**IMDb Reviews Sentiment Analysis**- Project Report

Problem Statement

In this project I am going analyse the sentiment in the IMDb movie reviews, the task is to create a model which will take in the movie reviews and predict if the movie review is a +ve review or a -ve review , because of which the users will not have to go through all the long movie reviews to figure out if he/she should watch the movie or not , he/she will know that these many people liked the movie and these many people hated the movie.

This is a binary classification problem.

About the data set

I picked up a publicly available data set which was having 50K+ samples and two columns **review** and **sentiment**.

The data set is well balanced i.e **about 25K positive and about 25k negative reviews.**

*"One of the other reviewers has mentioned that after watching just 1 Oz episode you'll be hooked. They are right, as this is exactly what happened with me.<br /><br />The first thing that struck me about Oz was its brutality and unflinching scenes of violence, which set in right from the word GO. Trust me, this is not a show for the faint hearted or timid. This show pulls no punches with regards to drugs, sex or violence. Its is hardcore, in the classic use of the word.<br /><br />It is called OZ as that is the nickname given to the Oswald Maximum Security State Penitentary. It focuses mainly on Emerald City, an experimental section of the prison where all the cells have glass fronts and face inwards, so privacy is not high on the agenda. Em City is home to many..Aryans, Muslims, gangstas, Latinos, Christians, Italians, Irish and more....so scuffles, death stares, dodgy dealings and shady agreements are never far away.<br /><br />I would say the main appeal of the show is due to the fact that it goes where other shows wouldn't dare. Forget pretty pictures painted for mainstream audiences, forget charm, forget romance...OZ doesn't mess around. The first episode I ever saw struck me as so nasty it was surreal, I couldn't say I was ready for it, but as I watched more, I developed a taste for Oz, and got accustomed to the high levels of graphic violence. Not just violence, but injustice (crooked guards who'll be sold out for a nickel, inmates who'll kill on order and get away with it, well mannered, middle class inmates being turned into prison bitches due to their lack of street skills or prison experience) Watching Oz, you may become comfortable with what is uncomfortable viewing....thats if you can get in touch with your darker side."*

This is a sample review, so we can see that the person who wrote this review is talking about some movie or series he watched, the review is positive in nature an the person is was very happy after watching the movie.

The **sentiment** is a binary categorical feature which has ‘positive’ and ‘negative’ string entries.

The above mentioned movie review is a ‘positive’ review.

**DATA PREPROCESSING**

On observing the data set I noticed that there were a lot of **html tags**, **unnecessary punctuations**, also noticed that there were almost 0 spelling mistakes , no emojis , also noticed the fact that there are words which are represented using different cases.

**One significant thing to notice here is that the reviews are very lengthy, the above mentioned review has 307 words.** From this one can rule out a few possible DL architectures which could have been used to solve this problem(more on this later)

Didn’t notice many numbers as well.

**Basic Preprocessing –**

1. Convert the review in lower case so that the same word doesn’t tokenise multiple times and we don’t get an inflated vocabulary
2. Remove html tags
3. Remove any special symbol i.e anything other than alphabets(lower upper case) and numbers , this will remove all the punctuation marks
4. Remove **stop words** , these are the words which are used in sentence formation but they don’t realty convey any meaning (replace the removed words using a single space)
5. Post doing all the above steps , remove the extra blank spaces using a single space , to do this we will use a regular expression

**This will finish our basic preprocessing part.**

*def preprocessing(sentence):*

*sentence=sentence.lower()*

*sentence=re.sub('<.\*?>',' ',sentence)*

*sentence=re.sub(r'[^a-zA-Z0-9\s]',' ',sentence)*

*stop\_words=stopwords.words('english')*

*sentence=word\_tokenize(sentence)*

*l=[]*

*for i in sentence:*

*if i in stop\_words:*

*l.append('')*

*else:*

*l.append(i)*

*cleaned\_sentence=' '.join(l)*

*cleaned\_sentence=re.sub(r'\s+',' ',cleaned\_sentence)*

*return cleaned\_sentence*

This is the function called **preprocessing** it does the job

1. Lowered the text using the string method .lower()
2. Used a regular expression to remove html tags and things inside it ‘<.\*?>’, to use these you will have to import **re** module i.e regular expression module
3. Removed special characters , removed anything other than a-z,A-Z and 0-9 , so this is how I removed all the punctuation marks(there was no need to remove A-Z, all my sentences were lowered) using ‘[^a-zA-Z0-9]’ regex
4. From nltk.corpus imported stopwords class , it is a list of stopwords of different languages , removed stopwords from my reviews using it
5. Then in the end I handles extra spaces using the following regular expression ‘\s+’

