantization_parameters': {'scales': array([], dtype=float32),
             'zero_points': array([], dtype=int32),
             'quantized_dimension': 0},
            'sparsity_parameters': {}}]
```

The input_details are helpful in considering the input shape of the tflite model.

```
In [177]:  input_shape = input_details[0]['shape']
           input_shape

Out[177]: array([1, 2])
```

The input given to our model should be array of size (1,2) and its datatype should be float32. So we change the datatype of our test_1p from float64 to float32.

```
In [265]:  test_1p.dtype

Out[265]: dtype('float64')

In [178]:  input_data = np.float32(test_1p)
```

Now we set the tensors to the input which we created so that we can get our prediction.

```
In [267]: interpreter.set_tensor(input_details[0]['index'],input_data)
```

```
In [268]: interpreter.invoke() #model.predict()
```

```
In [269]: output_data_tflite = interpreter.get_tensor(output_details[0]['index'])
```

```
In [270]: output_data_tflite
Out[270]: array([[2001.9922]], dtype=float32)
```

The output in this case is also 2001.9922.

Save TensorFlow Lite Model

```
In [272]: TF_LITE_MODEL_NAME = 'tf_lite_model.tflite'
```

```
In [273]: open(TF_LITE_MODEL_NAME , "wb").write(tflite_model)
Out[273]: 19572
```

Deliverable Number 6

In this task we have to access the weights and biases of model after training it. Then we have to form a equation by which we can predict values (without using model.predict)

Importing Libraries and setting initial values

```
In [9]: from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from pandas.plotting import scatter_matrix
import seaborn as sns
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from keras.models import load_model
from tensorflow.keras.layers.experimental import preprocessing
import plotly.express as px
import plotly.graph_objects as go
import math
print(tf.__version__) # 2.4.1
print(keras.__version__) # 2.4.0
# Some helping function definitions

2.9.1
2.9.0
```

```
In [10]:  v1_outputcolumns = -1
          v2_neurons = 128
          v3_nr_of_d_layers = 8
          v4_activation_function = "relu"
          v5_loss_function ="mean_absolute_error"
          v6_optimizer = "Adam"
          v7_metrics = 'accuracy'
          v8_learning_rate = 0.001
          v9_nr_of_epochs = 500
          v10_d_split = 0.2
          v11_train_function = 0.8
          v12_csv_separator = ";"

          if v3_nr_of_d_layers < 1:
              v3_nr_of_d_layers = 1
```

Setting the values of neurons, layers epochs and activation function

Creating Functions

Function for Weight and biases for a Particular Layer

```
In [64]:  def get_layers_weights_biases(model,nr_of_d_layers):
              layers_wights = list()
              layers_biases = list()
              for i in range(1, nr_of_d_layers+2):
                  weight = model.layers[i].get_weights()[0]
                  bias = model.layers[i].get_weights()[1]
                  layers_wights.append(weight)
                  layers_biases.append(bias)

              return(layers_wights,layers_biases)
```

Function for Mean and Standard Deviation

```
In [40]:  def get_mean_and_standarddeviation(model):
              mean_std = list()
              weights = model.layers[0].get_weights()[0]
              biases = model.layers[0].get_weights()[1]
              standard_deviation_all = list()
              mean_all = list()
              for i in range(len(biases)):
                  standard_deviation_all.append(np.sqrt(biases[i]))
                  mean_all.append(weights[i])
              mean_std.append(mean_all)
              mean_std.append(standard_deviation_all)
              return(mean_std)
```

Normalization Function

```
In [327]:   def normalize_list(val_list,mean_all_list,standard_deviation_all_list):
                test_fa = list()
                for i in range(len(val_list)):
                        x_normalize = list()
                        x1_normalize = (val_list[i][0]-mean_all_list[0])/standard_deviation_all_list[0]
                        x2_normalize = (val_list[i][1]-mean_all_list[1])/standard_deviation_all_list[1]
                        x_normalize.append([x1_normalize,x2_normalize])
                        test_fa.append(x_normalize)
            #             print(test_fa)



                return(test_fa)
```

ReLu and Sigmoid Activation Function

```
In [184]:   def ReLU(x):
                return x * (x > 0)
```

```
In [185]:   def sigmoid(x):
                return 1/(1+math.e**(-x))
```

Prediction Function which can be used to replace model.predict

```
In [328]:   def MyPrediction(X,layers_wights,layers_biases,mean_all,standard_deviation_all,activation_function):
                normalized_data = normalize_list(X,mean_all,standard_deviation_all)
                print('The normalized_data is {}',normalized_data)
                predicted_values = list()
                for a in normalized_data:
            #         print(a)
                    for i in range(len(a)):
                        weights = layers_wights[i-1]
            #             print('The weight value is {}',weights)
                        biases = layers_biases[i-1]
            #             print('The bias value is {}',biases)
            #             print('The type of a is {}',a[0].dtypes)
                        z =  dotproduct(a,weights)+biases
                        if i < len(layers_wights):
                            if activation_function == 'relu':
                                a = ReLU(z)
                            elif activation_function == 'sigmoid':
                                a = sigmoid(z)
                            else:
                                print("ERROR: Unknown Activation Function")
                                print("ERROR: RELU applied as activation function")
                                a=Relu(z)
                        else:
                            a=z #Final Layer

                    a = predicted_result(a)
                    predicted_values.append(a)
                return(predicted_values)
```

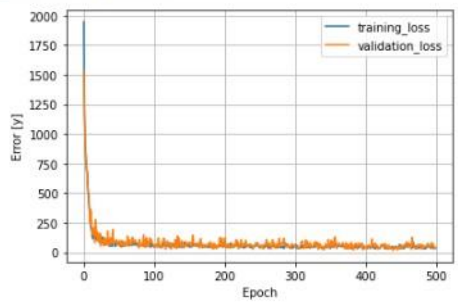
Compiling Model

```
In [278]:  # A DNN regression
normalizer = preprocessing.Normalization()
normalizer.adapt(np.array(train_features))
model = tf.keras.models.Sequential()
model.add(normalizer)
for i in range(v3_nr_of_d_layers):
    model.add(tf.keras.layers.Dense(v2_neurons, activation=v4_activation_function))
model.add(tf.keras.layers.Dense(1))
```

```
In [279]:  if v6_optimizer == "Adam":
    model.compile(loss=v5_loss_function,optimizer = tf.keras.optimizers.Adam(v8_learning_rate),metrics = [v7_metrics])
elif v6_optimizer == "SGD":
    model.compile(loss=v5_loss_function,optimizer = tf.keras.optimizers.SGD(v8_learning_rate),metrics = [v7_metrics])
else:
    model.compile(loss=v5_loss_function,optimizer = tf.keras.optimizers.Adam(v8_learning_rate),metrics = [v7_metrics])
```

Training and Validation Loss

```
In [280]:  history = model.fit(train_features,train_labels, validation_split = v10_d_split,verbose = 0, epochs = v9_nr_of_epochs)
plot_loss(history)
```



Testing Data for self_made and built_in Model

```
In [333]:  test_1p = [[1000,90]]
print('test_1_point')
print(test_1p)
print('y_by_model.predict')`
print(model.predict(test_1p))
print('y_by_MyPrediction')
test_1p_eq = MyPrediction(test_1p,layers_wights,layers_biases,mean_all,stddev_all,v4_activation_function)
print(test_1p_eq)
```

```
test_1_point
[[1000, 90]]
y_by_model.predict
1/1 [==============================] - 0s 78ms/step
[[2026.2643]]
y_by_MyPrediction
The normalized_data is {} [[[-1.419863963068285, -0.8695824558143754]]]
[2029.7986582481333]
```

The Model.predict() predicted the output value to be 2026.2643 when this built-in function was used and the Manual prediction procedure predicted that output to be 2029.7986, which within 1 to 1.5% percent of error which is acceptable in terms of its accuracy.

Deliverable Number 7

Provide python and plotly code for above plots and other additional plots recommended by you to analyse the the things in more details

a. Plot model accuracy in percentage

At First, we learned/trained/compiled our model by putting up the metrics = accuracy, and with this the history variable(that is the variable equal to *model.fit, which is used to train the model/adapt to the model) is used to compute the accuracy of our model by accessing its accuracy values of the predicted model as compared to the true values by using the command of history.history['accuracy']

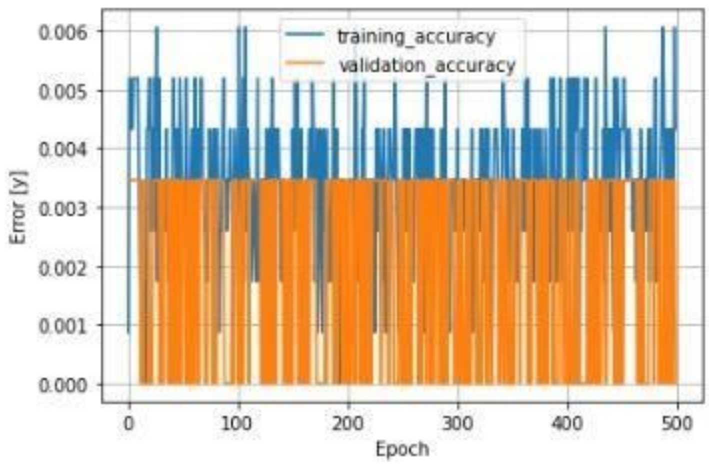
```
# A DNN regression
normalizer = preprocessing.Normalization()
normalizer.adapt(np.array(train_features))
model = tf.keras.models.Sequential()
model.add(normalizer)
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(1))
model.compile(loss="mean_absolute_error",optimizer=tf.keras.optimizers.Adam(0.02),metrics = 'accuracy')
history = model.fit(train_features, train_labels,validation_split=0.2,verbose=0,epochs=500)
plot_loss(history)
test_predictions = model.predict(test_features).flatten()
plot_prediction_comparison(test_labels, test_predictions)
test_1p=[[1000,90]] # x1=1000,x2=90 given by user

Predicted_Value = model.predict(test_1p)
print(Predicted_Value)
```

Through this function, I was able to access to the accuracy of the model, but we can see from the graph that our model is very much scattered, we don't have the consistent output and for that we made the function that would be calculating the accuracy in the form of percentage manually.

```python
from sklearn.metrics import mean_absolute_error
```

```python
def plot_accuracy(history):
                plt.plot(history.history['accuracy'], label='training_accuracy')
                plt.plot(history.history['val_accuracy'], label='validation_accuracy')
                plt.xlabel('Epoch')
                plt.ylabel('Error [y]')
                plt.legend()
                plt.grid(True)
                plt.show()
```

```python
plot_accuracy(history)
```

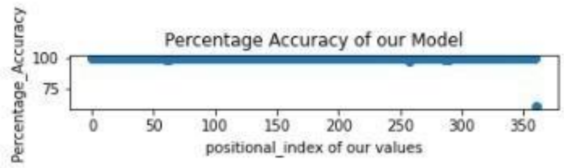


Function for Manual Prediction of the Accuracy of our model

```python
In [78]:  def plot_prediction_accuracy(pred_a, pred_b):
                    total_accuracy = list()
                    position = list()
                    lmin = min(min(pred_a), min(pred_b))
                    lmax = max(max(pred_a), max(pred_b))
                    plt.figure()
                    a = plt.axes(aspect='equal', title='Percentage Accuracy of our Model')
                    for i in range(len(pred_a)):
                        percentage_accuracy = 100 - (np.absolute(pred_a[i]-pred_b[i])/pred_b[i])
                        total_accuracy.append(percentage_accuracy)
                        position.append(i)
                    plt.xlabel('positional_index of our values')
                    plt.ylabel('Percentage_Accuracy')
                    plt.plot(position,total_accuracy)
          #           plt.scatter(position,total_accuracy)
                    plt.show()
```

In Total we have 361 positional indexes which means that our train_features ,test_features ,train_labels ,test_labels are of the length of 361. So we plotted our accuracy along the each iterated value of our positional index and we can see accuracy in the form of percentages

```
In [81]: plot_prediction_accuracy(test_labels_array,all_dataset_prediction)
```



b. Plot two 3D surfaces to compare surface generated by given data and the surface generated by predicted data.

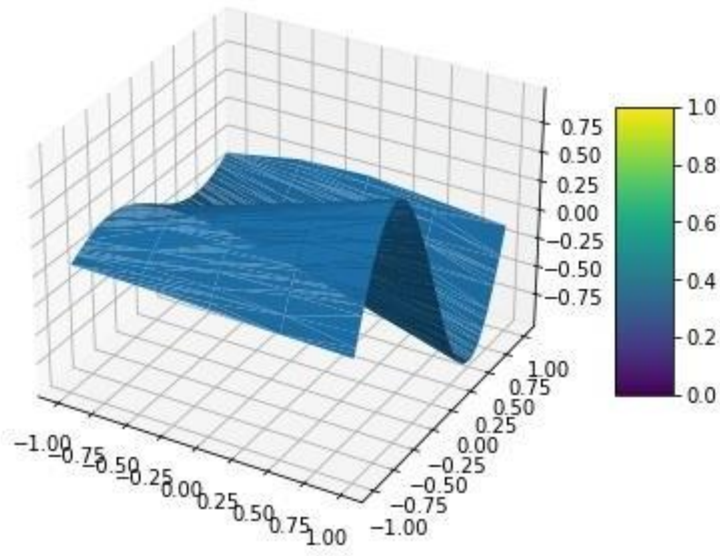
In this deliverable we have to plot two 3d surfaces one of which is generated by the given data. The Surface generated by the given data is plotted between the test features and test labels.The Surface Generated by the the predicted data is plotted between the test features and test predictions.

The Surface Generated by the given data:

Code:

```
In [88]: x1 = test_features['x1']
         x2 = test_features['x2']
         y = test_labels

         fig = plt.figure()
         ax = Axes3D(fig)
         surf = ax.plot_trisurf(x1, x2, y, linewidth=0.1)
         fig.colorbar(surf, shrink=0.5, aspect=5)
         plt.show()
```

Graph:



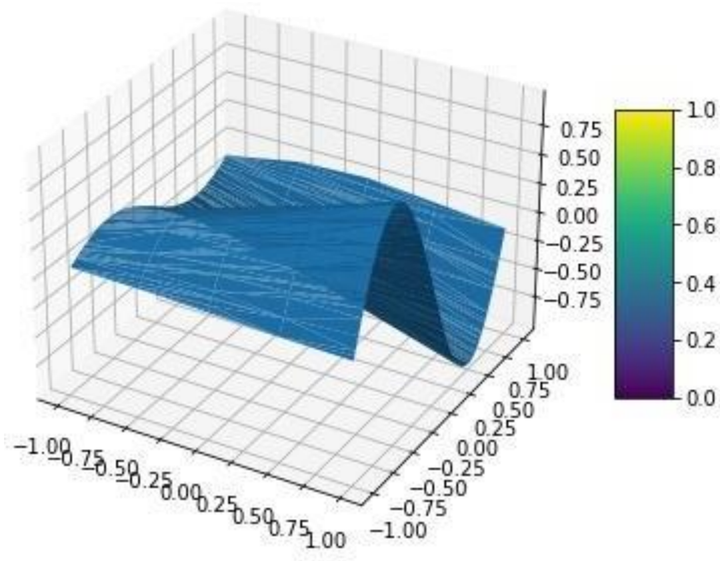
The Surface Generated by the Predicted Data

Code:

In [89]:

```
x1 = test_features['x1']
x2 = test_features['x2']
y = test_predictions

fig = plt.figure()
ax = Axes3D(fig)
surf = ax.plot_trisurf(x1, x2, y, linewidth=0.1)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```

Graph:



From 3d graph plot we can see that the graph between test_features and test_labels is similar to test_features and test_predictions.

c. Scatter plot for all features

Scatter Plot are used to observe the relationship between two numeric variables. We have two features in this model and one output variable. We will observe the relationship between each feature and output and also observe the relationship between two features.

