DATA ANALYSIS AND MACHINE LEARNING

EE514 ASSIGNMENT

NAME : RAFAELA GRACIA NONGRUM.

STUDENT NUMBER : 20210341

PROGRAMME : MENG IN ELECTRONIC AND COMPUTER ENGINEERING

Part 1

Aim : To predict human activity by using sensor data based on the Extrasensory dataset.

The dataset was downloaded, extracted and uploaded to Google Drive. The drive was then mounted and the provided Starter Code Notebook was run through once, to train on one user and test on a different user. It resulted in a training accuracy of 97.50%. To further examine the accurate prediction, the balanced accuracy was calculated as well which was found to be 83.54%.

IMPROVING THE TEST SET

Five different users were then further selected to individually test on each and the mean and variance for each was evaluated.

As per the question requirement, the users were appended into a single test set during which it was observed that the mean and variance had reduced as compared to just a single user. It resulted in the following :

	timestamp	raw_acc:magnitude_stats:mean	raw_acc:magnitude_stats:std	raw_acc:magnitude_stats:moment3	raw_acc:magnitu
Mean	1.442819e+09	1.015212	0.052990	0.055204	
variance	7.671798e+12	0.003520	0.016954	0.024955	

2 rows × 278 columns

CODE SNIPPET FOR APPENDING USERS:

```
a1= df_test1.append(df_test2)
a2= a1.append(df_test3)
a3= a2.append(df_test4)
a4= a3.append(df_test5)
pd.DataFrame([a4.mean(), a4.var()], index= ['Mean', 'variance'])
```

VALIDATION DATA AND INCREASED TRAINING DATA

For the next bit of the assignment, for the purpose of combining different users into a train and validation set each, a new colaboratory file was run using information from the previous starter code notebook. The notebook was saved and named ‘EE514_Part1’.

The drive was mounted and path was given for my dataset, which was saved under /content/drive/MyDrive/Extrasensory/data. Ten random users were selected for the purpose of making my model. The ten users were then appended together and stored to a variable ‘train_data’. By printing it we can see the output which consists of 73318 rows and 278 columns.

	timestamp	raw_acc:magnitude_stats:mean	raw_acc:magnitude_stats:std	raw_acc:magnitude_stats:moment3	raw_acc:magnitude_stats:moment4
0	1464121019	0.998324	0.002305	-0.001051	0.003047
1	1464121067	0.998647	0.052570	-0.038628	0.162579
2	1464121796	1.004814	0.002468	0.000855	0.003526
3	1464121851	1.003934	0.008411	0.011053	0.027717
4	1464121919	1.010905	0.004780	-0.004857	0.007757
...	...	...	...	...	...
8725	1444069057	1.006382	0.092322	0.142640	0.262679
8726	1444069117	1.003770	0.058084	0.067960	0.138514
8727	1444069177	1.002452	0.005126	0.009512	0.016179
8728	1444069237	1.003470	0.001823	0.001009	0.002387
8729	1444069297	1.003655	0.003620	0.008862	0.015476

73318 rows × 278 columns

CODE SNIPPET :

```
data_dir = Path('/content/drive/MyDrive/Extrasensory/data') #Providing the path.
os.chdir(data_dir)
extension = 'csv'
data_file = [i for i in glob.glob('*.{}'.format(extension))]
data_file_consideration = random.sample(data_file,10) # Selecting 10 random files for making model.
data = [] #for concatenating the users
for file in data_file_consideration:
  file = '/content/drive/MyDrive/Extrasensory/data/' + file
  df_from_each_valid_file = pd.read_csv(file)
  data.append(df_from_each_valid_file)
train_data = pd.concat(data)
train_data
```

For feature selection, the Accelerometer features had been chosen just as in the starter code. The ‘raw_acc’ and ‘watch_acceleration’ headers were taken to predict the target label which was ‘fix walking’. Both the headers were joined into a single dataframe and can be seen below consisting of 36539 rows and 72 columns.

	raw_acc:magnitude_stats:mean	raw_acc:magnitude_stats:std	raw_acc:magnitude_stats:moment3	raw_acc:magnitude_stats:moment4
2	1.004814	0.002468	0.000855	0.003526
3	1.003934	0.008411	0.011053	0.027717
4	1.010905	0.004780	-0.004857	0.007757
5	1.008356	0.004046	-0.001851	0.007553
6	1.007698	0.002867	-0.001442	0.003738
...	...	...	...	...
7274	1.052018	0.315003	0.327811	0.461477
7275	0.996582	0.136800	0.267213	0.436561
7276	0.984015	0.019549	0.031308	0.060158
7277	1.002833	0.134791	0.186954	0.284937
7278	1.194783	0.429404	0.461017	0.625537

36539 rows × 72 columns

CODE SNIPPET FOR ABOVE :

```
training_data_raw_acc = train_data.filter(regex=("raw_acc:.*"))
training_data_watch_acceleration = train_data.filter(regex=('watch_acceleration:.*'))
traget_data = train_data[['label:FIX_walking']]
training_data = pd.concat([training_data_raw_acc, training_data_watch_acceleration, traget_data], axis=1,
join='inner')
training_data = training_data.dropna()
traget_data = training_data['label:FIX_walking']
training_data = training_data.drop('label:FIX_walking', 1)
training_data.dropna()
```

If we scroll towards the right, we can see that the columns consist of only ‘raw_acc:’ and ‘watch_acceleration’ headers.