**Applied all these changes to all the reviews and post applying these changes saved them in a list**

Also I label encoded my dependent column , **‘positive’-1 and ‘negative’-0**

**TRAIN TEST SPLIT**

Did a train test split and stored 40k reviews in X\_train, Y\_train and saved 10k reviews in X\_test, Y\_test

**This is how a movie review looks like post basic preprocessing!!**

*' watershed event movie watching life went see theater came completely amazed bad movies like make wonder put money owed favor large favor special effects absolutely first grade level first grader could done toy rubber bats strings attempt hide strings arrows appear drawn film look shape arrow find street sign laughable story line ed wood made masterpieces compared conquest every film student see thing know definition bad movie'*

**ADVANCED TEXT PREPROCESSING**

**This includes stemming/lemmatisation and tokenisation of the reviews**

We have reviews in the above pre-processed format , we will convert a review into word tokens and then we will pass it to the lemmatiser word by word and then post lemmatisation we will get lemmatised word tokenised reviews.

We will do this with the help of a function

***def tokenize\_and\_lemmatize(text):***

***tokens = word\_tokenize(text)***

***lemmatizer = WordNetLemmatizer()***

***lemmatized\_tokens = [lemmatizer.lemmatize(token,pos='n') for token in tokens] Noun lemmatization***

***lemmatized\_tokens = [lemmatizer.lemmatize(token,pos='v') for token in lemmatized\_tokens] Verb Lemmatisation***

***return lemmatized\_tokens***

Now here we have imported word\_tokenize from  **nltk.tokenize**  and **nltk.stem import WordNetLemmatizer**

**Input- ‘sahil good boy icecream boys played plays play’**

**Output -** tokens pre lemmatisation: ['sahil', 'good', 'boy', 'icecream', 'boys', ‘played’, ‘plays’, ‘play’]

tokens post lemmatisation: ['sahil', 'good', 'boy', 'icecream', 'boy', ’play’, ‘play’, ‘play’]

post joining the words = “sahil good boy icecream boy play play play”

# Tokenize and lemmatize the text data

X\_train = [' '.join(tokenize\_and\_lemmatize(text)) for text in X\_train]

X\_test = [' '.join(tokenize\_and\_lemmatize(text)) for text in X\_test]

Here we have applied this tokenisation thing to X\_train and X\_test , also we have joined the words and saved it in X\_train and X\_test which are lists consisting of 40k and 10k reviews.

The following is the same review which I have mentioned above as well (post basic pre processing), this is post lemmatisation

This is the output of X\_train[10] post noun+verb lemmatisation

**‘watershed event movie watch life go see theater come completely amaze bad movie like make wonder put money owe favor large favor special effect absolutely first grade level first grader could do toy rubber bat string attempt hide string arrow appear draw film look shape arrow find street sign laughable story line ed wood make masterpiece compare conquest every film student see thing know definition bad movie’**

**NOUNS AND VERBS ARE NOW IN THEIR BASE FORMS**

**Why I chose lemmatization over stemming?**

Following are the reasons of choosing lemmatisation over stemming-

1. Lemmatization is more effective in reducing the vocabulary size as compared to stemming , example- say in a document we have three words , run , ran , running , now these words mean the same specially for tasks like sentiment analysis they are just the same , so they should be tokenized as one , but if you apply stemming to these words you will get , run , ran and runn. So we have 3 different words which mean the same post stemming , infact one of them is not even a word in English vocabulary and these will be counted as 3 different words in the vocabulary of the corpus .

No if you apply **lemmatisation** you will get run,run,run (verb lemmatisation) wich is very accurate and these will be tokenised as a single word also they will not be identified separately in the vocabulary of the corpus effectively reducing the dimensions of the vectors.

1. The only advantage which stemming has over lemmatisation in context of my problem is the fact that stemming is very quick to apply because it uses a set of algorithms developed by linguistic experts where as lemmatisation uses WordNet which is a lexical dictionary which has relations between words.