There are three quantities in total. Three different variables are created to store there values.

In [298]:
```
dataset
```

Out[298]:

	x1	x2	y
0	1000	0	1000
1	1000	1	1017
2	1000	2	1034
3	1000	3	1052
4	1000	4	1069
...	...	...	...
1800	5000	356	651
1801	5000	357	738
1802	5000	358	825
1803	5000	359	912
1804	5000	360	999

1805 rows × 3 columns

In [299]:
```
x1 = dataset['x1']
x2 = dataset['x2']
y = dataset['y']
```

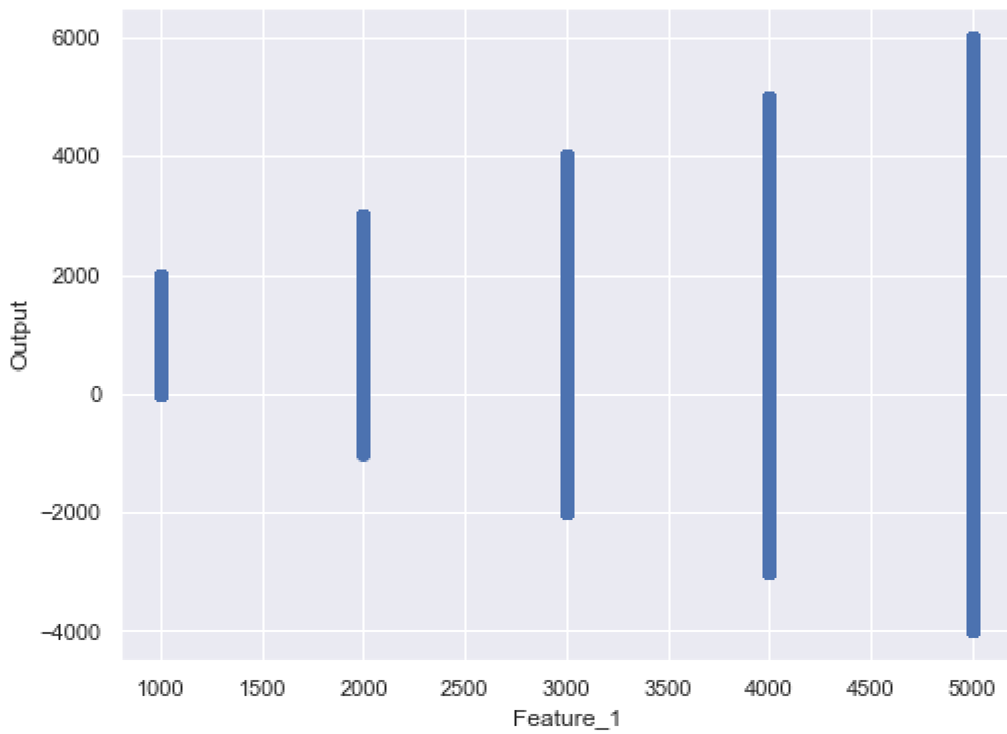
Scatter Plot between Feature 1 and Output

Code:

In [305]:
```
plt.scatter(x1 , y )

plt.xlabel('Feature_1')
plt.ylabel('Output')
```

Out[305]: Text(0, 0.5, 'Output')

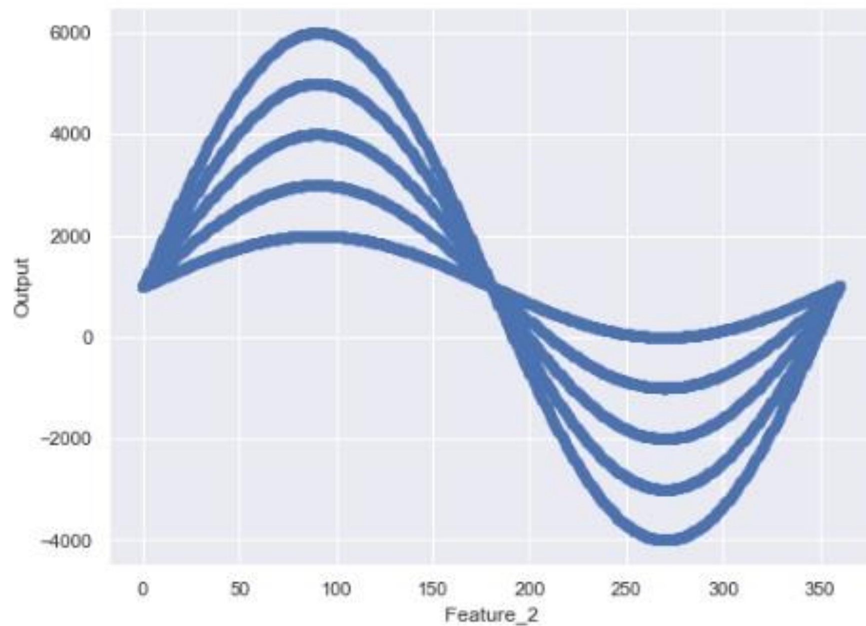
Graph:



Scatter Plot between Feature 2 and Output

In [311]:
```
#Scatter Plot Between Feature 2 and Output
plt.figure(figsize=(8,6))
plt.scatter(x2 , y )

plt.xlabel('Feature_2')
plt.ylabel('Output')
```

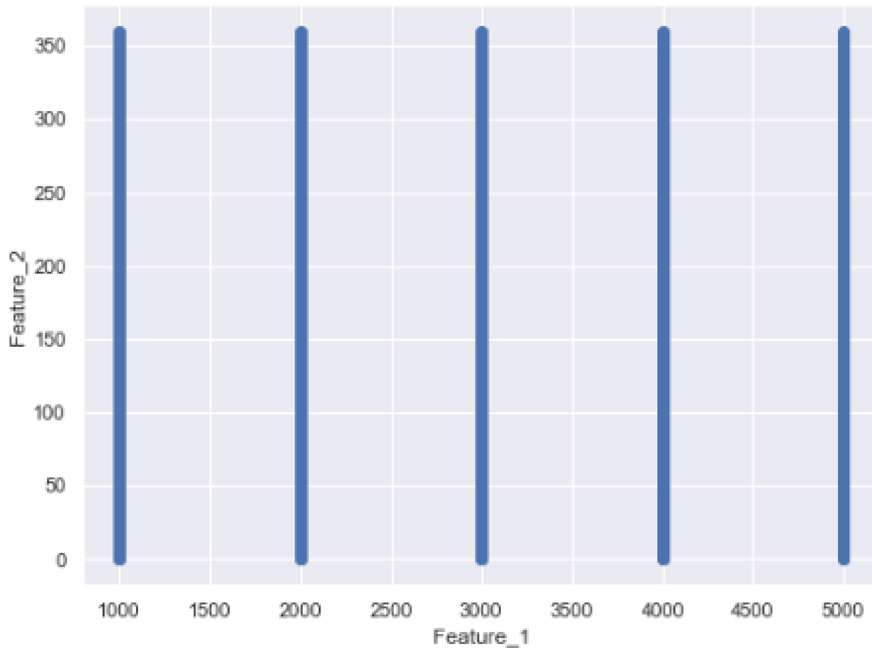
Out[311]: Text(0, 0.5, 'Output')



Scatter Plot between Feature 1 and Feature 2

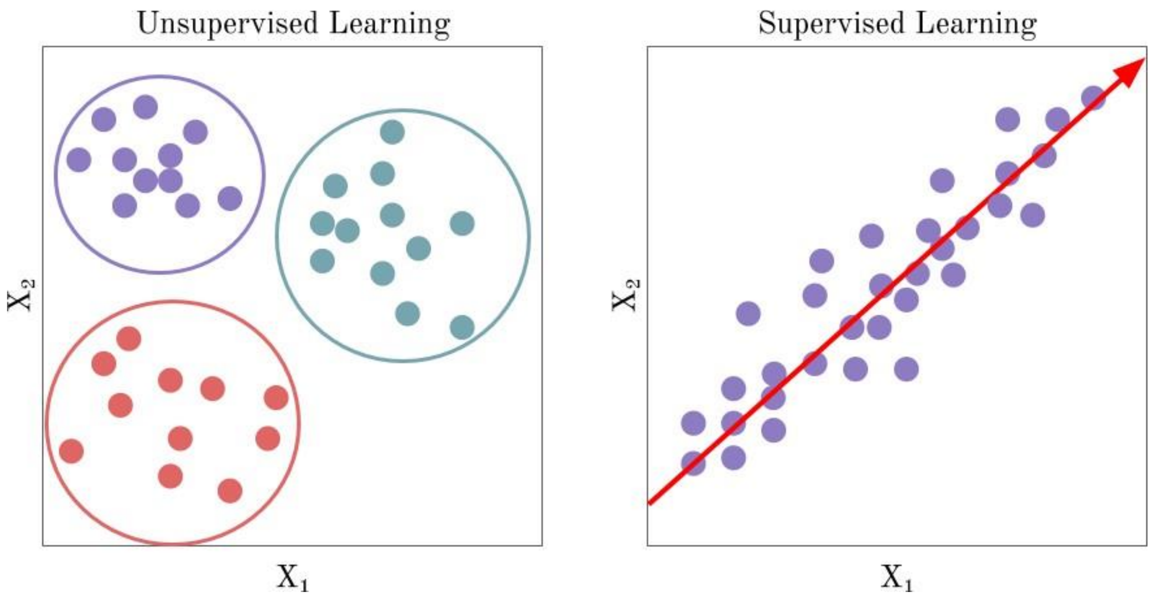
In [312]:
```
#Scatter Plot Between Feature 1 and Feature 2
plt.figure(figsize=(8,6))
plt.scatter(x1 , x2 )

plt.xlabel('Feature_1')
plt.ylabel('Feature_2')
```

Out[312]: Text(0, 0.5, 'Feature_2')



d. Data Cluster with multiple colors (e.g. 3 or 4 clusters using K-mean)

Clustering is the task of dividing the data points into several groups such that data points in the same groups are more similar to other data points in the same group than those in other groups. Clustering is a un-supervised learning. Unsupervised learning is a type of algorithm that learns patterns from untagged data. There is no output label in unsupervised learning.



First we will import KMeans from sklearn and create an object of KMeans. The object demands one parameter that is what is number of clusters. We did this for two values of K(number of clusters).

```
In [93]:  from sklearn.cluster import KMeans
          from sklearn.preprocessing import MinMaxScaler
```

```
In [95]:  km = KMeans(n_clusters=3)
```

Number Of Clusters = n_clusters = K = 3

As this is Un-Supervised Learning a new dataset without labels ‘y’ is created.

In [96]:
```
dataset_c = dataset.drop('y',axis = 1)
```

In [97]:
```
dataset_c
```

Out[97]:

	x1	x2
0	1000	0
1	1000	1
2	1000	2
3	1000	3
4	1000	4
...	...	...
1800	5000	356
1801	5000	357
1802	5000	358
1803	5000	359
1804	5000	360

1805 rows × 2 columns

Clusters are predicted on the base of features 'x1' and 'x2'.

In [98]:
```
y_predicted = km.fit_predict(dataset_c[['x1','x2']])
```

In [99]:
```
dataset_c['cluster'] = y_predicted
```

In [100]:
```
dataset_c.head()
```

Out[100]:

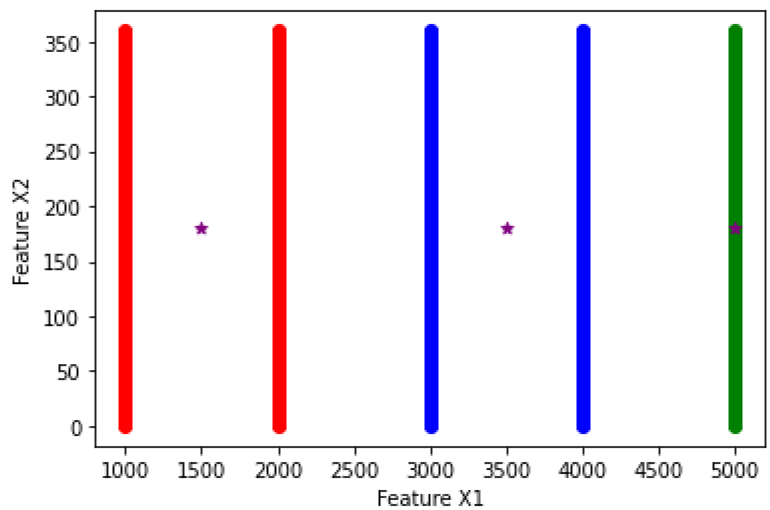
	x1	x2	cluster
0	1000	0	1
1	1000	1	1
2	1000	2	1
3	1000	3	1
4	1000	4	1

The data is divided on the base of clusters and stored into a new data frame. Scatter Plot of each data frame is made and assigned different color.

In [101]:
```
df_1 = dataset_c[dataset_c.cluster == 0]
df_2 = dataset_c[dataset_c.cluster == 1]
df_3 = dataset_c[dataset_c.cluster == 2]
```

In [102]:
```
plt.xlabel('Feature X1')
plt.ylabel('Feature X2')

plt.scatter(df_1['x1'] , df_1['x2'] , color = 'green')
plt.scatter(df_2['x1'] , df_2['x2'] , color = 'red')
plt.scatter(df_3['x1'] , df_3['x2'] , color = 'blue')

plt.scatter(km.cluster_centers_[:,0],km.cluster_centers_[:,1],color = 'purple' , marker='*' , label = 'centroid')
```

Final Visualization:



The Purple Stars are the centroids of clusters.

