HW3

1. Problem 1 : Apply Variational Autoencoder on the CIFAR10 Dataset.

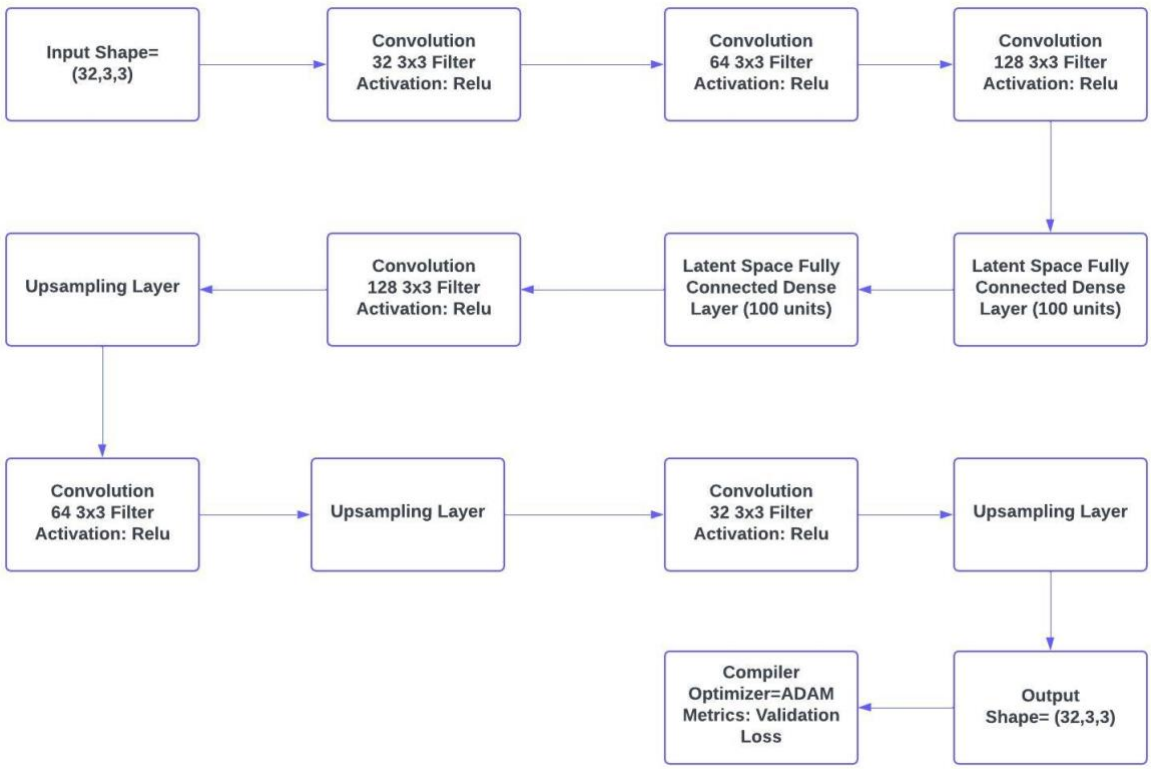
a. Use 3 convolutional layers in the encoder and 3 deconvolutional layers (Conv2DTranspose/ upscale) in the decoder.

a. Use 3 convolutional layers in the encoder and 3 deconvolutional layers (Conv2DTranspose/ upscale) in the decoder.

```
# Defining the input shape
input_shape = (32, 32, 3)
```

```
# Defining the size of the latent space
latent_dim = 100
```

Model Architecture



This is a Variational Autoencoder (VAE) architecture that takes an input image of shape (32, 32, 3) and produces a reconstructed image of the same shape.

The encoder model takes the input image and applies three convolutional layers with kernel size 3 and 32, 64, and 128 filters respectively, followed by max-pooling layers. The output of the third convolutional layer is flattened and passed through two dense layers, one with latent dimension of 100 (latent_space_mean) and the other with the same latent dimension (latent_space_log_var).

The decoder model takes the latent representation produced by the encoder model and passes it through three transposed convolutional layers with kernel size 3 and 128,64 and 32 respectively, followed by up-sampling layers. The output of the last transposed convolutional layer has a sigmoid activation function that produces the reconstructed image.