**Why I applied noun and verb lemmatisation?**

The reason is pretty simple, I observed nouns which were not in their base form example “boys”, similarly verbs in their continuous and past forms example ‘biting’ etc , also I noticed adverbs like ‘completely’, so as to get transform these words into their base form for effective tokenisation I decided to apply noun, verb, adverb lemmatisation

**TOKENIZER**

**This gives a unique number to unique words of the corpus**

*from tensorflow.keras.preprocessing.text import Tokenizer*

*tokenizer = Tokenizer(num\_words=30000, oov\_token='<OOV>')*

*tokenizer.fit\_on\_texts(["sahil good boy play play icecream",'sahil like mango icecream tasy good sweet'])*

*sequences= tokenizer.texts\_to\_sequences(["sahil good boy play play icecream",'sahil like mango icecream tasy good sweet'])*

*sequences*

Here I have tokenizer class has been imported and I have provided it some parameters

*num\_words=30000* – it means that the upper limit to the vocab -30000 , it will consider most frequent

30k words, when this parameter is set the tokenizer will tokenise on the basis of

word frequency.

*oov\_token=’<OOV>’* – the words which aren’t in the vocab will be represented using this (oov-out of vocb)

tokenizer.fit\_on\_texts expects a list of documents , it counts the unique words in the document and then gives them a numerical indexing

**[[2, 3, 6, 4, 4, 5], [2, 7, 8, 5, 9, 3, 10]] – This is the output of the above code –** The word sahil has been tokened using the number 2 and play has been tokened with the number 4 and we have a total of 10 unique words in the vocabulary thus the token number varies between 1 to 10 , index 1 is reserved of the oov token

APPLYING TOKENIZATION ON THE WHOLE TRANING AND TESTING REVIEWS

*from tensorflow.keras.preprocessing.text import Tokenizer*

*# Initialize the Tokenizer*

*tokenizer = Tokenizer(num\_words=50000, oov\_token='<OOV>')*

*tokenizer.fit\_on\_texts(X\_train)*

*# Convert the text to sequences*

*X\_train\_sequences = tokenizer.texts\_to\_sequences(X\_train)*

*X\_test\_sequences = tokenizer.texts\_to\_sequences(X\_test)*

The X\_train[10] have been tokenized as following : unable to paste it here but it is a list consisting of 67 entries

**PADDING**

No the reviews have been tokenized , and each and every review has different number of words, some reviews have 300 words some has 50-60 , thus we need to make them uniform , so we will perform padding and set a max length

How to decide max length :  
**I used X\_train\_sequences which is a list of lists and found out the average length of lists in this lists i.e the average number of words in the processed reviews . It came out to be 119.6 , so I set max length to 120**

*X\_train\_pad=pad\_sequences(X\_train\_sequences,padding='post',maxlen=maxlen)*

We have done post padding !!

X\_train\_pad has got 40k reviews where each review is being represented using 120 numbers , where each number is the tokened form of a word !!

**Truncating:** If maxlen is set to 120, any review longer than 120 words will be truncated. This could lead to loss of important information if many reviews exceed this length.

**Padding:** Reviews shorter than 120 words will be padded, which could introduce a lot of padding tokens if maxlen is much higher than necessary

**Thus this parameter has to be tuned depending on the models performance**

Now we will use this to vectorise the words using GloVe Embedding technique and then we will feed those vectors to our LSTM architecture!!

**GloVe EMBEDDING (Global Vector)**

This is an excellent technique to perform word level vectorisation. I chose this technique because it is capable of extracting the semantic relationship between words, also we have pretrained word embeddings , which have been trained on large data sets consisting of billions of unique words. I preferred it over the Word2Vec technique because it vectorises on the basis of whole corpus using the co-occurrence matrix whereas Word2Vec only considers local context while vectorising, thus it captures the semantic relationship better.

The reason for using pretrained embeddings!

1. Obviously it is going save time and computational power
2. When you are not working on a very unique vocab or your vocab consists of very common words then you can use these pretrained embeddings which have been trained on a very large corpus sometimes crossing billions of tokens(words)
3. Pre-trained embeddings are trained on vast amounts of data, capturing a broad range of semantic and syntactic relationships between words. This helps in improving the performance of your models on tasks like sentiment analysis, text classification, and named entity recognition.
4. Models using pre-trained embeddings often generalize better to unseen data, especially if the pre-trained embeddings are from a corpus similar to the target task.
5. Pre-trained embeddings often handle rare words better by leveraging the co-occurrence information from the large training corpus.