Now this task is repeated for K = 4. Number of Clusters = 4

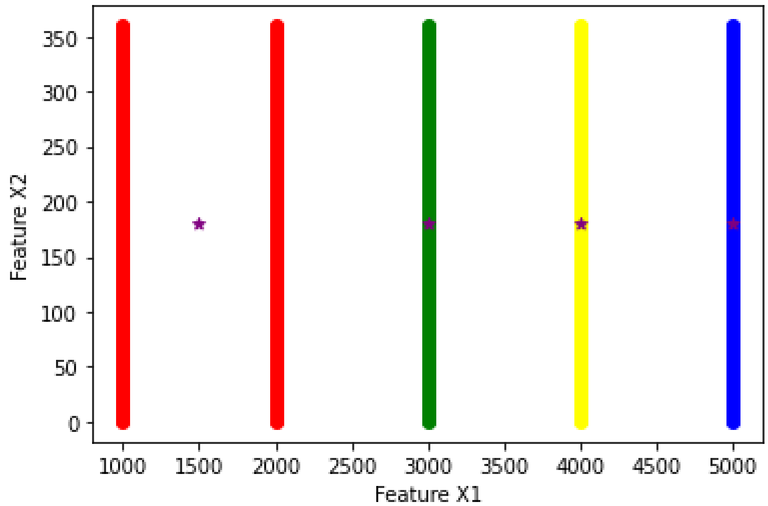
For Clusters, K = 4

In [103]:
```
km = KMeans(n_clusters=4)
dataset_c = dataset.drop('y',axis = 1)
y_predicted = km.fit_predict(dataset_c[['x1','x2']])
dataset_c['cluster'] = y_predicted
```

In [104]:
```
df_1 = dataset_c[dataset_c.cluster == 0]
df_2 = dataset_c[dataset_c.cluster == 1]
df_3 = dataset_c[dataset_c.cluster == 2]
df_4 = dataset_c[dataset_c.cluster == 3]
```

In [105]:
```
plt.xlabel('Feature X1')
plt.ylabel('Feature X2')

plt.scatter(df_1['x1'] , df_1['x2'] , color = 'green')
plt.scatter(df_2['x1'] , df_2['x2'] , color = 'red')
plt.scatter(df_3['x1'] , df_3['x2'] , color = 'blue')
plt.scatter(df_4['x1'] , df_4['x2'] , color = 'yellow')

plt.scatter(km.cluster_centers_[:,0],km.cluster_centers_[:,1],color = 'purple' , marker='*' , label = 'centroid')
```

Final Visualization:

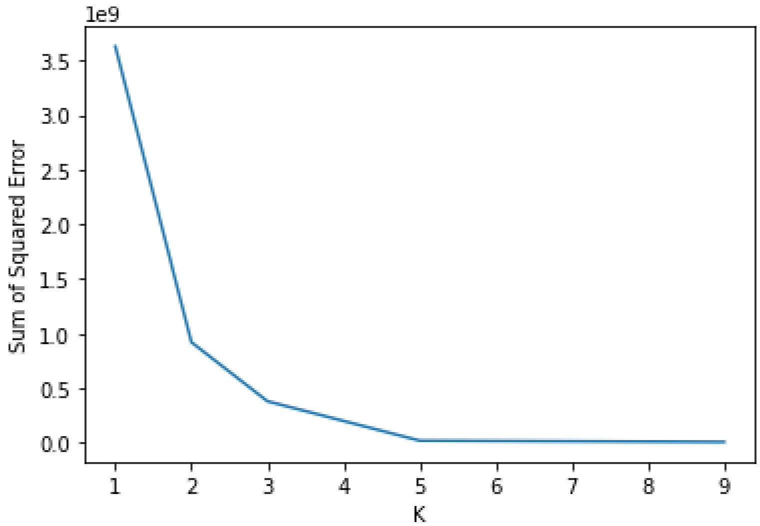


We can also find the best value of K (number of clusters) by elbow method in which sum of squared error is plotted against different number of K and the elbow region is declared as the best value.

In [106]:
```
k_rng = range(1,10)
SSE = []

for K in k_rng:
    km = KMeans(n_clusters=K)
    km.fit(dataset_c[['x1','x2']])
    SSE.append(km.inertia_)
```

In [107]:
```
plt.plot(k_rng,SSE)
plt.xlabel('K')
plt.ylabel('Sum of Squared Error')
```

Figure:



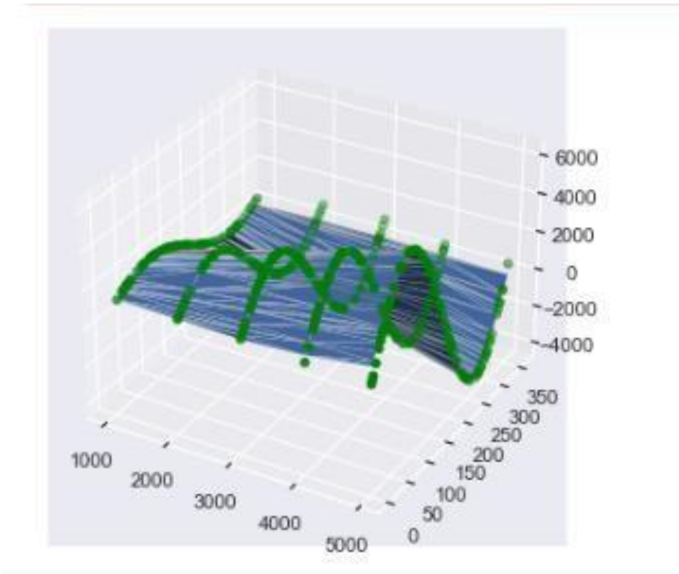
K = 5 will give zero error.

e. Plots 3D – surface for predicted data and put scatter points from given data on that surface to observe model quality

Code:

```
In [108]:  x1 = test_features['x1']
           x2 = test_features['x2']

           y1 = test_predictions
           y2 = test_labels

           fig = plt.figure()
           ax = Axes3D(fig)
           surf = ax.plot_trisurf(x1, x2, y1, linewidth=0.1)
           fig.colorbar(surf, shrink=0.5, aspect=5)

           #plt.show()
           #ax = plt.axes(projection ="3d")
           # Creating plot

           ax.scatter3D(x1, x2, y2, color = "green")
           plt.show()
```

Figure:



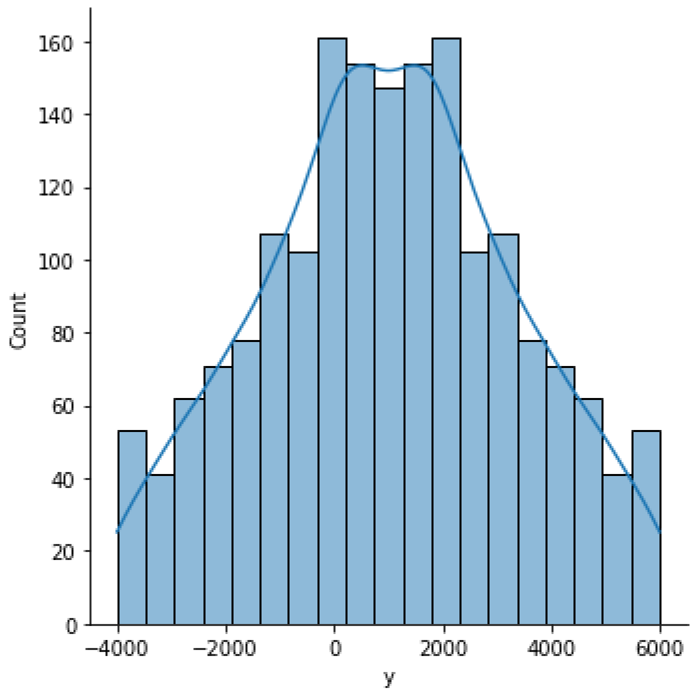
f. Plots to show data distribution

We are going to use Seaborn library for this task. Distribution Plots tell us about where most of our data lies.

Distribution Plot of Output ‘y’

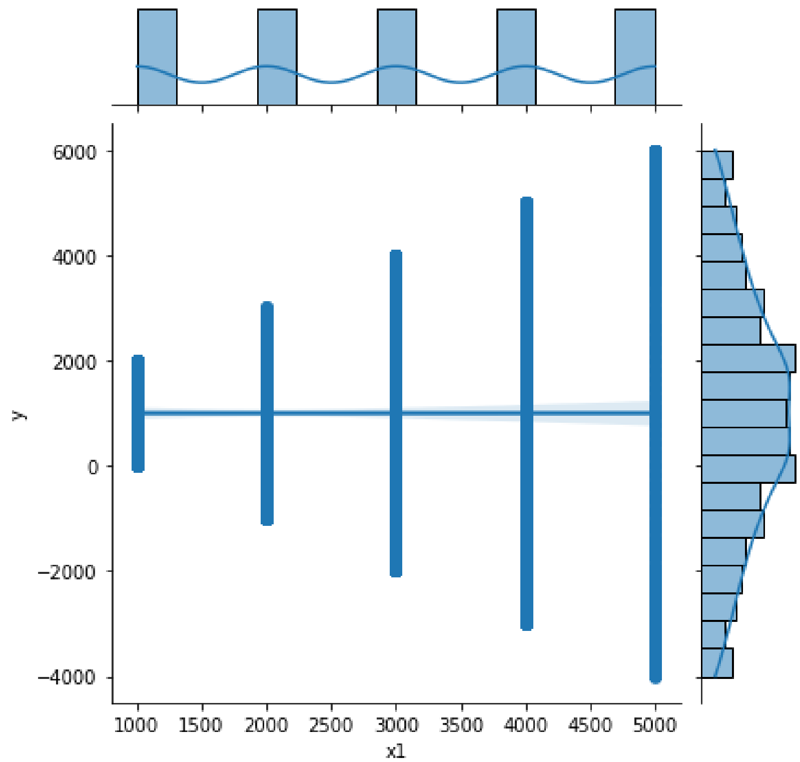
In [111]:
```
sns.displot(dataset['y'],kde = True)
```

Out[111]: <seaborn.axisgrid.FacetGrid at 0x1d94c17d760>



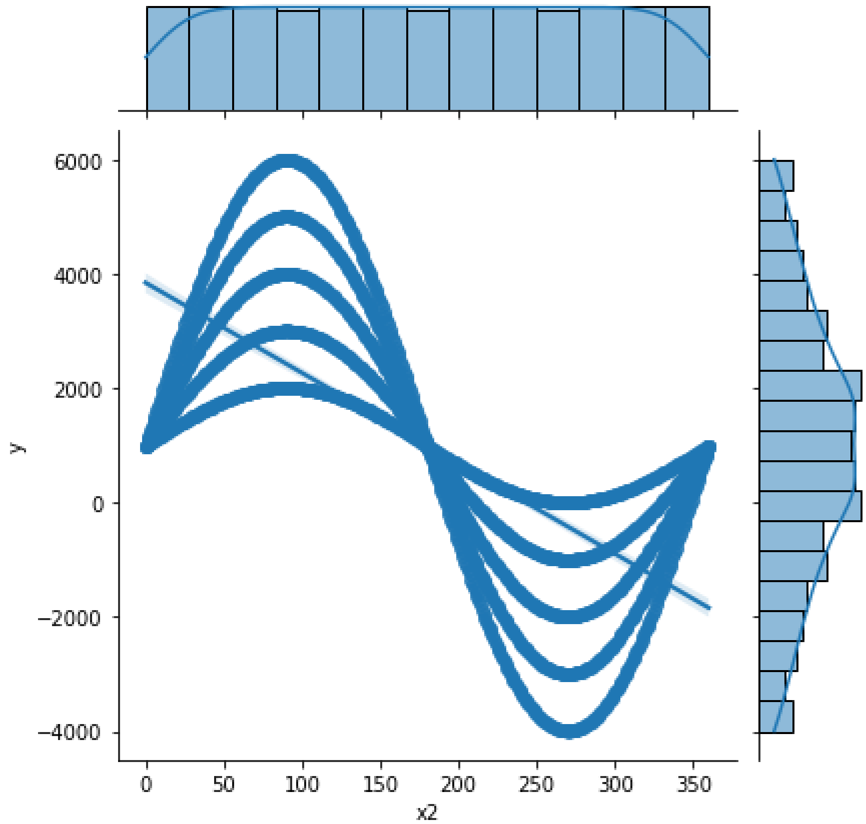
Joint Plot between the Feature 1 and Output:

```
In [112]: sns.jointplot(x = 'x1' , y = 'y' , data = dataset , kind = 'reg')
Out[112]: <seaborn.axisgrid.JointGrid at 0x1d94c1982e0>
```



The variable kind = reg means a regression line is fitted between their relationship.