The VAE model combines the encoder and decoder models and introduces a regularization term (kl_loss) that encourages the encoder to produce a normally distributed latent space representation. The regularization term is added as a loss to the VAE model.

Hyper-Parameter Used

Encoder					
Number of Convolution Layers: 3					
Convolutions Layers	**Filter Size**	**Stride**	**Number of Filters**	**Padding**	**Activation Function**
Convo Layer 1	3x3	1	32	Same	Relu
Convo Layer 2	3x3	1	64	Same	Relu
Convo Layer 3	3x3	1	128	Same	Relu
Latent Space (Latent Dim = 100)					
Fully Connected Layer	Units				
Dense Layer 1	100 (equal to Latent Dim)				
Dense Layer 2	100 (equal to Latent Dim)				
Decoder					
Dense Layer	Units= 2048				
Number of De-Convolution Layers: 3					
Deconvolutions Layers	**Filter Size**	**Stride**	**Number of Filters**	**Padding**	**Activation Function**
De-convo Layer 1	3x3	1	128	Same	Relu
De-convo Layer 2	3x3	1	64	Same	Relu
De-convo Layer 3	3x3	1	32	Same	Relu
Output Layer	**Convo2dTranspose**			**Activation Function**	
	size= 3 and filters = 3			Sigmoid	
Reconstruction Loss					
KL Divergence (random normal distribution)					
Model Compilation					
Loss	**Optimizer**	**Learning Rate**	**Evaluation Metric**		
Binary Cross entropy	Adam	0.0001	Validation Loss		
Training					
Epochs	**Batch Size**		**Validation Split**		
50	32		0.2		

```
Model: "encoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d (Conv2D)             (None, 32, 32, 32)        896

 max_pooling2d (MaxPooling2D)  (None, 16, 16, 32)      0

 conv2d_1 (Conv2D)           (None, 16, 16, 64)        18496

 max_pooling2d_1 (MaxPooling2D)  (None, 8, 8, 64)      0

 conv2d_2 (Conv2D)           (None, 8, 8, 128)         73856

 flatten (Flatten)           (None, 8192)              0

 dense (Dense)               (None, 100)               819300

 dense_1 (Dense)             (None, 100)               819300

=================================================================
Total params: 1,731,848
Trainable params: 1,731,848
Non-trainable params: 0
_________________________________________________________________
```

The encoder model has a total of 1,731,848 parameters, all of which are trainable. The model consists of 6 layers: 3 convolutional layers, 2 max pooling layers, and 2 dense layers. The input shape of the model is (None, 32, 32, 3), and the output shapes of each layer are as follows:

conv2d_9: (None, 32, 32, 32) max_pooling2d_6: (None, 16, 16, 32) conv2d_10: (None, 16, 16, 64) max_pooling2d_7: (None, 8, 8, 64) conv2d_11: (None, 8, 8, 128) dense_8: (None, 100) dense_9: (None, 100) The encoder model takes an input tensor of shape (None, 32, 32, 3) and outputs two tensors, representing the mean and log variance of the latent space, respectively. The encoder model contains a total of 3 convolutional layers with 32, 64, and 128 filters, respectively. Each convolutional layer is followed by a max pooling layer to reduce the spatial dimensions of the output. The output of the last convolutional layer is flattened and passed through two dense layers, each with 100 units, to obtain the mean and log variance of the latent space.

```
Model: "decoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 100)]             0

 dense_2 (Dense)             (None, 2048)              206848

 reshape (Reshape)           (None, 4, 4, 128)         0

 conv2d_transpose (Conv2DTra  (None, 4, 4, 128)        147584
 nspose)

 up_sampling2d (UpSampling2D  (None, 8, 8, 128)        0
 )

 conv2d_transpose_1 (Conv2DT  (None, 8, 8, 64)         73792
 ranspose)

 up_sampling2d_1 (UpSampling  (None, 16, 16, 64)       0
 2D)

 conv2d_transpose_2 (Conv2DT  (None, 16, 16, 32)       18464
 ranspose)

 up_sampling2d_2 (UpSampling  (None, 32, 32, 32)       0
 2D)

 conv2d_transpose_3 (Conv2DT  (None, 32, 32, 3)        867
 ranspose)

=================================================================
Total params: 447,555
Trainable params: 447,555
Non-trainable params: 0
```