The data was then split into an 8:2 ratio with 80% going to the training set and 20% to validation. It is always advantageous to have more number of data in the training set so that the model has ample data to learn from and the weights can be updated properly. More training data leads to calculating loss function in a more optimal manner which ultimately leads to increased performance . Both the training and validation data was then reshaped and standardized so as to have a mean of 0 and a standard deviation of 1. Training was then carried using the Logistic regression model and it gave a validation accuracy of 93.99%. For further assessing the performance of the model and also to get a more precise accuracy, K-fold cross validation was carried out. The difference was minimal and led to an accuracy of 94.05%.

```
[ ]  result = model.score(x_test, y_test)
```

```
[ ]  print("Accuracy: %.2f%%" % (result*100.0))

     Accuracy: 93.99%
```

```
[ ]  print("Accuracy: %.2f%%" % (result_kfold.mean()*100.0))

     Accuracy: 94.05%
```

CODE SNIPPET :

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(training_data,traget_data, test_size=0.2, random_state=100
)
y_train = y_train.values.reshape(-1,1)
y_test = y_test.values.reshape(-1,1)
scaler = StandardScaler()
imputer = SimpleImputer(strategy='mean')
```

```
x_train = scaler.fit_transform(x_train)
y_train = imputer.fit_transform(y_train)
x_test = scaler.fit_transform(x_test)
y_test = imputer.fit_transform(y_test)
# Training Logistic Regression.
model = LogisticRegression(solver='liblinear', max_iter=1000, C=1.0)
model.fit(x_train, y_train)
result = model.score(x_test, y_test)
print("Accuracy: %.2f%%" % (result*100.0))
from sklearn.model_selection import KFold #k-fold cross-validation
kfold = KFold(n_splits=10, shuffle=True)
from sklearn import model_selection
model_logitisc_regression = LogisticRegression(solver='liblinear', max_iter=1000, C=1.0)
result_kfold = model_selection.cross_val_score(model_logitisc_regression, training_data, traget_data, cv=
kfold)
print("Accuracy: %.2f%%" % (result_kfold.mean()*100.0))
```

MODEL SELECTION

The model accuracy was further investigated by experimenting with the its C parameter (I had experimented for C=0.5 and C=1.5), for C=0.5 the accuracy was 94.01% and C=1.5 resulted in 93.98% . Hence, as observed, the accuracy only improved slightly for a decrease in the C value. Hence, testing was done using the unchanged value of C=1.

The Validation Accuracy was further evaluated using the Decision Tree and SVM models.

DECISION TREE:

- First, The Decision Tree classifier was run using its default hyperparameters, this model resulted in a validation accuracy of 92.56%.
- For the second run, criterion was changed to entropy, the accuracy for this was 92.99% .
- For the third tuning, square root was assigned to max features which also gave an accuracy of 92.13%
- finally pruning was assigned a value of 0.5 and 1.0 for the fourth and both gave an accuracy of 92.70 % each.

CODE SNIPPET AND PASTED RESULTS :

CODE SNIPPET :

```
# Decision Tree (Default Hyperparameters.)
from sklearn.tree import DecisionTreeClassifier
decisionTreeModel = DecisionTreeClassifier(random_state=0)
result_kfold_decisionTree = model_selection.cross_val_score(decisionTreeModel, training_data, traget_da
ta, cv=kfold)
print("Accuracy: %.2f%%" % (result_kfold_decisionTree.mean()*100.0))
```

just the second line, that is in calling the DecisionTreeClassifier was changed for the next three experimentation.

for changing entropy criterion -
decisionTreeModel = DecisionTreeClassifier(random_state=0, criterion='entropy')

for changing max_features-
decisionTreeModel = DecisionTreeClassifier(random_state=0, max_features="sqrt")

for changing pruning-
decisionTreeModel = DecisionTreeClassifier(random_state=0, ccp_alpha=1.0)

RESULTS:

```
# Decision Tree (Default Hyperparameters.)
from sklearn.tree import DecisionTreeClassifier
decisionTreeModel = DecisionTreeClassifier(random_state=0)
result_kfold_decisionTree = model_selection.cross_val_score(decisionTreeModel, training_data, traget_data, cv=kfold)
print("Accuracy: %.2f%%" % (result_kfold_decisionTree.mean()*100.0))
```

```
Accuracy: 92.56%
```

```
# Decision Tree (Changing the criterion on Entropy)
from sklearn.tree import DecisionTreeClassifier
decisionTreeModel = DecisionTreeClassifier(random_state=0, criterion='entropy')
result_kfold_decisionTree = model_selection.cross_val_score(decisionTreeModel, training_data, traget_data, cv=kfold)
print("Accuracy: %.2f%%" % (result_kfold_decisionTree.mean()*100.0))
```

```
Accuracy: 92.99%
```

```
# Decision Tree (Changing the max_features to sqrt)
from sklearn.tree import DecisionTreeClassifier
decisionTreeModel = DecisionTreeClassifier(random_state=0, max_features="sqrt")
result_kfold_decisionTree = model_selection.cross_val_score(decisionTreeModel, training_data, traget_data, cv=kfold)
print("Accuracy: %.2f%%" % (result_kfold_decisionTree.mean()*100.0))
```

```
Accuracy: 92.13%
```

```
[ ] # Decision Tree (Changing the cost-Complexity Pruning=1.0)
    from sklearn.tree import DecisionTreeClassifier
    decisionTreeModel = DecisionTreeClassifier(random_state=0, ccp_alpha=1.0)
    result_kfold_decisionTree = model_selection.cross_val_score(decisionTreeModel, training_data, traget_data, cv=kfold)
    print("Accuracy: %.2f%%" % (result_kfold_decisionTree.mean()*100.0))
```

```
    Accuracy: 92.70%
```

SVM :

- Three experiments were carried out for the SVM model.
- The first was with the default model which resulted in 93.45% Accuracy.
- the second was introducing a regularization parameter of C=0.5. This resulted in 93.15% Accuracy.
- the third was changing the same to 1.5 which showed an accuracy of 93.53%

CODE SNIPPETS AND RESULTS :

```
# SVM (Default Hyperparameters.)
from sklearn.svm import SVC
SVMModel = SVC()
result_kfold_SVC = model_selection.cross_val_score(SVMModel, training_data, traget_data, cv=kfold)
print("Accuracy: %.2f%%" % (result_kfold_SVC.mean()*100.0))

# SVM (Changing Regularization parameter = 0.5)
from sklearn.svm import SVC
SVMModel = SVC(C = 0.5)
result_kfold_SVC = model_selection.cross_val_score(SVMModel, training_data, traget_data, cv=kfold)
print("Accuracy: %.2f%%" % (result_kfold_SVC.mean()*100.0))
```

Similar way was done for C=1.5

RESULTS:

```
[ ]  # SVM (Default Hyperparameters.)
     from sklearn.svm import SVC
     SVMModel = SVC()
     result_kfold_SVC = model_selection.cross_val_score(SVMModel, training_data, traget_data, cv=kfold)
     print("Accuracy: %.2f%%" % (result_kfold_SVC.mean()*100.0))