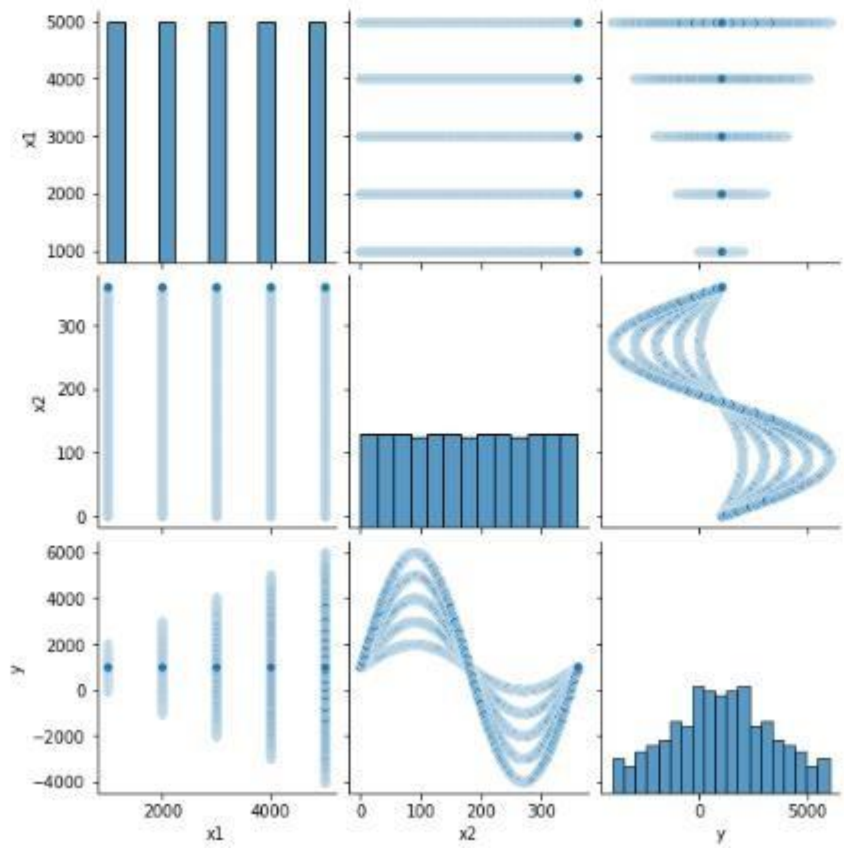
Joint Plot between the Feature 2 and Output:



Pair Plot:

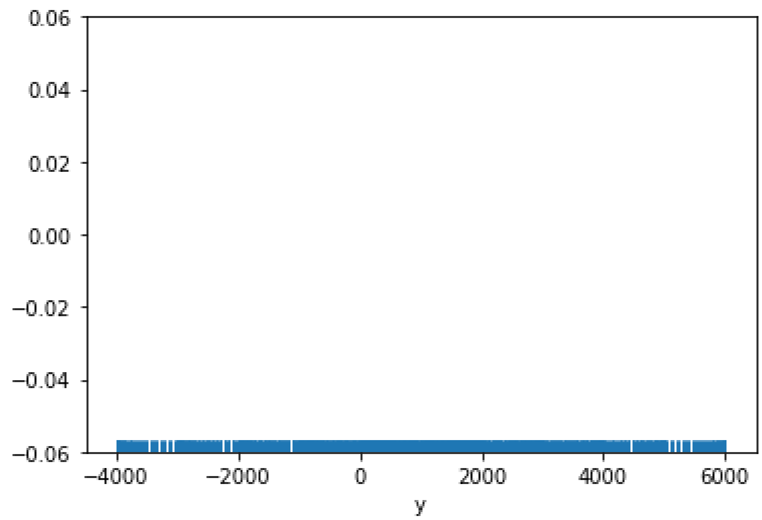
Pairplot is a technique in which each variable is plotted against each variable to check the relationship between them. The input to the pairplot is the whole dataset.

In [114]:
```
sns.pairplot(dataset)
```
Out[114]: <seaborn.axisgrid.PairGrid at 0x1d94c53e9d0>



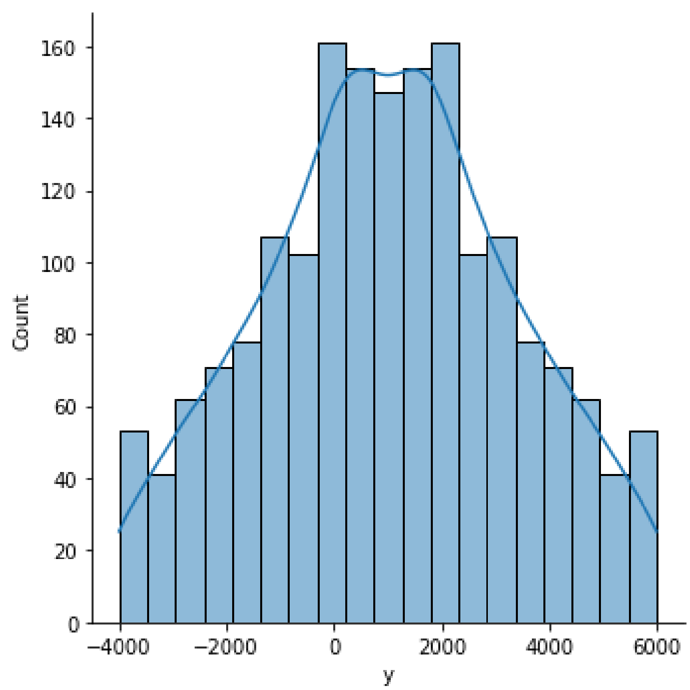
Rugplot:

In [115]:
```
sns.rugplot(dataset['y'])
```
Out[115]: <AxesSubplot:xlabel='y'>



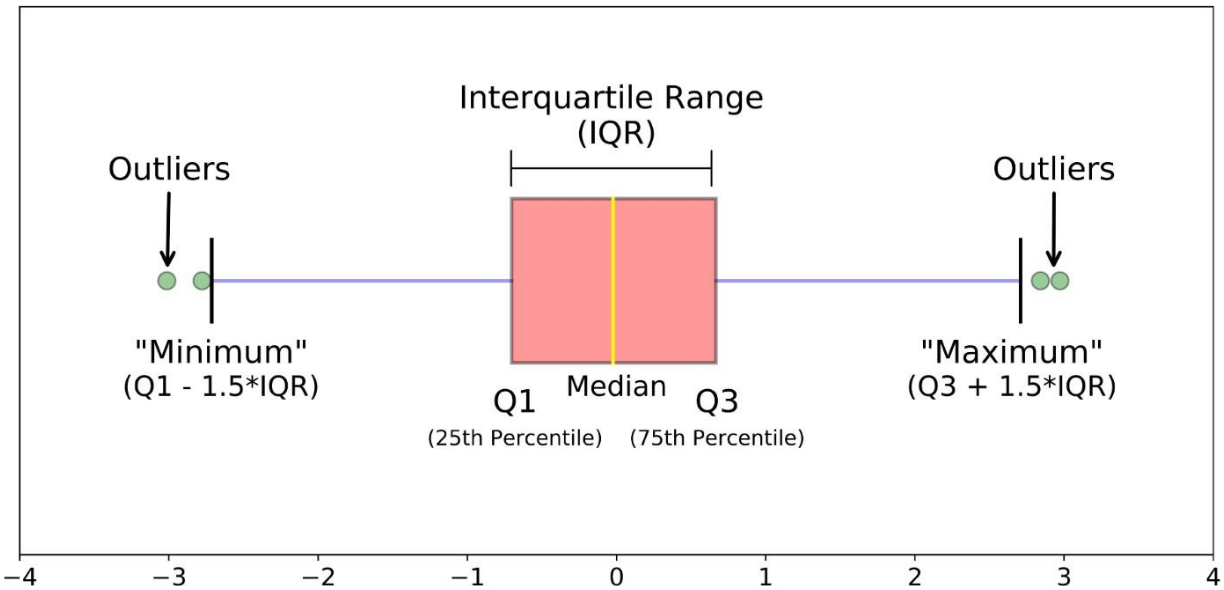
The RugPlot and the histogram plots are kind of similar. At each point in Rug Plot there is a uniform distribution. These all uniform distribution combine together to form hde.

In [111]:
```
sns.displot(dataset['y'],kde = True)
```
Out[111]: <seaborn.axisgrid.FacetGrid at 0x1d94c17d760>



g. Box-Plot showing statistics (mean, min, max, top 25% and 75%, etc.)

BoxPlot

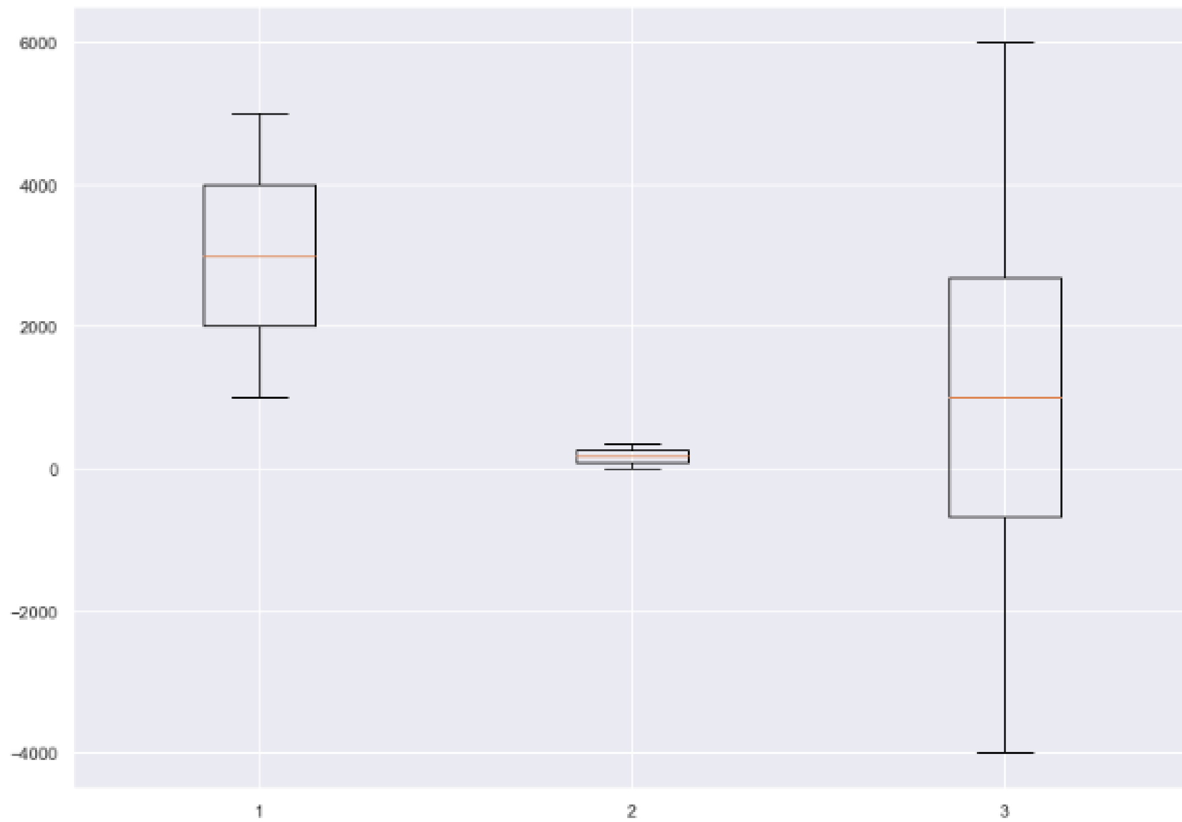


- The Line Signifies the Median
- The Box in the middle shows the start of 25 Percentile and end of 75 Percentile
- The Whiskers (Left - Right) show the Minimum Quartile and Maximum Quartile
- The Dots Represent the Outliers

Code:

```
In [117]:
x1 = dataset['x1']
x2 = dataset['x2']
y = dataset['y']

data = [x1, x2, y]

fig = plt.figure(figsize =(10, 7))

# Creating axes instance
ax = fig.add_axes([0, 0, 1, 1])

# Creating plot
bp = ax.boxplot(data)

# show plot
plt.show()
```

Boxplot:



The first one from left side is Feature 1 the middle one is Feature 2 and the last from left side is Output 'y '.

Individual BoxPlot of Feature

h. Covariance matrix / Correlation matrix

For Covariance / Correlation Matrix correlation between each variable is found out and with heatmap it is plotted.

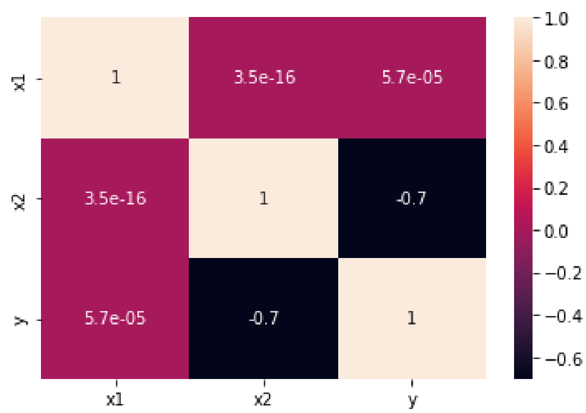
In [118]:
```
dataset
```
Out[118]:

	x1	x2	y
0	1000	0	1000
1	1000	1	1017
2	1000	2	1034
3	1000	3	1052
4	1000	4	1069
...	...	...	...
1800	5000	356	651
1801	5000	357	738
1802	5000	358	825
1803	5000	359	912
1804	5000	360	999

1805 rows × 3 columns

In [119]:
```
PC = dataset[['x1','x2','y']].corr()
```
In [120]:
```
PC
```
Out[120]:

	x1	x2	y
x1	1.000000e+00	3.459433e-16	0.000057
x2	3.459433e-16	1.000000e+00	-0.702316
y	5.654649e-05	-7.023155e-01	1.000000

In [121]:
```
HM = sns.heatmap(PC , annot=True)
```


Deliverable Number 8

Please, prepare dummy data using python to create similar plots using matplotlib and plotly.

Creation of Dummy Dataset

For creation of Dummy Dataset we took help from the internet and learned how to create a new dataset. We are going to create a Car_Data with the help of random function.

```
In [123]: cars = ['Audi', 'BMW', 'Kia', 'Honda', 'Skoda']

          models = {0:['A1', 'A3', 'A4', 'A5', 'A6'],
                    1:['1 Series','3 Series','5 Series','7 Series','X5'],
                    2:['Rio','Spotage','Sorento','Soul','Ceed'],
                    3:['Accord','Civic','Jazz','Fit'],
                    4:['Citigo','Scala','Octavia']}

          prices =  {0:[30000, 32000, 55000, 60000, 70000],
                    1:[35000, 50000, 60000, 70000, 65000],
                    2:[19000, 40000, 58000, 37000, 25000],
                    3:[32000, 25000, 30000, 15000],
                    4:[15000, 20000, 35000]}

          years = [2021, 2020, 2019, 2018, 2017]

          salespeople = ['Talha', 'Arslan', 'Hassan', 'Ali', 'Subas', 'Bilal']
```

We made some rules and according to the rules we then randomly selected the values from these dictionaries to create a new data frame.

```
In [124]: car_data = pd.DataFrame()

          for i in range(0, 250):
              make = random.choice(cars)
              modelindex = cars.index(make)
              model = random.choice(models.get(modelindex))
              priceindex = models.get(modelindex).index(model)
              price = prices.get(modelindex)[priceindex]
              year = random.choice(years)
              deval = (2021 - year) * .1
              purchase_price = round(price * (1 - (deval + random.uniform(-.05, .1))),2)
              repair_cost = round(purchase_price * random.uniform(.01, .1),2)
              sales_price = round((purchase_price + repair_cost) * random.uniform(.95, 1.1),2)
              stock_days = random.randint(1, 90)
              date_sold = datetime.datetime(2021, 9, 1) + datetime.timedelta(days = random.randint(0, 30))
              profit = sales_price - purchase_price - repair_cost
              salesperson = random.choice(salespeople)

              linedict = {'make': [make], 'model': [model], 'year':[year], 'purchase_price':[purchase_price],'repair_cost'
                          :[repair_cost], 'sales_price':[sales_price], 'profit':[profit],'stock_days':[stock_days],
                          'date_sold':[date_sold], 'sales_person':[salesperson]}
              line = pd.DataFrame(linedict)
              car_data = pd.concat([car_data, line])
```