**Advantages of self trained embeddings**

1. **Training Time:** Training word embeddings from scratch can be computationally expensive and time-consuming, especially with large datasets and complex models.
2. **Resource Usage:** It requires significant computational resources, including memory and processing power, which might not be available in all environments.
3. **Overfitting Risks:** With limited data, there is a risk of overfitting the embeddings to the specifics of your dataset, leading to poor generalization to unseen data or tasks.
4. **Data Dependence:** Training embeddings from scratch typically requires a substantial amount of data to capture meaningful semantic relationships. In cases where your dataset is small, the embeddings might not perform as well as pre-trained embeddings, which are trained on vast and diverse corpora

**Popular Pre-trained GloVe Embeddings:**

1. **GloVe Common Crawl:**
   * **Corpus:** 42 billion tokens
   * **Dimensionality:** 50, 100, 200, 300 dimensions
   * **URL:** [Common Crawl GloVe](https://nlp.stanford.edu/projects/glove/)
2. **GloVe Wikipedia 2014 + Gigaword 5:**
   * **Corpus:** 6 billion tokens
   * **Dimensionality:** 50, 100, 200, 300 dimensions
   * **URL:** [Wikipedia + Gigaword GloVe](https://nlp.stanford.edu/projects/glove/)
3. **GloVe Stanford News:**
   * **Corpus:** 1.6 billion tokens
   * **Dimensionality:** 50, 100, 200, 300 dimensions
   * **URL:** [Stanford News GloVe](https://nlp.stanford.edu/projects/glove/)

WE USED THE STANFORD ONE WHICH HAS 1.6 BILLION TOKENS i.e 1.6 Billion WORDS, and each word was represented using a 100 dimensional vector

This is the code !!

**GloVe File**

*embeddings\_dictionary = dict()*

*# Open the GloVe file (a2\_glove.6B.100d.txt) with utf-8 encoding*

*glove\_file = open("C:\IIT B\IMDB Reviews Sentiment Analysis\glove\_embeddings.txt", encoding="utf8")*

*for line in glove\_file:*

*records = line.split()*

*word = records[0]*

*vector\_dimensions = np.asarray(records[1:], dtype='float32')*

*embeddings\_dictionary [word] = vector\_dimensions*

*glove\_file.close()*

EMBEDDING MATRIX

*vocab\_size = min(len(tokenizer.word\_index) + 1, tokenizer.num\_words)*

*top\_words = {word: index for word, index in tokenizer.word\_index.items() if index < vocab\_size}*

*embedding\_matrix = np.zeros((vocab\_size, 100))*

*for word, index in top\_words.items():*

*embedding\_vector = embeddings\_dictionary.get(word)*

*if embedding\_vector is not None:*

*embedding\_matrix[index] = embedding\_vector*

Post this you get an embedding matrix of 50000,100 shape , i.e in this matrix we have 100 dimensional vectors representing 50k words

Now to vectorise a sentence/document we will place the word embeddings one after the other to get the vector representation of the document and we will feed it to the LSTM architecture timestamp by timestamp.

**THE ARCHITECTURE**

For this task we are going to use an LSTM based architecture , we will not use SimpleRNN because of the obvious disadvantages it carries including bad long term memory, and problems related to vanishing gradient and exploding gradient , also we have long reviews , i.e as we saw earlier a review has got 120 words at an average so we need an architecture which has really good long term memory , so we will be using the obvious LSTMs , we are not going to use any further advanced architecture because it is a guided project and given the fact that there are more people who need to cover RNNs and LSTMs the project guide decided to go with LSTMs , further LSTM works just fine !

**Vanishing/Exploding Gradient Problem**:

* **SimpleRNNs**: These suffer from the vanishing/exploding gradient problem, especially when dealing with long sequences. This makes it difficult for them to learn and retain information from earlier in the sequence.
* **LSTMs**: Designed with special gating mechanisms (forget gate, input gate, and output gate) that help regulate the flow of information, allowing the network to retain or forget information as needed. This helps in mitigating the vanishing gradient problem.

**Long-Term Dependencies**:

* **SimpleRNNs**: Struggle to capture long-term dependencies, which are crucial for understanding the context in tasks like sentiment analysis where the sentiment of a sentence can depend on words appearing far apart.
* **LSTMs**: Are explicitly designed to capture long-term dependencies by maintaining a cell state that carries information across time steps.