The decoder model has a total of 447,555 trainable parameters. It has 4 layers including 3 convolutional transpose layers and 3 upsampling layers. The summary of the decoder model is as follows:

InputLayer: an input layer that takes in a 2D tensor of shape (None, 100) (where None indicates a variable batch size, and 100 is the number of input features).

Dense: a fully connected layer with 2048 neurons and a ReLU activation function. This layer takes the input tensor and applies a linear transformation to generate a new tensor of shape (None, 2048).

Reshape: a layer that reshapes the tensor into a 4D tensor of shape (None, 4, 4, 128).

Conv2DTranspose: a convolutional layer with 128 filters, a kernel size of (3, 3), a stride of (1, 1), and a padding of 'same'. This layer upsamples the tensor to a new shape of (None, 4, 4, 128) using transposed convolution.

UpSampling2D: a layer that doubles the spatial dimensions of the tensor using bilinear interpolation. This layer increases the tensor shape to (None, 8, 8, 128).

Conv2DTranspose: another convolutional layer with 64 filters, a kernel size of (3, 3), a stride of (1, 1), and a padding of 'same'. This layer upsamples the tensor to a new shape of (None, 8, 8, 64). UpSampling2D: another layer that doubles the spatial dimensions of the tensor, increasing the tensor shape to (None, 16, 16, 64).

Conv2DTranspose: another convolutional layer with 32 filters, a kernel size of (3, 3), a stride of (1, 1), and a padding of 'same'. This layer upsamples the tensor to a new shape of (None, 16, 16, 32). UpSampling2D: another layer that doubles the spatial dimensions of the tensor, increasing the tensor shape to (None, 32, 32, 32).

Conv2DTranspose: a final convolutional layer with 3 filters, a kernel size of (3, 3), a stride of (1, 1), and a padding of 'same'. This layer produces the final output tensor of shape (None, 32, 32, 3)

```
# Define the VAE model
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def call(self, inputs):
        z_mean, z_log_var = self.encoder(inputs)
        epsilon = keras.backend.random_normal(shape=(keras.backend.shape(z_mean)[0], latent_dim),
                                              mean=0.0, stddev=1.0)
        z = z_mean + keras.backend.exp(0.5*z_log_var) * epsilon
        reconstructed = self.decoder(z)
        kl_loss = -0.5 * keras.backend.mean(1.0 + z_log_var - keras.backend.square(z_mean) -
                                            keras.backend.exp(z_log_var), axis=-1)
        self.add_loss(kl_loss)
        return reconstructed

vae = VAE(encoder, decoder)
```

This is the call method of a custom Keras model that implements a Variational Autoencoder. The method takes inputs as argument and returns reconstructed outputs.

The encoder component of the model takes the inputs and maps them to a latent space using a mean and log variance layer. The method then samples from the normal distribution with mean z_mean and variance z_log_var, using the reparameterization trick to add noise to the output.

The decoder component of the model takes the sample from the latent space and generates the reconstructed output.

The method calculates the Kullback-Leibler divergence between the distribution from the encoder and a standard normal distribution, which is used as a regularization term to prevent overfitting. This regularization term is added to the loss function of the model during training.