After creation of dataset we saved our data into a csv file.

```
In [125]: car_data
```

Out[125]:

	make	model	year	purchase_price	repair_cost	sales_price	profit	stock_days	date_sold	sales_person
0	Honda	Accord	2018	23444.10	2246.93	27050.34	1359.31	89	2021-09-24	Ali
0	Audi	A5	2018	39493.22	2941.93	40650.46	-1784.69	7	2021-09-20	Hassan
0	Honda	Jazz	2019	22147.72	548.13	23076.17	380.32	74	2021-09-12	Hassan
0	Honda	Civic	2018	15285.23	1281.42	18124.68	1558.03	62	2021-10-01	Hassan
0	BMW	3 Series	2020	46011.93	2883.53	52889.78	3994.32	56	2021-09-03	Arslan
...	...	...	...	...	...	...	...	...	...	...
0	Honda	Accord	2019	26097.66	973.58	26806.74	-264.50	5	2021-09-22	Arslan
0	Audi	A1	2020	27569.46	453.71	27786.97	-236.20	83	2021-09-20	Ali
0	Audi	A5	2020	48845.88	1318.82	49528.43	-636.27	32	2021-09-28	Arslan
0	Honda	Jazz	2021	28037.81	692.33	27820.50	-909.64	60	2021-09-09	Arslan
0	Honda	Civic	2017	13686.39	1006.48	15928.65	1235.78	12	2021-09-03	Bilal

250 rows × 10 columns

```
In [126]: car_data.to_csv('Sales_Data_Sept21.csv',index = False)
```

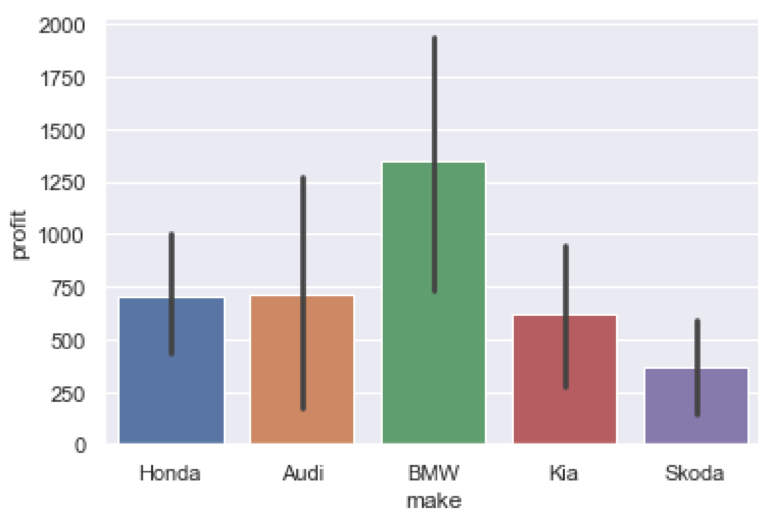
```
In [127]: car_data = pd.read_csv('Sales_Data_Sept21.csv')
```

Now, we are going to create different plots for this dataset.

Bar Plots

Bar Plot between Make and Profit

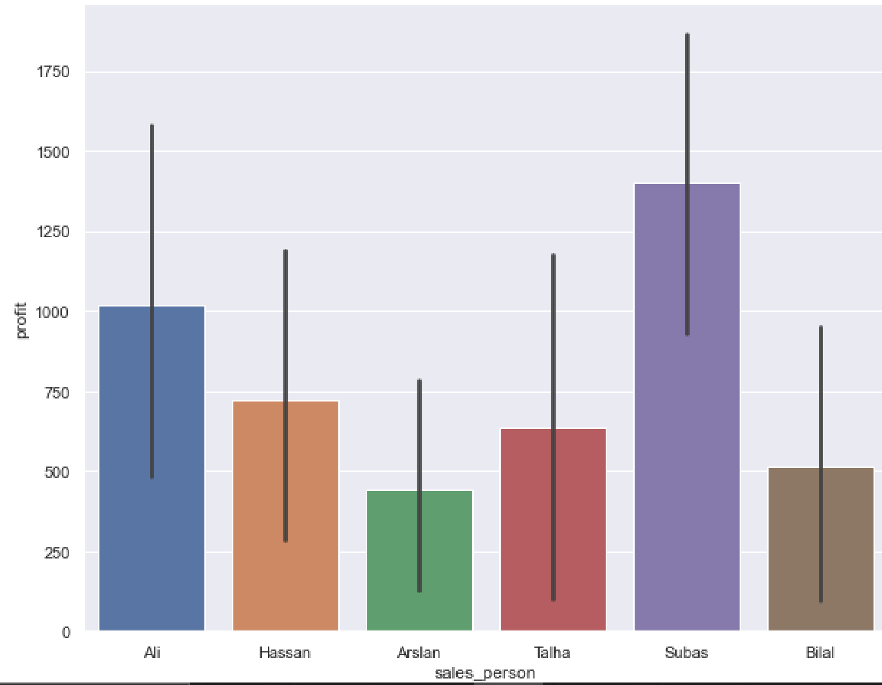
```
In [130]: ax = sns.barplot(x = 'make' , y = 'profit' , data = car_data)
```



We can see from the graph that BMW cars make the most Profit. This kind of information can be extracted from Graphs.

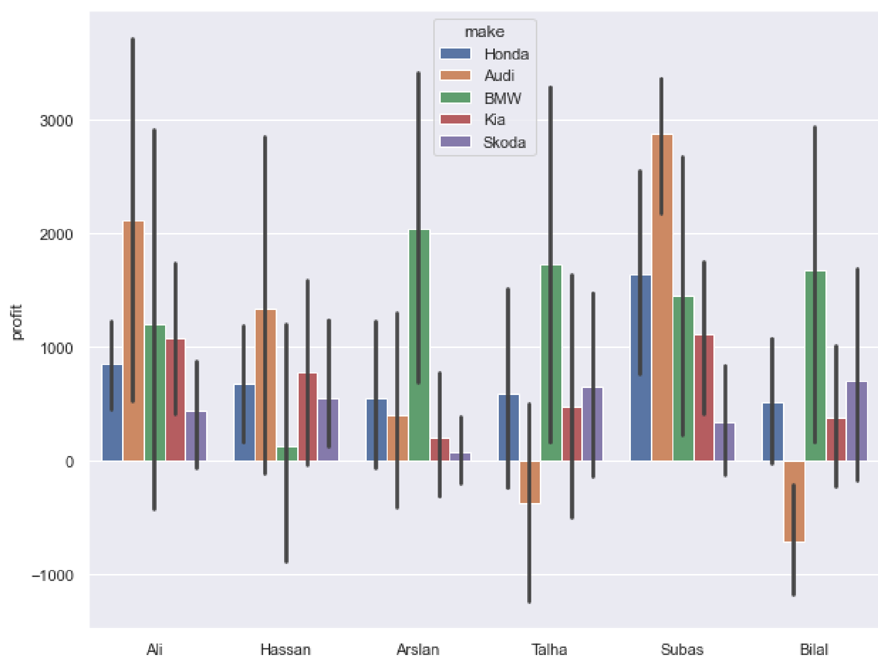
Bar Plot between Sales_Person and Profit

```
In [132]: ax = sns.barplot(x = 'sales_person' , y = 'profit' , data = car_data)
```



The graph below is also between sales_person and profit but this time we have distinguish it more on the basis of car's make.

```
In [133]: ax = sns.barplot(x = 'sales_person' , y = 'profit' , hue = 'make' ,data = car_data)
```

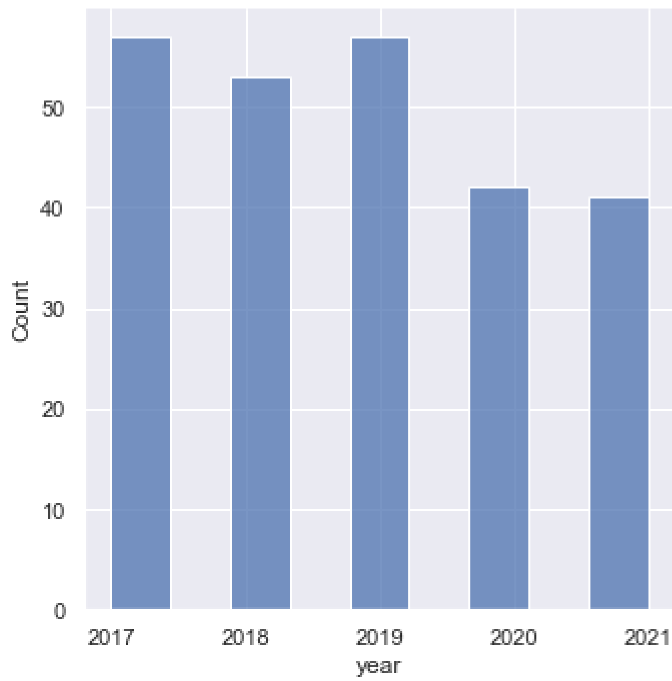


Histograms

This Car's dataset is of 250 rows and if we want to see that where our data lies we can plot histograms.

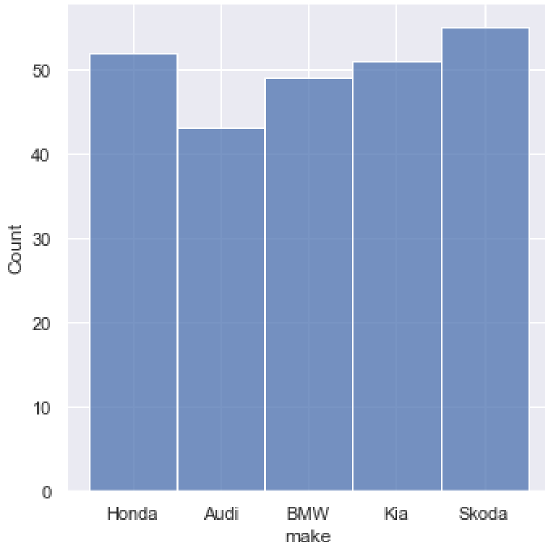
Histogram of Year

```
In [135]:  sns.displot(car_data['year'])
Out[135]: <seaborn.axisgrid.FacetGrid at 0x1d96ba91340>
```



Histogram of Car's Make

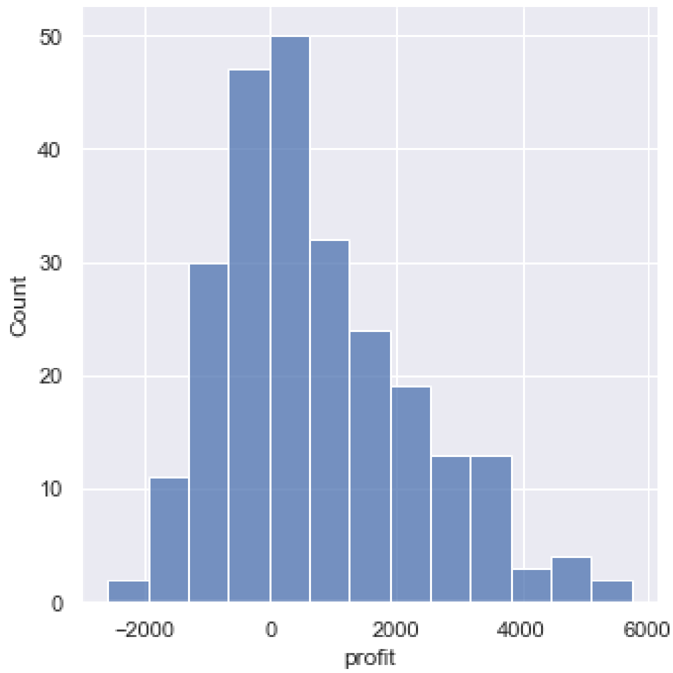
In [136]:
```
sns.displot(car_data['make'])
```
Out[136]: <seaborn.axisgrid.FacetGrid at 0x1d96b8bd790>



Histogram of Profit

```
In [138]: sns.displot(car_data['profit'])
```

```
Out[138]: <seaborn.axisgrid.FacetGrid at 0x1d96b9507f0>
```

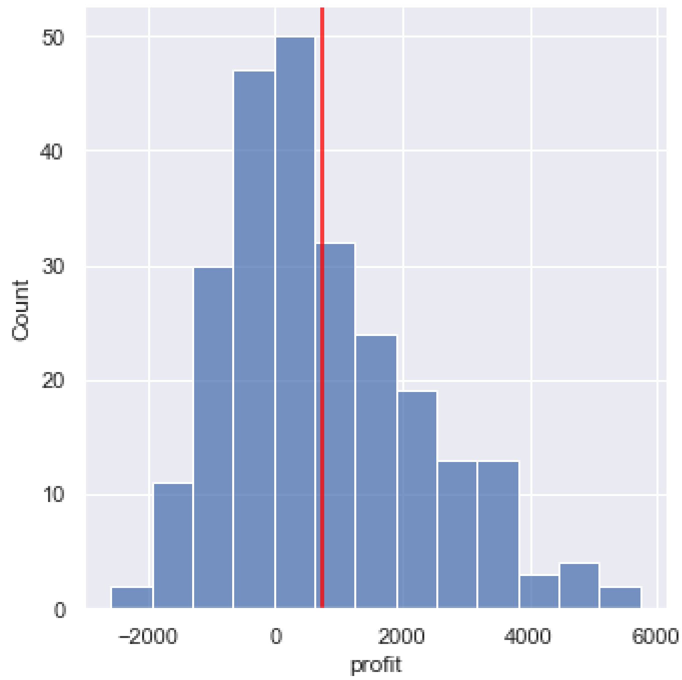


The Profit's Histogram is Right Skwed because it's mean is towards the right side