     Accuracy: 93.45%

[ ]  # SVM (Changing Regularization parameter = 0.5)
     from sklearn.svm import SVC
     SVMModel = SVC(C = 0.5)
     result_kfold_SVC = model_selection.cross_val_score(SVMModel, training_data, traget_data, cv=kfold)
     print("Accuracy: %.2f%%" % (result_kfold_SVC.mean()*100.0))

     Accuracy: 93.15%

[ ]  # SVM (Changing Regularization parameter = 1.5)
     from sklearn.svm import SVC
     SVMModel = SVC(C = 1.5)
     result_kfold_SVC = model_selection.cross_val_score(SVMModel, training_data, traget_data, cv=kfold)
     print("Accuracy: %.2f%%" % (result_kfold_SVC.mean()*100.0))

     Accuracy: 93.53%
```

MODEL TESTING

After completion of all the experimented models, it was observed that the Logistic Regression model along with a C parameter of 1, provided the best performance and hence this was chosen for the purpose of testing on 5 other users. These randomly selected users were not present in the 10 previously selected set of users. The 5 users now formed the test set on which accuracy was calculated. The same features were selected with fix walking as the target for testing. The accuracy on the unseen test data was found to be 91.36%.

```
Accuracy on Unseen Data: 91.36%
```

CODE :

```
unSeenDataList =  [i for i in glob.glob('*.{}'.format(extension)) if i not in data_file_consideration] # To av
oid the duplication of data.
test_data_file_consideration = random.sample(unSeenDataList,5) #Selecting 5 random users for test.
test_data = []
for file in test_data_file_consideration:
  file = '/content/drive/MyDrive/Extrasensory/data/' + file
  df_from_each_test_file = pd.read_csv(file)
  test_data.append(df_from_each_test_file)
testing_data = pd.concat(test_data)
testing_data_raw_acc = testing_data.filter(regex=("raw_acc:.*"))
testing_data_watch_acceleration = testing_data.filter(regex=('watch_acceleration:.*'))
traget_test_data = testing_data[['label:FIX_walking']]
testing_data = pd.concat([testing_data_raw_acc, testing_data_watch_acceleration, traget_test_data], axis=1
, join='inner')
testing_data = testing_data.dropna()
traget_test_data = testing_data['label:FIX_walking']
testing_data = testing_data.drop('label:FIX_walking', 1)
testing_data.dropna()

testing_data = scaler.fit_transform(testing_data)
testing_data_target = traget_test_data.values.reshape(-1,1)
testing_data_target = imputer.fit_transform(testing_data_target)
Test_result = model.score(testing_data, testing_data_target)
print("Accuracy on Unseen Data: %.2f%%" % (Test_result*100.0))
```

Other metrics were then calculated and resulted in the following :

1. Balanced Accuracy : 0.661

```
print("Balanced Accuracy Score of the logistic regression: ",balanced_accuracy_score(y_test,y_pred))
```

```
Balanced Accuracy Score of the logistic regression:  0.6611239608474122
```

2. F1 : 0.446

```
F1 Score for the logistic regression:  0.44640605296343006
```

3. Precision : 0.667

```
Precision for the logistic regression:  0.6679245283018868
```

4. Recall : 0.335

```
Recall score for the logistic regression:  0.3352272727272727
```

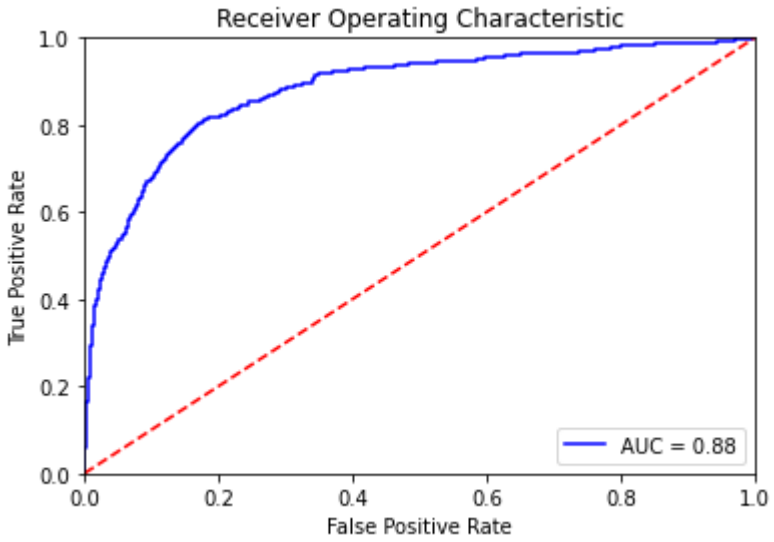
5. ROC-AUC : 0.661

```
ROC AUC score for the logistic regression:  0.6611239608474121
```

The outputs seem to be low because of the dataset being imbalanced with more negative examples than positive ones. The balanced accuracy score seems to have accounted for both the positive and negative outcome for the classes quite well. 66% of precisions indicated that the model is able to predict 66% of the relevant results. The recall measure indicates that 0.335 ratio has been correctly predicted in relation to all other observations.

In relation to real world data, it can be stated that the model would still provide appropriate results because as seen for the testing set it had given an accuracy of close to 92% which was not too bad. It would probably predict well with very few imprecisions if any.

The ROC Curve has been plotted and shown below :



From the ROC curve, it can be seen that the chosen model has not done great but it is pretty acceptable considering the curve tends closely towards 1.It seems to have a high TPR with a low FPR. AUC has given a prediction of 0.88 which means that our model's performance has not been too bad.

CODE SNIPPETS:

```
 # Balanced Accuracy Score
 from sklearn.metrics import balanced_accuracy_score
 y_pred = model.predict(x_test)
   print("Balanced Accuracy Score of the logistic regression: ",balanced_accuracy_score(y_test,y_pred))

# F1 Score.
from sklearn.metrics import f1_score
print('F1 Score for the logistic regression: ', f1_score(y_test,y_pred))
# Precision
```