**Context Retention**:

* **SimpleRNNs**: Might lose important contextual information over longer sequences.
* **LSTMs**: By controlling what information to keep or discard, LSTMs are better at retaining important context over longer sequences.

**Performance**:

* **LSTMs**: Tend to perform better than SimpleRNNs in many sequential tasks, including sentiment analysis, because of their ability to capture complex patterns and long-term dependencies.

**Embedding layer in the LSTM architecture**

* One point to note down , we have the embedding matrix of the size (50000,100) , which has the vector representation of 50k words , each word in 100 dimensions, now we will be feeding the sentences X\_train\_pad (i.e padded documents of length 120 remember we set maxlen=120 by figuring out the average number of words in a document)
* The sentences will have to be fed to the LSTM word by word , so we will have to vectorise the sentences first using word embedding
* For this we will have to have an embedding layer in our architecture, we will be using pretrained embeddings so the weights are going to be non trainable!

Here’s how it works:

1. **Tokenized and Padded Input**:
   * Your input documents are tokenized (converted into sequences of integers where each integer represents a word) and padded (to ensure that all sequences have the same length).
2. **Embedding Layer**:
   * The embedding layer takes these tokenized sequences and converts each integer (word index) into a dense vector of fixed size, using the pretrained embeddings you've provided (in your case, a matrix of size 50,000 x 100).
   * Essentially, each word in your document is replaced by its corresponding vector from the embedding matrix. For example, if a word's tokenized index is 42, the embedding layer retrieves the 42nd vector from your embedding matrix and uses that vector as the representation of the word.
3. **Output of Embedding Layer**:
   * The output of the embedding layer is a 2D or 3D tensor (depending on whether you process single or batch inputs). This tensor has the shape (batch\_size, sequence\_length, embedding\_dim), where:
     + batch\_size is the number of documents in a batch.
     + sequence\_length is the length of each padded document (number of words).
     + embedding\_dim is the size of each word vector (100 in your case).
4. **Feeding to LSTM Layer**:
   * The LSTM layer takes this tensor as input and processes each word vector sequentially, considering the context provided by previous words. This helps the model capture the relationships and dependencies between words in the sequence.

*model=Sequential()*

*model.add(Embedding(input\_dim=vocab\_size, # vocab size*

*output\_dim=embedding\_dim, # dim of word vectors post embedding*

*weights=[embedding\_matrix], # pre tranied embedding matrix*

*input\_length=maxlen, # this is the length of the words in a given document , we set this before padding, currently its 120*

*trainable=False))*

*model.add(LSTM(128,return\_sequences=True,input\_shape=(maxlen,embedding\_dim)))*

*model.add(LSTM(64))*

*model.add(Dense(64,activation='relu'))*

*model.add(Dropout(0.2))*

*model.add(Dense(1, activation='sigmoid'))*

**Note the fact that the default activation of the LSTM layer is tanh , which is compressive in nature, we didn’t change it to tanh because we observed that we were facing exploding gradient problem (sudden increase in the training loss during training)**

The loss function used is ‘binary\_crossentropy’

**Total params:** 5,170,881 (19.73 MB)

**Trainable params:** 170,881 (667.50 KB)

**Non-trainable params:** 5,000,000 (19.07 MB)

A total of 171k trainable parameters !!

**EARLY STOPPING**

**from tensorflow.keras.callbacks import EarlyStopping**

**es=EarlyStopping(monitor='val\_accuracy',**

**min\_delta=0.001,**

**restore\_best\_weights=True,**

**patience=20,**

**verbose=1)**

* + **DO NOTICE THE DROPOUT LAYERS IN THE DENSE LAYER OF MY ARCHITECTURE, PLACED THERE TO MITIGATE OVERFITTING!!**

**MODEL EVALUATION**

1. Plotted curves like training loss vs validation loss
2. Also plotted training accuracy vs validation accuracy curve !!

Since it is a binary classification problem the output layer has a softmax activation function , so the output will be probabilistic i.e if for a sample the output is 0.9 it is likely to belong to category 1.