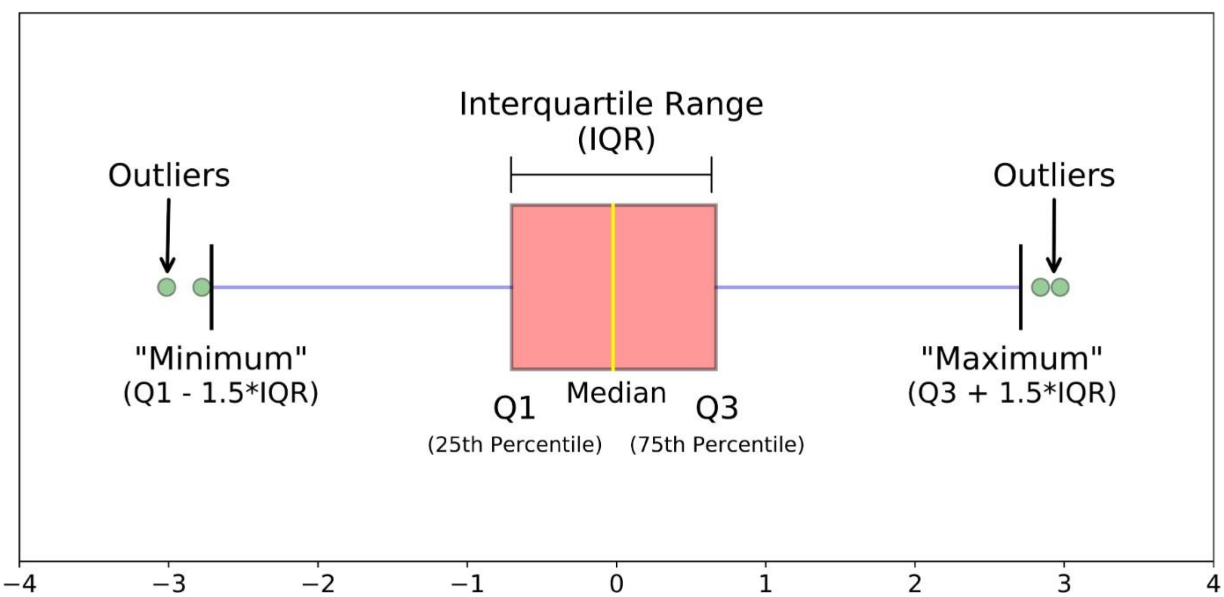
We can also plot the mean on the graph.

```
In [139]:  profit_mean = car_data['profit'].mean()
           sns.displot(car_data['profit'])
           plt.axvline(profit_mean,0,1 , color = 'red')
```

```
Out[139]: <matplotlib.lines.Line2D at 0x1d96bbf8100>
```



Box Plot

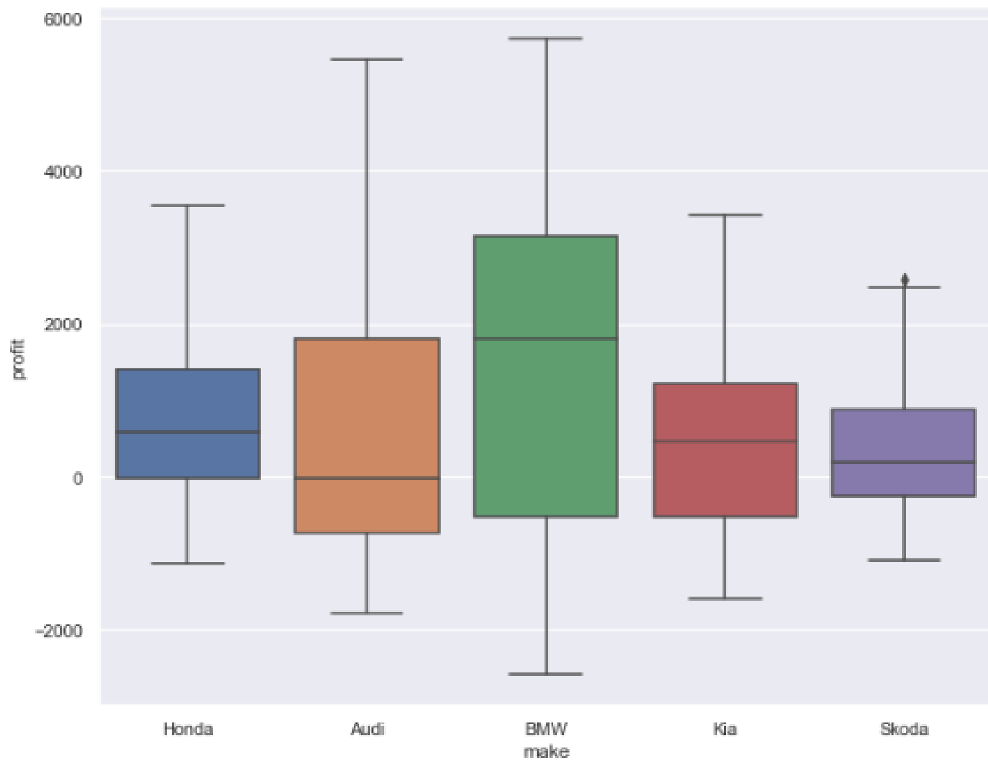


- The Line Signifies the Median

- The Box in the middle shows the start of 25 Percentile and end of 75 Percentile
- The Whiskers (Left - Right) show the Minimum Quartile and Maximum Quartile
- The Dots Represent the Outliers

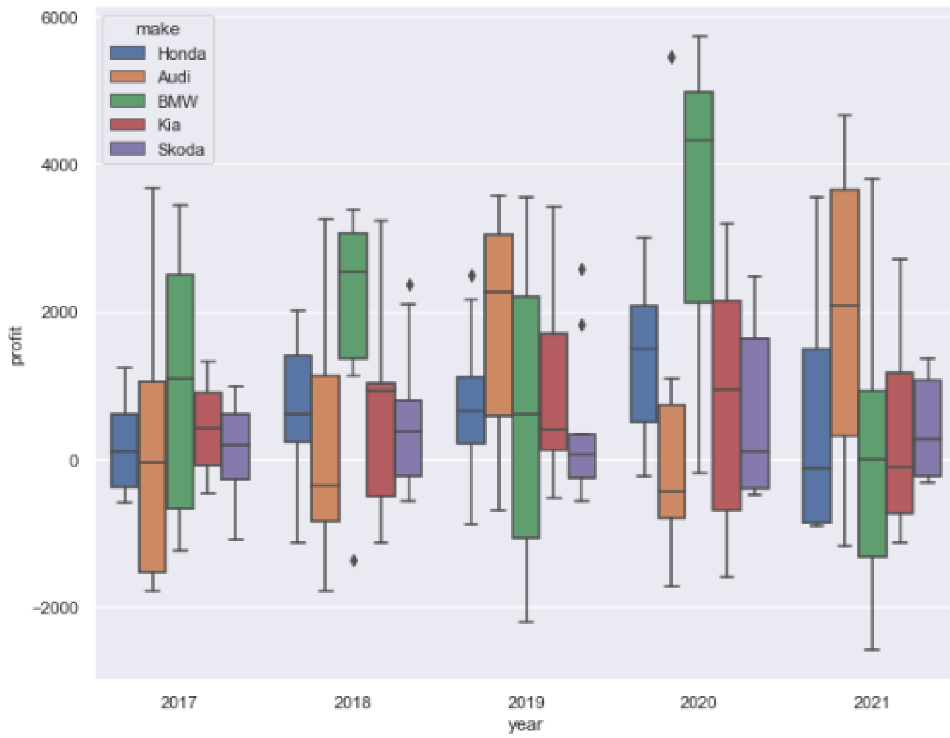
Box Plot between make and profit

In [142]:
```
# Investigating the Relationship between Make of the Car and Profit
ax = sns.boxplot(x = 'make', y = 'profit' , data = car_data)
```



Box Plot between year and profit

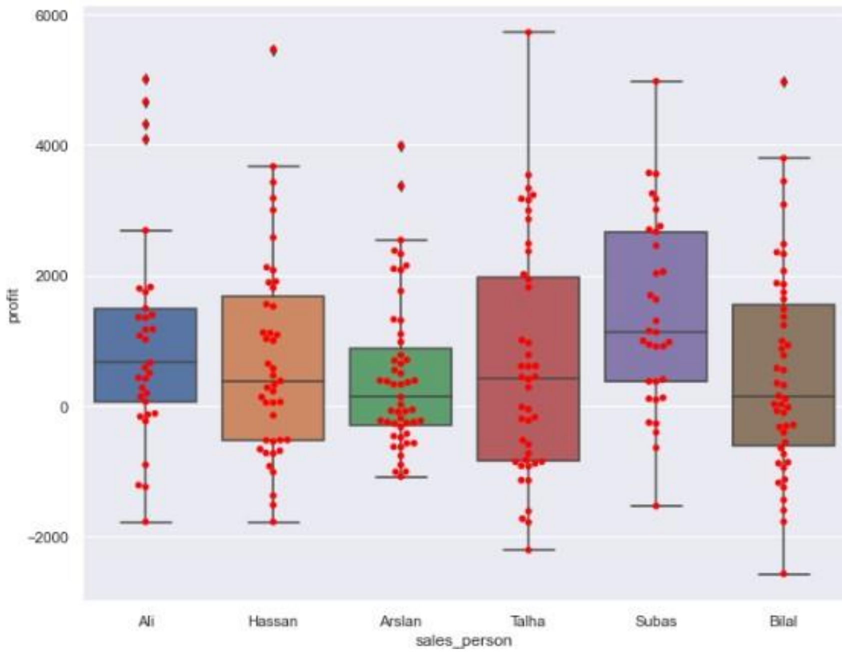
In [143]:
```
ax = sns.boxplot(x = 'year', y = 'profit' , hue = 'make', data = car_data)
```



In this graph we have investigated the relationship between year and profit. For example in which year we have had the most profit and then we can also differentiate it on the basis of make of the vehicle.

Box Plot between sales_person and profit

```
In [144]: ax = sns.boxplot(x = 'sales_person', y = 'profit' , data = car_data)
          ax = sns.swarmplot(x = 'sales_person', y = 'profit' , data = car_data , color = 'red')
```

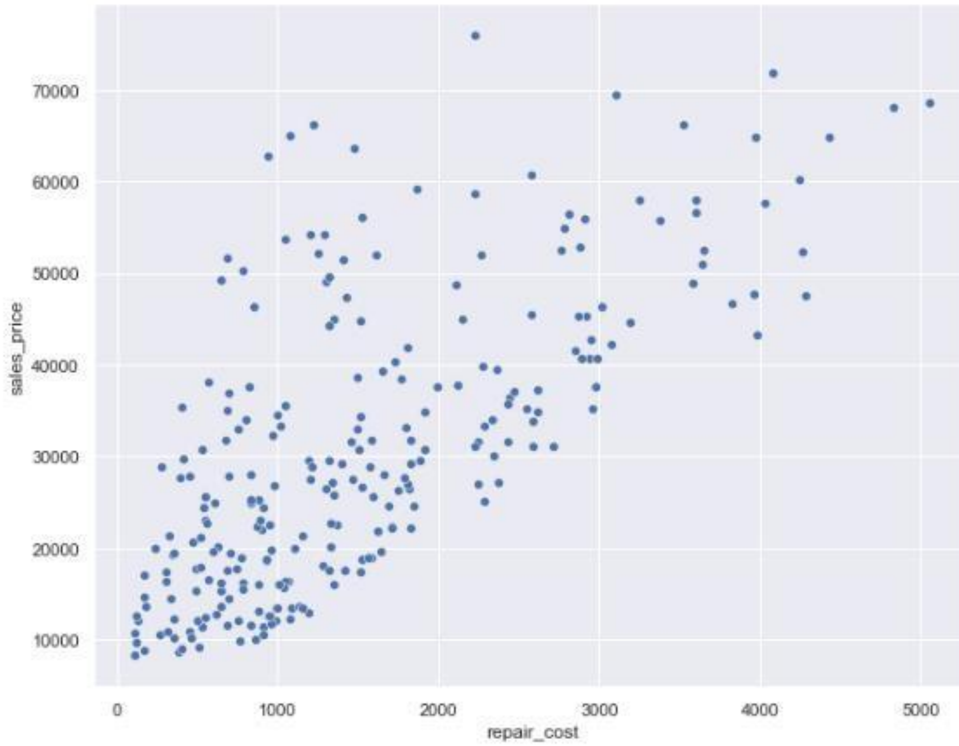


In this graph we plotted the relationship between sales_person and profit and we also plotted each dataset point.

Scatter Plots

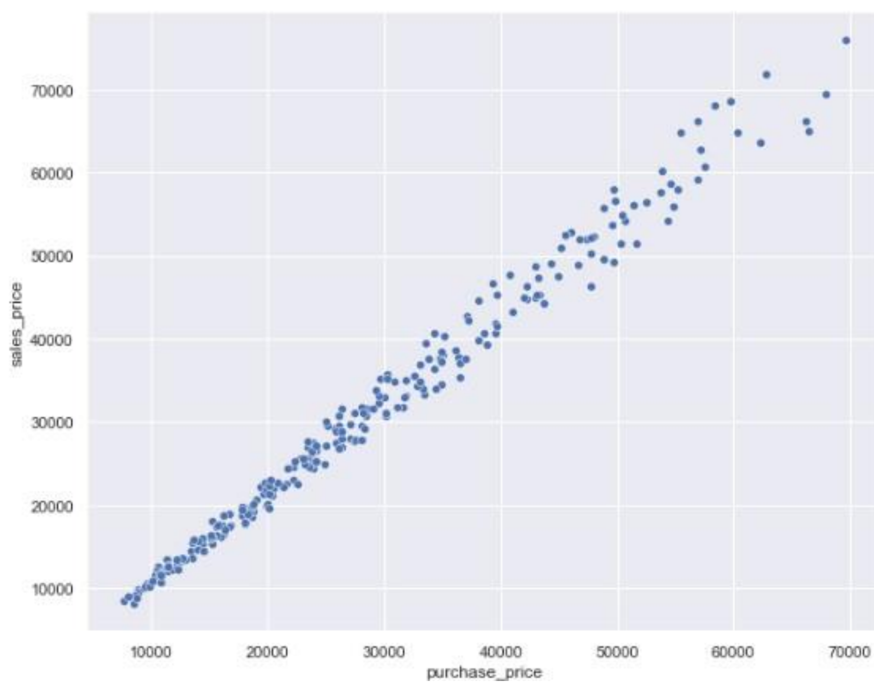
Scatter Plot between repair_cost and sales_price

In [146]:
```
ax = sns.scatterplot(x = 'repair_cost' , y = 'sales_price' , data = car_data)
```



Scatter Plot between purchase_price and sales_price

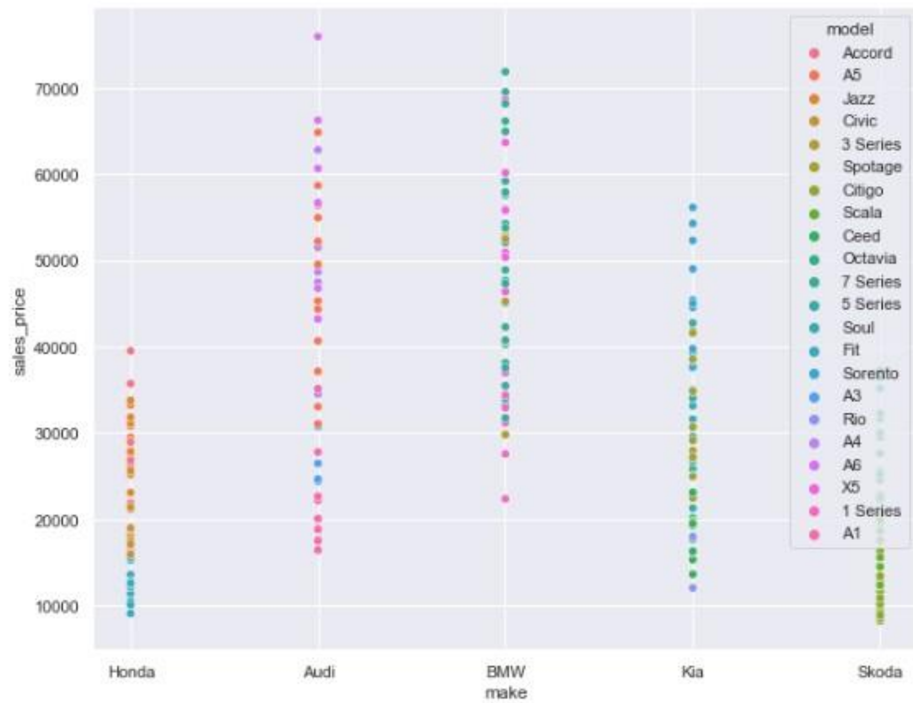
In [147]:
```
ax = sns.scatterplot(x = 'purchase_price' , y = 'sales_price' , data = car_data)
```



The sales_price and purchase_price have a linear relationship among them.