```
from sklearn.metrics import precision_score
print('Precision for the logistic regression: ', precision_score(y_test,y_pred))
# Recall
from sklearn.metrics import recall_score
print('Recall score for the logistic regression: ', recall_score(y_test,y_pred))
# ROC AUC score.
from sklearn.metrics import roc_auc_score
print('ROC AUC score for the logistic regression: ', roc_auc_score(y_test,y_pred))
```

Plotting ROC Curve-

```
probs = model.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
roc_auc = metrics.auc(fpr, tpr)

plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

PREDICTING OTHER ACTIONS

Apart from walking, prediction was also made for bicycling. The Logistic model was used again . The training data was loaded with ‘Bicycling’ as the target. The same feature headers were re-selected. The data was split again with a 80:20 ratio. After evaluation, the accuracy was found to be 97. 07%.

```
Accuracy: 97.07%
```

The following are the metrics achieved :

1. Balanced Accuracy : 0.76
2. F1 Score : 0.66
3. Precision : 0.89
4. Recall : 0.52
5. ROC-AUC Score : 0.76

```
[ ]  # Balanced Accuracy Score
     from sklearn.metrics import balanced_accuracy_score
     y_pred = model.predict(x_test)
     print("Balanced Accuracy Score of the logistic regression: ",balanced_accuracy_score(y_test,y_pred))
```

```
Balanced Accuracy Score of the logistic regression:  0.7600799443865137
```

```
[ ]  # F1 Score.
     from sklearn.metrics import f1_score
     print('F1 Score for the logistic regression: ', f1_score(y_test,y_pred))
```

```
F1 Score for the logistic regression:  0.6599999999999999
```

```
[ ]  # Precision
     from sklearn.metrics import precision_score
     print('Precision for the logistic regression: ', precision_score(y_test,y_pred))
```

```
Precision for the logistic regression:  0.8918918918918919
```

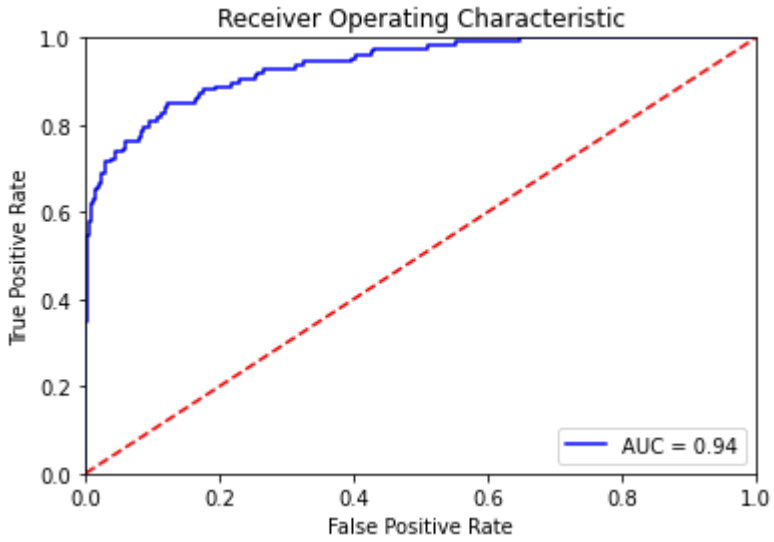
```
# Recall
from sklearn.metrics import recall_score
print('Recall score for the logistic regression: ', recall_score(y_test,y_pred))
```

```
Recall score for the logistic regression:  0.5238095238095238
```

```
# ROC AUC score.
from sklearn.metrics import roc_auc_score
print('ROC AUC score for the logistic regression: ', roc_auc_score(y_test,y_pred))
```

```
ROC AUC score for the logistic regression:  0.7600799443865137
```

The ROC curve has been plotted below :



We can see that the model performed better and gave a more precise result for Bicycling target compared to fix walking. The ROC curve is clearly improved than the one for fix walking. The AUC came out to be 0.94 which tells us that the model very clearly was able to distinguish between

different classes. Similar code as the previous one for fix walking metrics was used for achieving these results.

Conclusion : The various models were thus examined and testing was made on the Logistic Regression Model. All the metrics had been calculated and resulting ROC curves were also plotted.

Part 2

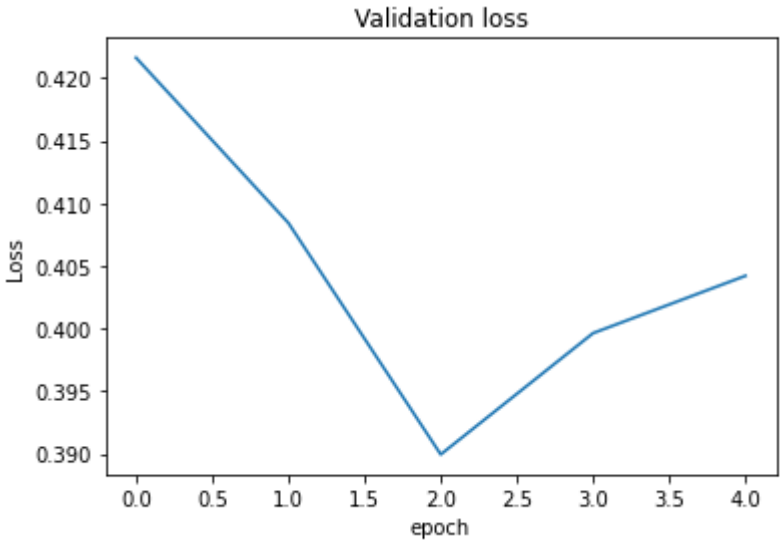
Aim : To experiment with Deep Convolutional Neural Networks to identify the presence of features on the surface of Mars from imagery.