T**he important part to note here is that it isn’t necessary that the threshold is of 0.5 i.e if probability of a query point is lesser than 0.5 it belongs to category 0 and if it is more than 0.5 it belongs to category 1 .**

**We need to figure out the optimal threshold so as maximise the correct classification of the model and ultimately to maximise the accuracy score and f1 score as well .**

**WE WILL CALCULATE THIS OPTIMAL THRESHOLD USING THE ROC AUC CURVE :**

*Y\_scores=Y\_scores.reshape(10000,)*

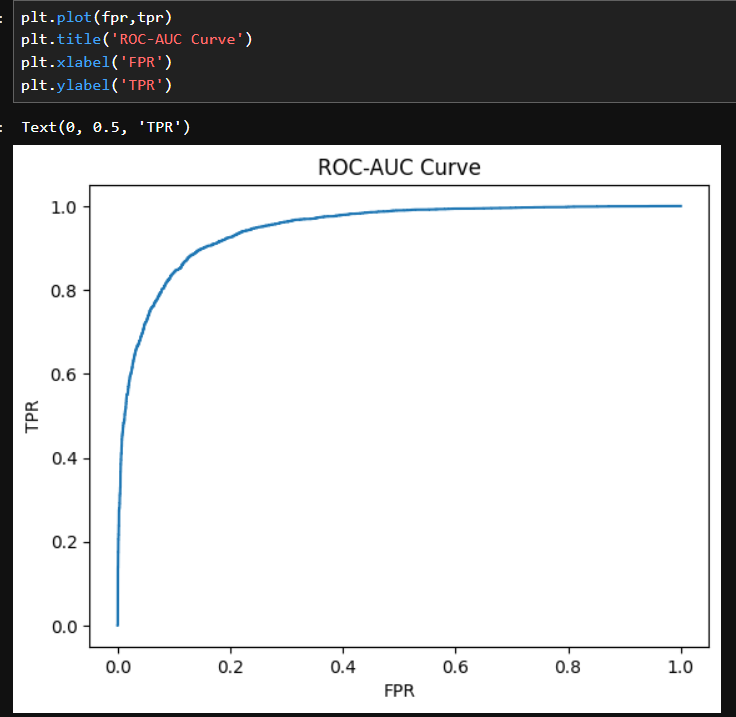
*Y\_scores*

Y\_scores has probabilistic output for these 10000 testing samples

This is how we have our fpr, tpr, thresholds

*from sklearn.metrics import fpr , tpr , thresholds,, accuracy\_score, f1\_score*

*fpr, tpr, thresholds = roc\_curve(Y\_test, Y\_scores)*

**

This is the ROC-AUC curve

This will give us the optimal threshold

*optimal\_idx = np.argmax(tpr - fpr) # THIS IS THE*

*optimal\_threshold = thresholds[optimal\_idx]*

*print("Optimal threshold is:", optimal\_threshold)* which is 0.46972537 in my case !!

Now the fpr, tpr and thresholds are arrays , and we have values of fpr and tpr for different thresholds

(Tpr-fpr) is going to be the matrix of the same shape as that of tpr, fpr and thresholds, argmax will extract the index number for which (tpr-fpr) is max and will extract the threshold from the corresponding index

In my case the size of tpr, fpr, thresholds is (1872,) I,e 1 D array with 1872 entries.

The optimal threshold has been mentioned above !!

**ROC-AUC SCORE**

**This one uses the trapezoidal rule to calculate the area under the ROC curve !**

*from sklearn.metrics import roc\_auc\_score*

*score=roc\_auc\_score(Y\_test,Y\_scores)*

*score*

0.945714814404145

**ACCURACY SCORE**

*Y\_pred=np.where(Y\_scores>=optimal\_threshold,1,0)*

*print(accuracy\_score(Y\_pred,Y\_test))*

Best score of 90.45 and a very similar F1 score

**Calculating the AUC (Area Under Curve):**

* **Trapezoidal Rule**: The AUC is essentially the area under the ROC curve. It can be calculated using the trapezoidal rule, which estimates the area under a curve by summing up the areas of trapezoids formed by consecutive points on the curve.

**The area under a trapezoid is ½\*height\*(sum of parallel sides )**

Now if we apply the same on the AUC curve, we will divide it into n trapezoids, higher the n more accurate the results will be , **height = FPR(i+1)-FPR(i)**

**Sum of parallel sides = TPR(i+1)+TPR(i)**

**And we will add the area under all the n trapezoids to get the AUC !!**