Scatter Plot between make and sales_price

```
In [150]: ax = sns.scatterplot(x = 'make' , y = 'sales_price' , hue = 'model', data = car_data)
```



This is the graph in which we have investigated the relationship between car's make and sales_price and we have differentiated them also on the basis of car's model.

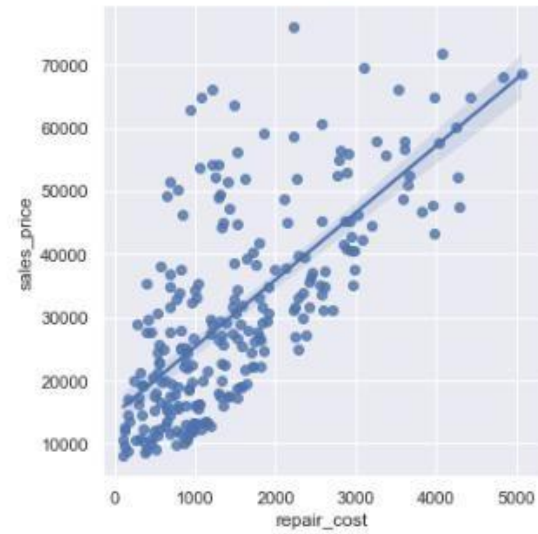
LM Plots

LM Plots are the scatter plots in which linear regression line is fitted among the quantities.

LM Plot between repair_cost and sales_price

In [152]:

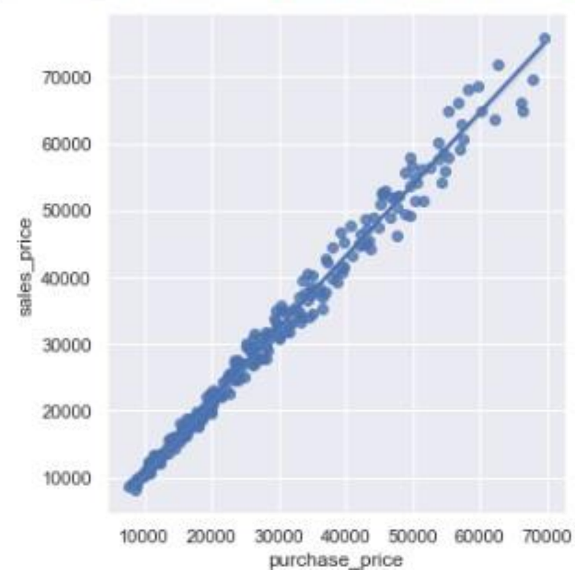
```
#Relationship between Repair Cost and Profit
ax = sns.lmplot(x = 'repair_cost' , y = 'sales_price' , data = car_data)
```



LM Plot between purchase_price and sales_price

In [153]:

```
ax = sns.lmplot(x = 'purchase_price' , y = 'sales_price' , data = car_data)
```

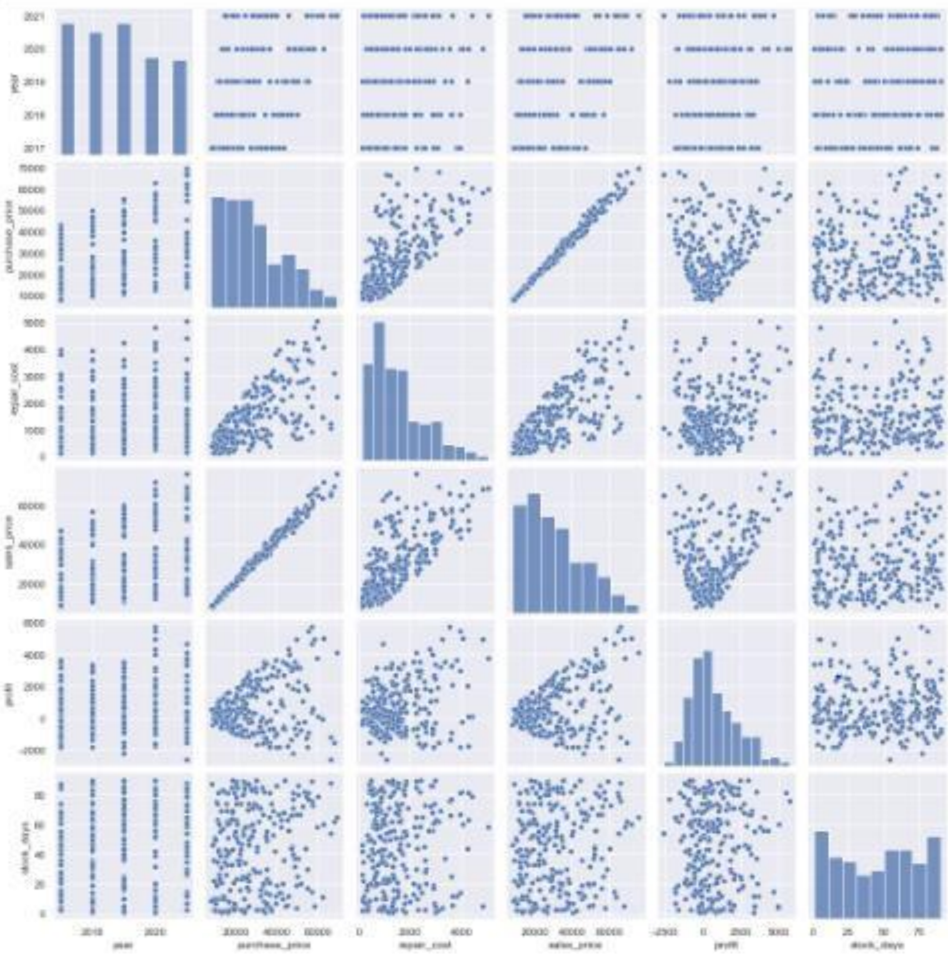


Pair Plot

Pairplot is a technique in which each variable is plotted against each variable to check the relationship between them. The input to the pairplot is the whole dataset.

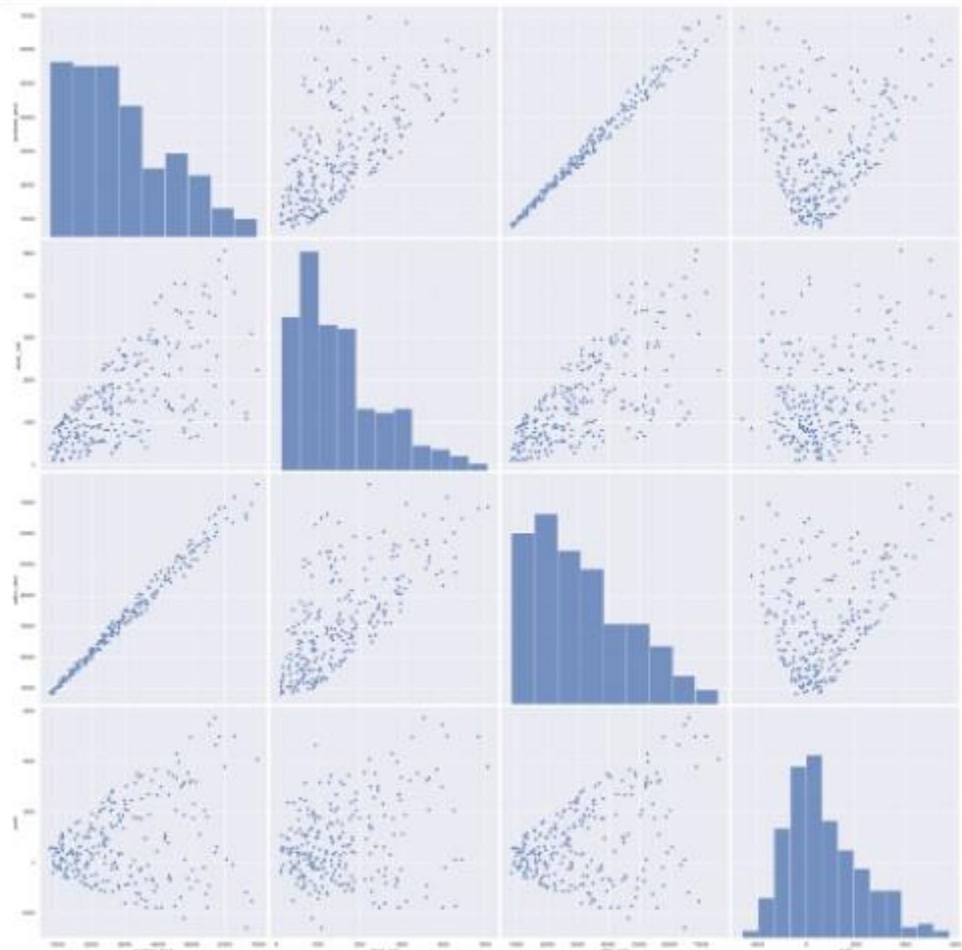
In [156]:

```
ax = sns.pairplot(car_data)
```



We can also draw the pair plot between the selected variables of dataset.

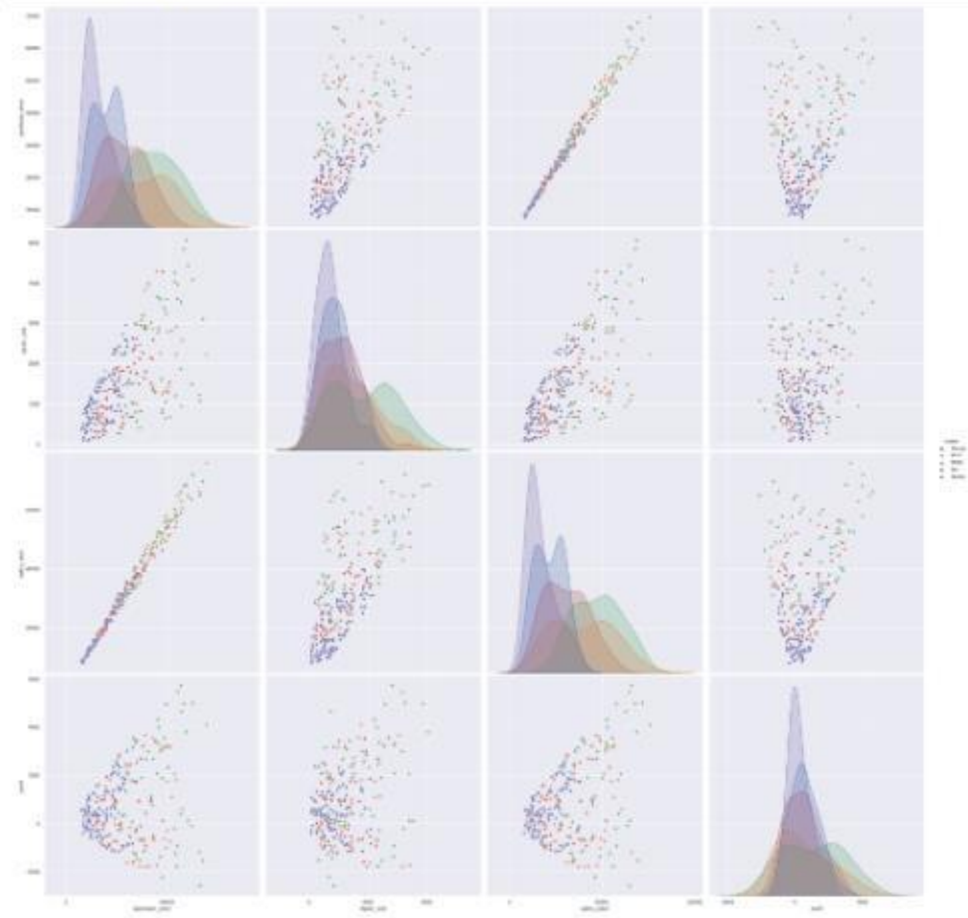
```
In [157]: ax = sns.pairplot(car_data[['purchase_price', 'repair_cost', 'sales_price',
                 'profit']] , height = 7)
```



This is the pair plot among the purchase_price, repair_cost, sales_price and profit. In the same graph if we differentiate them more on the basis of make we can add car’s make as hue element.