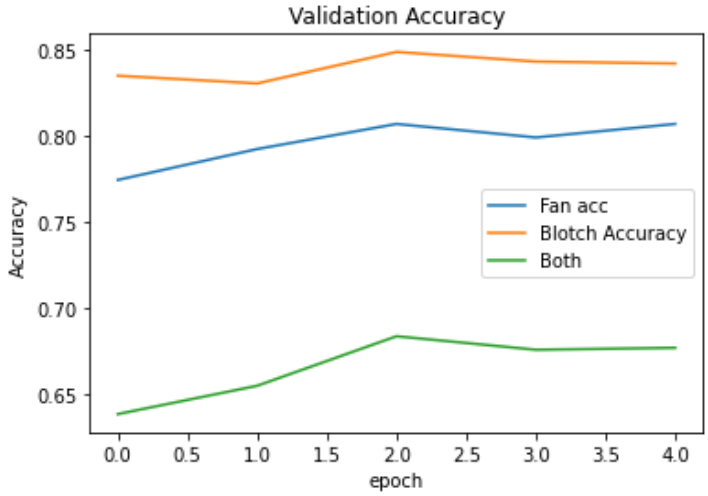
TRAIN A MODEL

The starter Code notebook was uploaded to Google Colab and the drive consisting of the dataset was mounted. The input images were rescaled for ease of computational purposes and shuffled so that the model gets the variation of data for learning. The pre-trained ResNet50 model was first trained for 5 epochs by simply running the given starter code and saving the model to a checkpoint. After plotting the learning curves, it was seen that with each epoch, the validation loss kept decreasing and then increased again. the validation accuracy on the other hand kept improving. This shows that the model was very inaccurate and not trained for enough epochs to tell how well it had learned. The reported validation loss and accuracy were as follows:

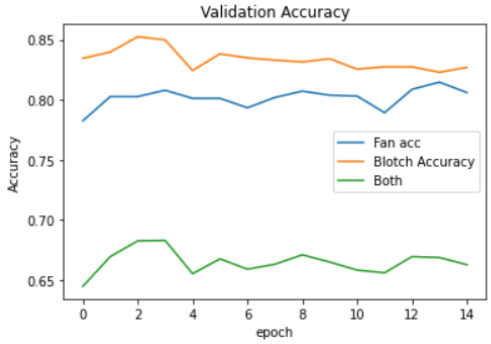
```
100%|██████████| 376/376 [1:26:16<00:00, 13.77s/it]
[01] train loss: 0.4791  valid loss: 0.4216   fan acc: 0.7746   blotch acc: 0.8353   both acc: 0.6383
100%|██████████| 376/376 [02:52<00:00,  2.19it/s]
[02] train loss: 0.3959  valid loss: 0.4084   fan acc: 0.7926   blotch acc: 0.8308   both acc: 0.6548
100%|██████████| 376/376 [02:51<00:00,  2.20it/s]
[03] train loss: 0.3544  valid loss: 0.3899   fan acc: 0.8072   blotch acc: 0.8491   both acc: 0.6836
100%|██████████| 376/376 [02:51<00:00,  2.19it/s]
[04] train loss: 0.3215  valid loss: 0.3996   fan acc: 0.7993   blotch acc: 0.8435   both acc: 0.6758
100%|██████████| 376/376 [02:51<00:00,  2.20it/s]
[05] train loss: 0.2880  valid loss: 0.4042   fan acc: 0.8072   blotch acc: 0.8424   both acc: 0.6769
```

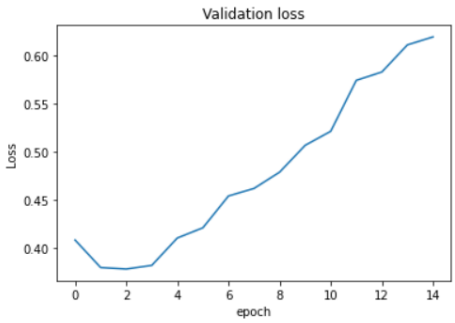
The graphs were plotted and the model shows to have given accuracies of close to 85% for blotch detection, 80s for fan detection and around 68% for both.The graphs have been pasted below :





This next part has been done in a separate colab file named part2_cont . The model was trained further for 15 epochs with SGD as the optimizer along with momentum of 0.9 and weight decay of 0.0001. The model was compiled with SGD as the optimizer because it fitted well for this dataset. It is also known to be a low-cost optimizer. The low learning rate of 0.001 was chosen because this made the model learn slowly which would enable it to give a more accurate prediction. Binary cross entropy loss was used for updating the weights. L2 regularization or 'weight decay' was also used so as to avoid overfitting and exploding gradients. It was observed that the model was overfitting regardless,





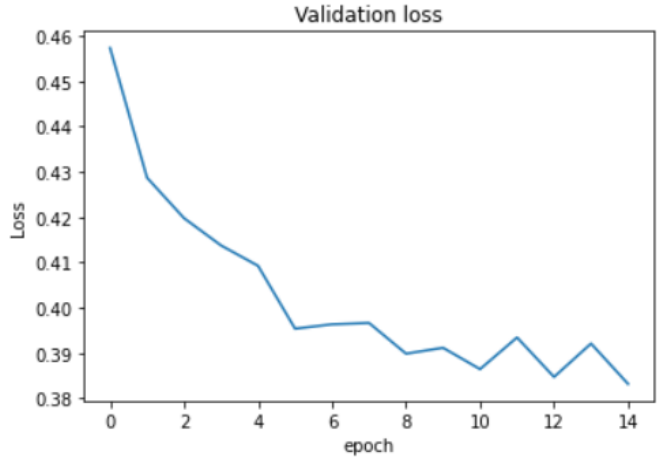
For solving this issue, augmentation had been implemented before training all the other models. This was done so as to increase the network's complexity and also to provide variation and extra data for it to learn better and thus to provide a more accurate result. The Different Augmentation techniques applied were : **Flipping**- for flipping the image horizontally with a momentum of p = 0.9, **rotation**- to rotate the image by an angle of 180 degrees, a **conversion transform** so as to read data in a tensor format, **cropping** the tensor image at the center and **normalizing** the images to have a normalized mean and standard deviation. The model was then trained and saved to '/content/drive/myDrive/Models/res_sgd.pkl'. The following is a snippet from the used code :

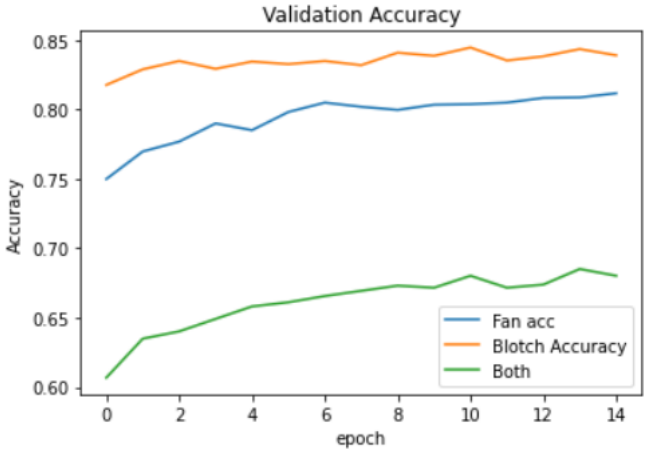
```
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.9),
    transforms.RandomRotation(degrees=180),
    transforms.CenterCrop((200,100)),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

valid_transform = transforms.Compose([
```

```
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])
```