Compiling the Model

```
[119] vae.compile(optimizer=keras.optimizers.Adam(learning_rate=0.0001))
```

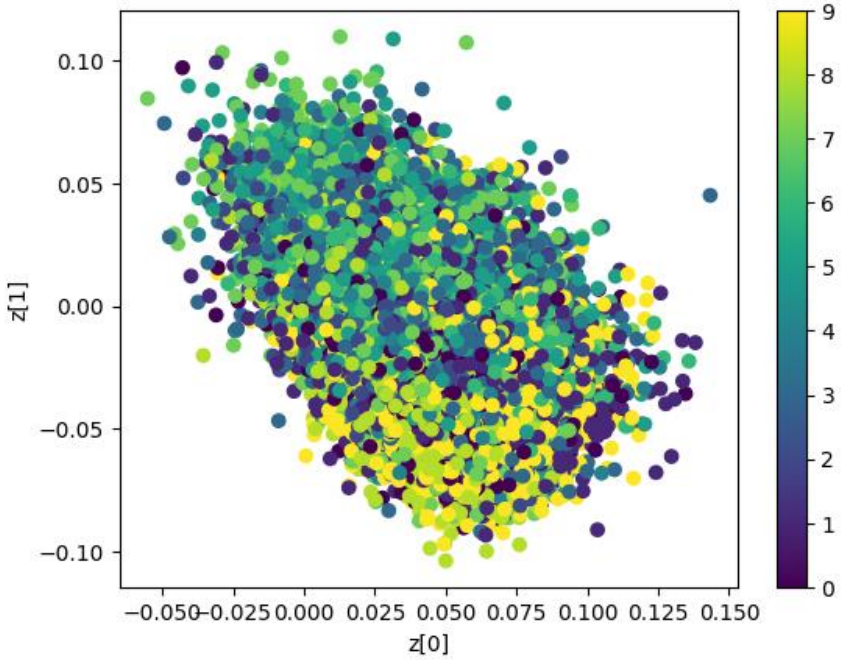
Training the Model

```
[120] # Training the VAE model

      history= vae.fit(x_train, x_train, epochs=50, batch_size= 32,validation_data=(x_test, x_test))
```

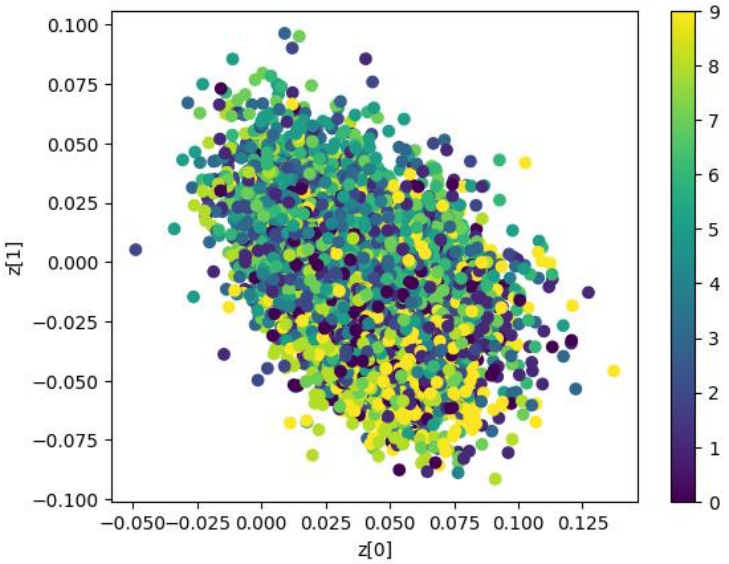
```
Epoch 2/50
1563/1563 [==============================] - 6s 4ms/step - loss: 2.3771e-09 - val_loss: 1.9497e-09
Epoch 3/50
1563/1563 [==============================] - 8s 5ms/step - loss: 1.4161e-09 - val_loss: 5.1504e-10
Epoch 4/50
1563/1563 [==============================] - 7s 5ms/step - loss: 1.5215e-09 - val_loss: -4.6870e-10
Epoch 5/50
1563/1563 [==============================] - 7s 5ms/step - loss: 1.6497e-09 - val_loss: 4.7963e-09
```

b. Provide a plot for latent space.

```
z_mean, _ = encoder.predict(x_train)

# Plot the latent space
plt.scatter(z_mean[:, 0], z_mean[:, 1], c=y_train)
plt.colorbar()
plt.xlabel("z[0]")
plt.ylabel("z[1]")
plt.show()
```



```
z_mean, _ = encoder.predict(x_test)

# Plot the latent space
plt.scatter(z_mean[:, 0], z_mean[:, 1], c=y_test)
plt.colorbar()
plt.xlabel("z[0]")
plt.ylabel("z[1]")
plt.show()
```