In [158]:

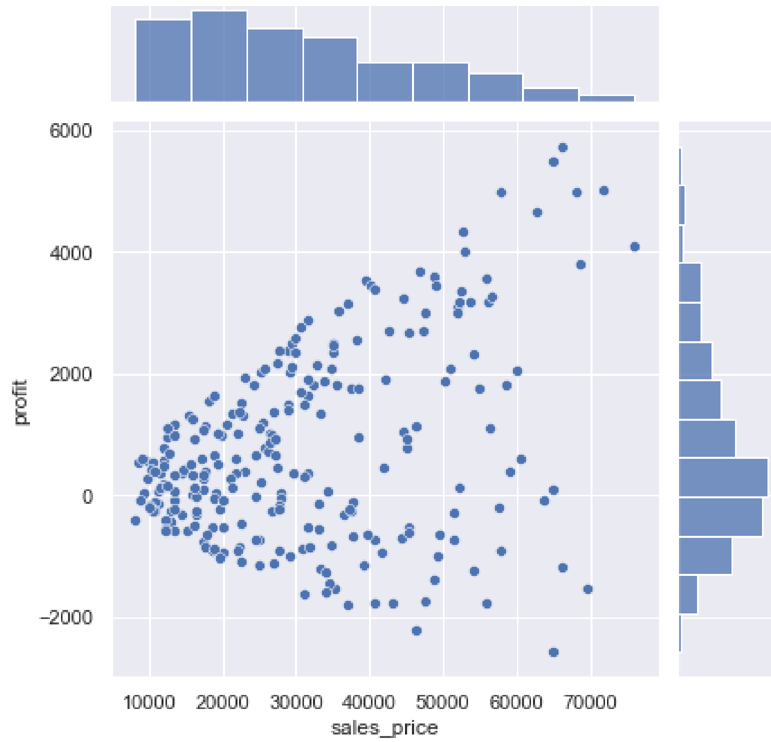
```
ax = sns.pairplot(car_data[['purchase_price', 'repair_cost', 'sales_price',
        'profit', 'make']] , height = 7 , hue = 'make')
sns.set(font_scale = 3)
```



Joint Plots

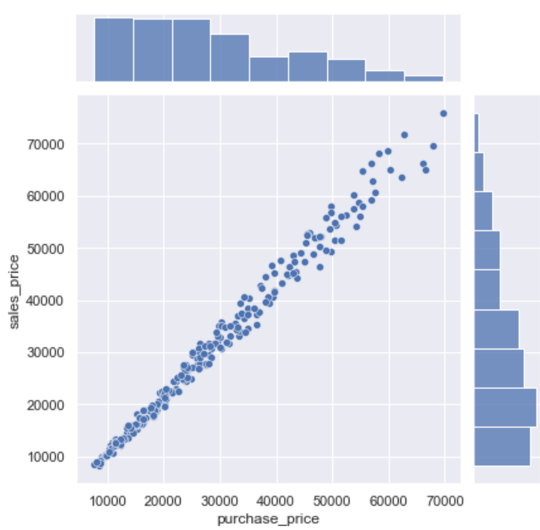
Joint Plot among sales_price and profit

```
In [159]: sns.set()
          ax = sns.jointplot(x = 'sales_price' , y = 'profit' , data = car_data )
```



Joint Plot among purchase_price and sales_price

```
In [162]: ax = sns.jointplot(x = 'purchase_price' , y = 'sales_price' , data = car_data )
```



Covariance Matrix

Covariance can be found between only the numerical quantities and to find covariance we are using a built in function corr().

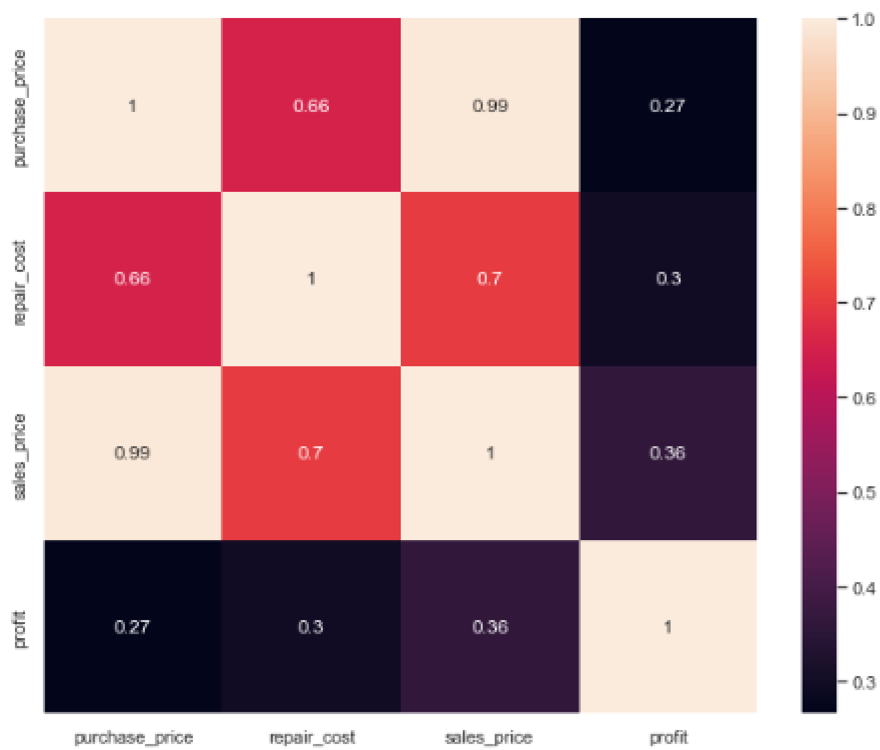
In [163]:
```
PC = car_data[['purchase_price', 'repair_cost', 'sales_price',
         'profit']].corr()
```

In [164]:
```
PC
```

Out[164]:

	purchase_price	repair_cost	sales_price	profit
purchase_price	1.000000	0.655342	0.993532	0.266055
repair_cost	0.655342	1.000000	0.700777	0.297696
sales_price	0.993532	0.700777	1.000000	0.363013
profit	0.266055	0.297696	0.363013	1.000000

In [165]:
```
HM = sns.heatmap(PC , annot = True)
```



Violin Plots

A violin plot is a hybrid of a box plot and a kernel density plot, which shows peaks in the data. It is used to visualize the distribution of numerical data. Unlike a box plot that can only show summary statistics, violin plots depict summary statistics and the density of each variable.

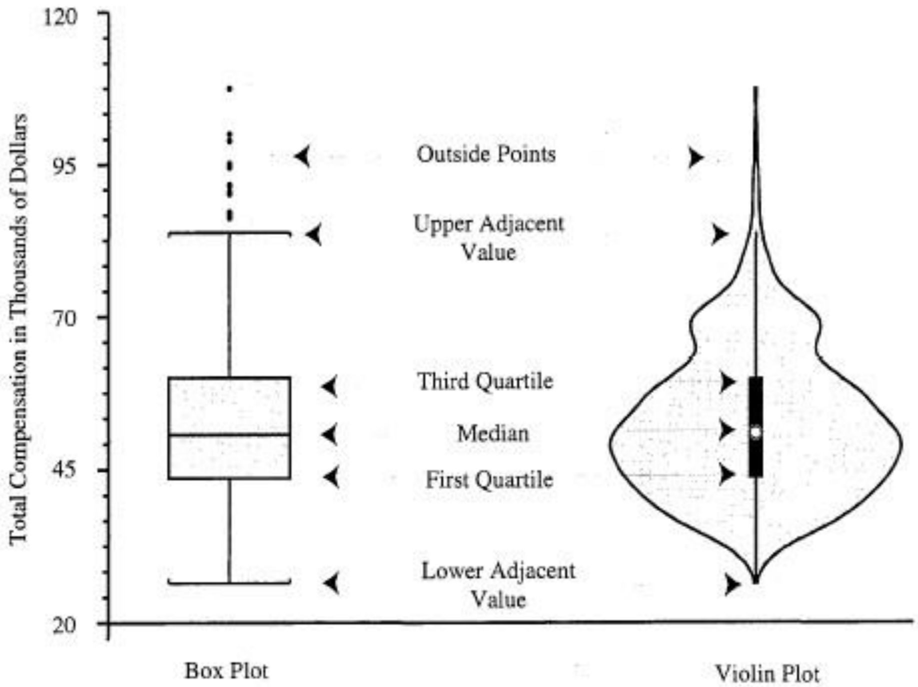
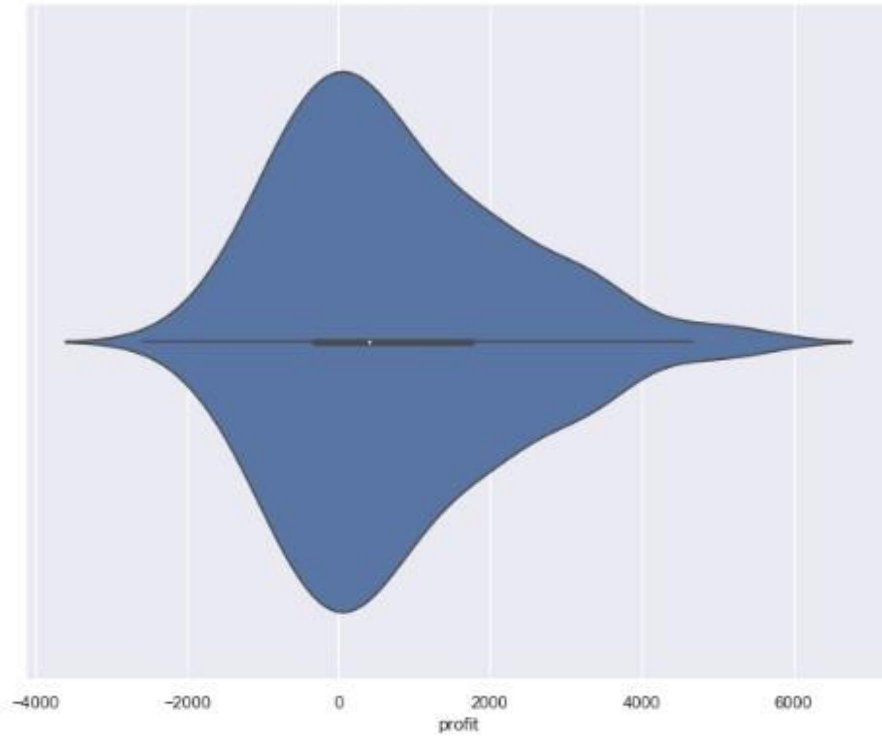


Figure 1. Common Components of Box Plot and Violin Plot. Total compensation for all academic ranks.

Violin Plot of Profit:

In [166]:

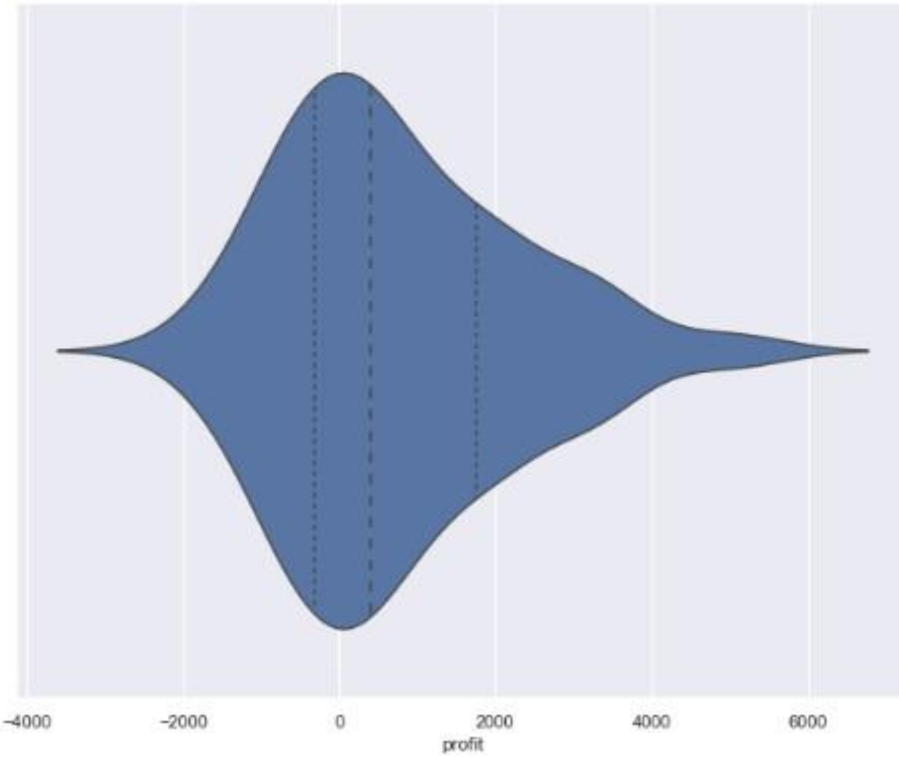
```
ax = sns.violinplot(x = car_data['profit'], inner = 'box')
```



Violin Plot of Profit with quartile:

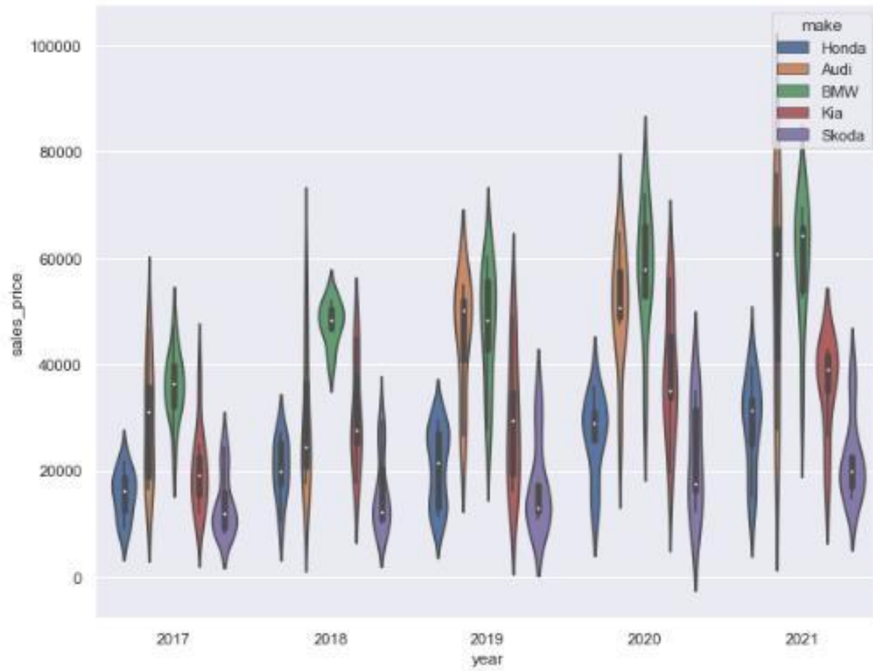
In [167]:

```
ax = sns.violinplot(x = car_data['profit'], inner = 'quartile')
```



Violin Plot between year and sales_price and ‘make’ as hue

In [169]:
```
ax = sns.violinplot(x = 'year' , y = 'sales_price' ,hue='make', data = car_data)
sns.set(rc={'figure.figsize':(20,16)})
```



Violin Plot between year and sales_price and scaling based on count

In [335]:

```
ax = sns.violinplot(x = 'year' , y = 'sales_price' , data = car_data, scale = 'count')
```