On plotting the Validation Loss and Accuracy graphs, it can be seen that the model performed a lot better with augmentation. A checkpoint was saved for each time the model was done training. The resulting graphs for 15 epochs are as follows:





EXPERIMENTS

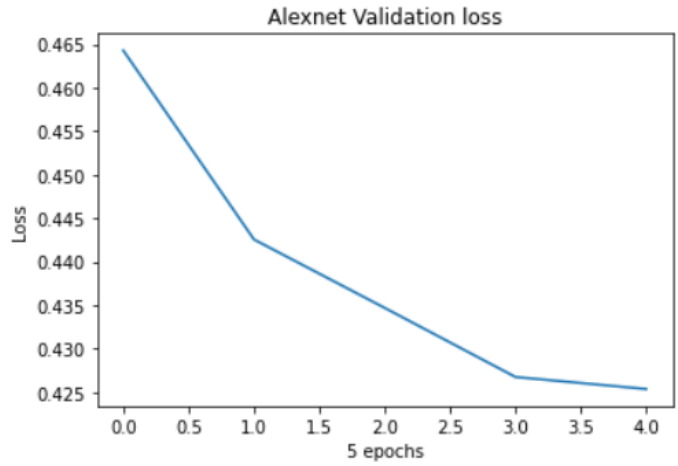
Experiments had been done on the following models(Each model had been trained for 5 epochs only):

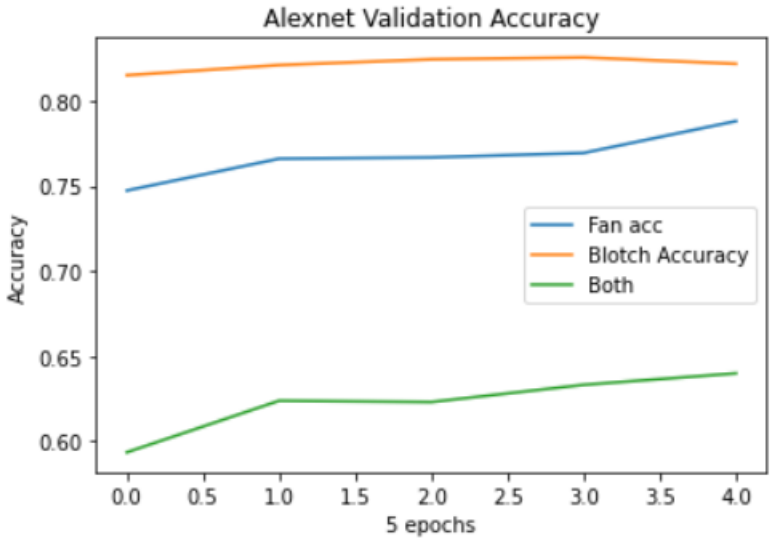
1. Alexnet Network

The first model to be experimented on was the AlexNet network architecture. The following code snippet was used :

```
model = models.alexnet(pretrained=True)
model.classifier[6] = nn.Linear(4096,2)
model.to(device);
```

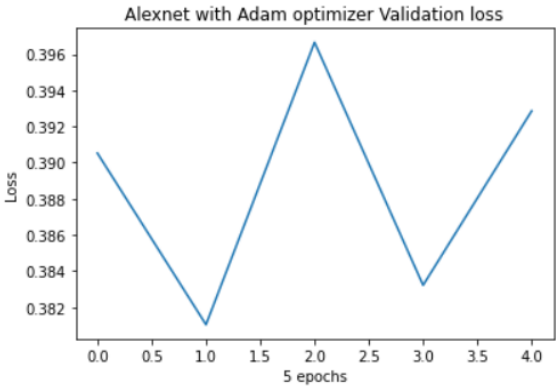
A pretrained AlexNet model was loaded. The final layer was tuned to our network requirement so as to derive our predictions. SGD was used here as the optimizer with lr= 0.001 and momentum of 0.9. The model had given a validation accuracy in 80s for blotch detection, high 70s for fan and around 65 for both. Final loss of 0.425 was observed .The graphs of the same have been shown below :

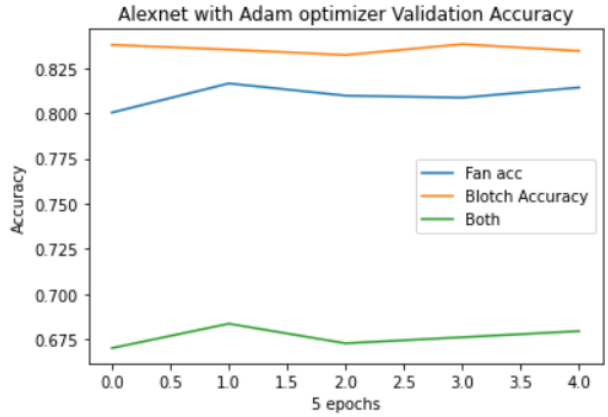




2. Alexnet with Adam as optimizer

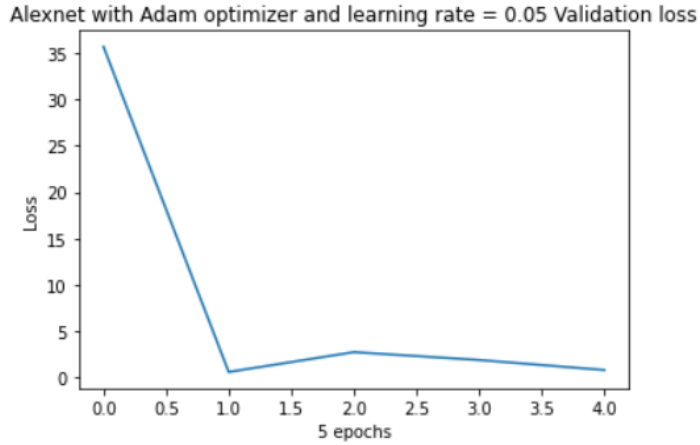
   The same AlexNet network was again loaded. After changing the optimizer to ‘Adam’, the network performance was satisfactory and can be seen in the following graphs :

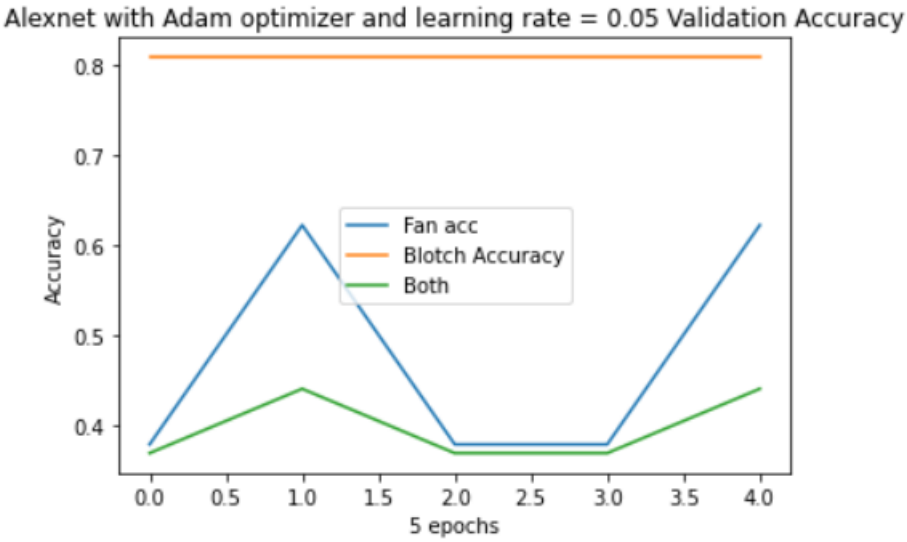




3. <u>Alexnet with Adam optimizer and learning rate= 0.05</u>

   Further experimentation was done by changing the Learning rate of the same model to 0.05 and this too did not seem to improve the accuracy. It also did not seem to be doing any learning.



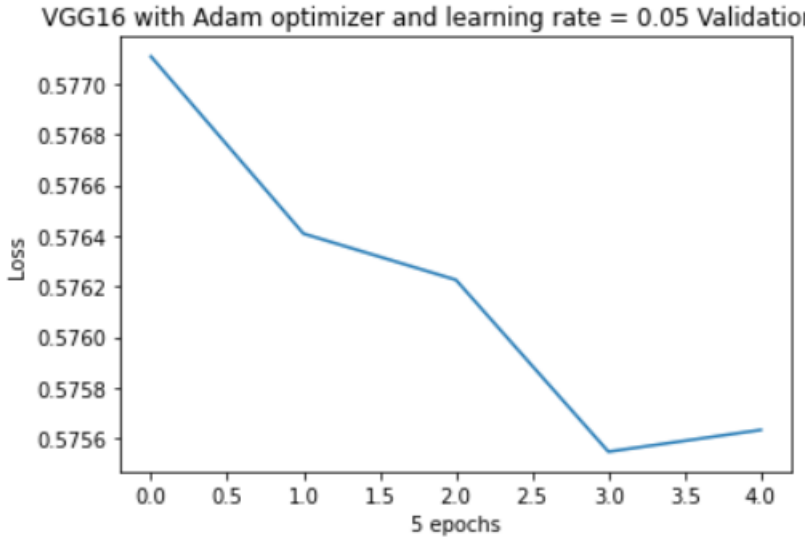


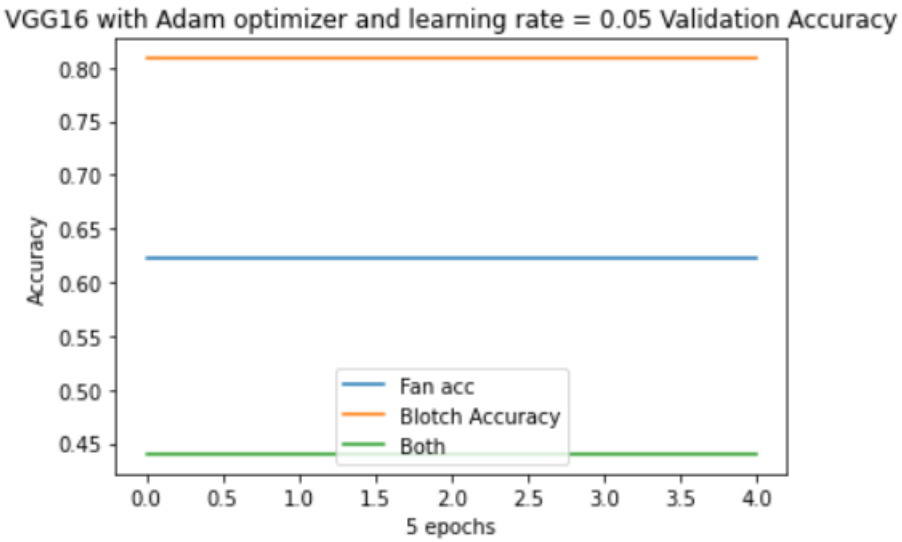
4. VGG Network with Adam optimizer and lr=0.05

Experimentation was then changed to the VGG Network with continuation of lr=0.05. Here too the pre-trained vgg16 Network was loaded and final layer modified according to our requirement. The code snippet used is as follows:

```
model = models.vgg16(pretrained=True)
model.classifier[6] = nn.Linear(4096,2)
model.to(device);
```

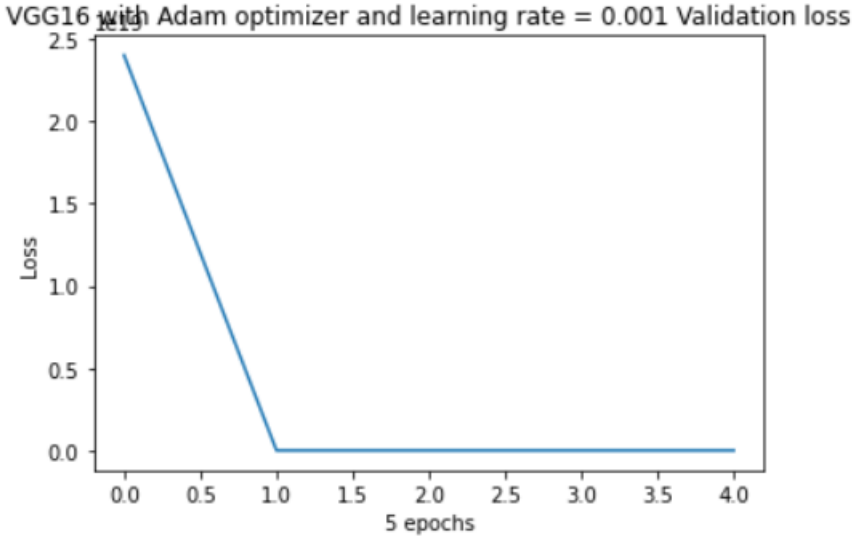
The model was trained for 5 epochs and gave the following plotted predictions :

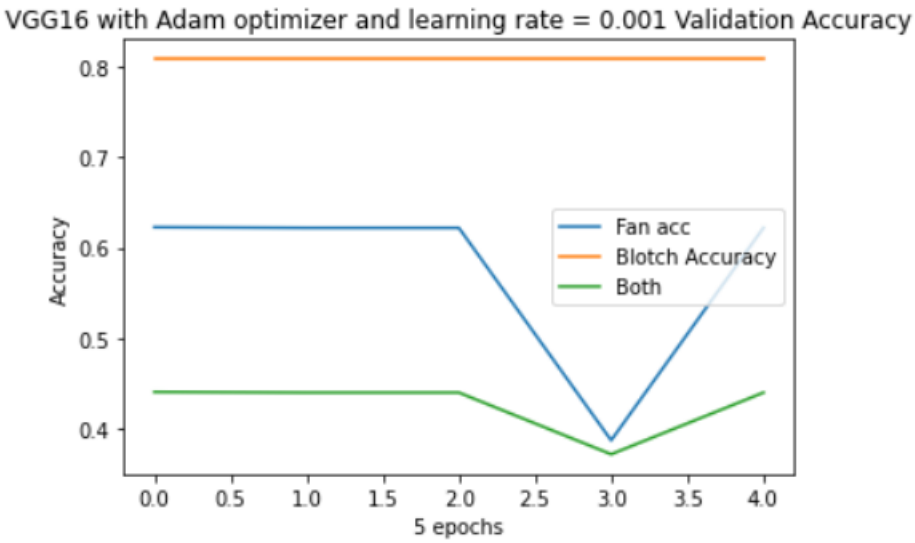




5. VGG with Adam optimizer and lr = 0.001

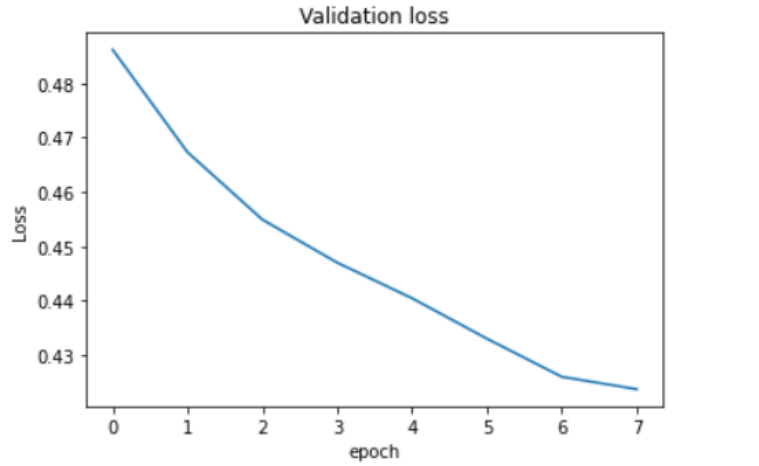
The previous VGG network was then changed by training with a learning rate of 0.001. As the previous model did not do any learning, the learning rate was reduced so that the model would train slower and not overshoot the minima. However, the model still did not prove to be any more efficient than any of the others. The validation loss and accuracy here was as follows :

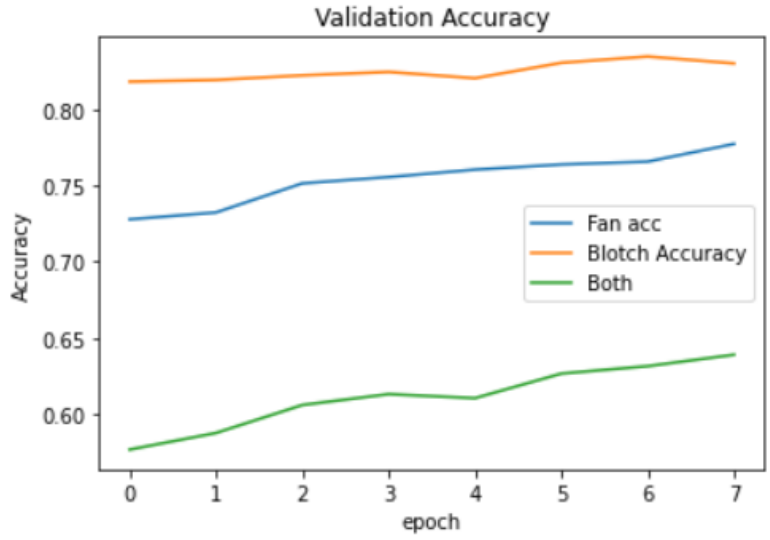




6. VGG16 with SGD optimizer lr=0.001

Finally, the VGG16 model was trained with SGD optimizer. This was done on the fifth file named as 'vgg_sgd'. Sticking to the learning rate of 0.001, the model performed far better as compared to all the other models, except ofcourse the resnet50 with the same SGD optimizer. The plotted graph outputs are as shown below :





EVALUATION

After Assessing all the models, it can finally be concluded that although both the resnet50 and VGG16 network when used along with SGD optimizer provided very good accuracies which were almost similar, the resnet50 model should be chosen to evaluate the final test set as it was learning better.