c. Provide minimum 5 newly generated fake images from using the decoder of the network.

```
import numpy as np
import matplotlib.pyplot as plt

# Generate 5 random points in latent space
latent_points = np.random.normal(size=(5, latent_dim))

# Use the decoder to generate fake images from the latent points
fake_images = decoder.predict(latent_points)

# Rescale the pixel values from [-1,1] to [0,1] for visualization
fake_images = (fake_images + 1) / 2.0

# Plot the generated fake images
fig, axs = plt.subplots(nrows=1, ncols=5, figsize=(12, 6))
for i, ax in enumerate(axs.flatten()):
    ax.imshow(fake_images[i])
    ax.axis('off')
plt.show()
```

Results

Model was trained for 50 epochs with batch size =32. Adam optimizer was used to optimise the loss function with learning rate 0.0001. From the figure we can see overlapping latent space for the target class which has 10 classes. Also predicition was done for 5 random image by adding noise. Pictures are not clear, they can be optimise futher my training model for more epochs.

Problem 2 : Use the IMDB Movie review dataset:

a. Perform Text Preprocessing
   a. Tokenization
   b. Stopwords removing
   c. HTML removing
   d. Convert to lower case
   e. Lemmatization/stemming

Use the IMDB Movie review dataset

```
# importing the libraries
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import pandas as pd
import numpy as np
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
```

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive

```python
# Defining the stopword list and the lemmatizer
stopwords_list = stopwords.words('english')
lemmatizer = WordNetLemmatizer()
```

```python
# Defining a function for preprocessing
def preprocess_text(text):

    # Removing HTML tags
    text = re.sub('<[^>]+>', '', text)

    # Converting to lower case
    text = text.lower()

    # Tokenizing
    words = nltk.word_tokenize(text)

    # Removing stopwords and non-alphabetic words
    words = [word for word in words if word.isalpha() and word not in stopwords_list]

    # Lemmatizing
    words = [lemmatizer.lemmatize(word) for word in words]

    # Joining the words back into a single string
    text = ' '.join(words)

    return text
```

```python
df.head(5)
```

	review	sentiment	preprocessed_review
0	One of the other reviewers has mentioned that ...	positive	one reviewer mentioned watching oz episode hoo...
1	A wonderful little production. The...	positive	wonderful little production filming technique ...
2	I thought this was a wonderful way to spend ti...	positive	thought wonderful way spend time hot summer we...
3	Basically there's a family where a little boy ...	negative	basically family little boy jake think zombie ...
4	Petter Mattei's "Love in the Time of Money" is...	positive	petter mattei love time money visually stunnin...

b. Build the following sentiment analysis models and create a performance comparison table:

a. TF-IDF + GausianNB

Hyper-Paramter used in Tf-idf

> max_features: maximum number of features (i.e., unique words) to be included in the vocabulary, based on the frequency of each word in the corpus. This parameter is used to limit the memory usage and improve the computational efficiency of the model. This is set to 5000.

> stop_words: a list of words to be excluded from the vocabulary. In this case, we are using the built-in stop words provided by scikit-learn for the English language.

```
# building TF-IDF feature vectors
tfidf = TfidfVectorizer(stop_words='english',max_features= 5000)
X_train_tfidf = tfidf.fit_transform(X_train)
X_test_tfidf = tfidf.transform(X_test)
```

```
# label encoding the y_train and y_test
from sklearn.preprocessing import LabelEncoder
label= LabelEncoder()
label.fit(y_train)
y_train_encoded= label.transform(y_train)
y_test_encoded= label.transform(y_test)
```

Training

```
[67] # Initializing the Gaussian Naive Bayes classifier
     gnb = GaussianNB()
```

```
# Fitting the model on the dense training data

gnb.fit(X_train_tfidf.toarray(), y_train_encoded)
```

▾ GaussianNB

GaussianNB()

Metrics

```
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score
import matplotlib.pyplot as plt
import seaborn as sns
```

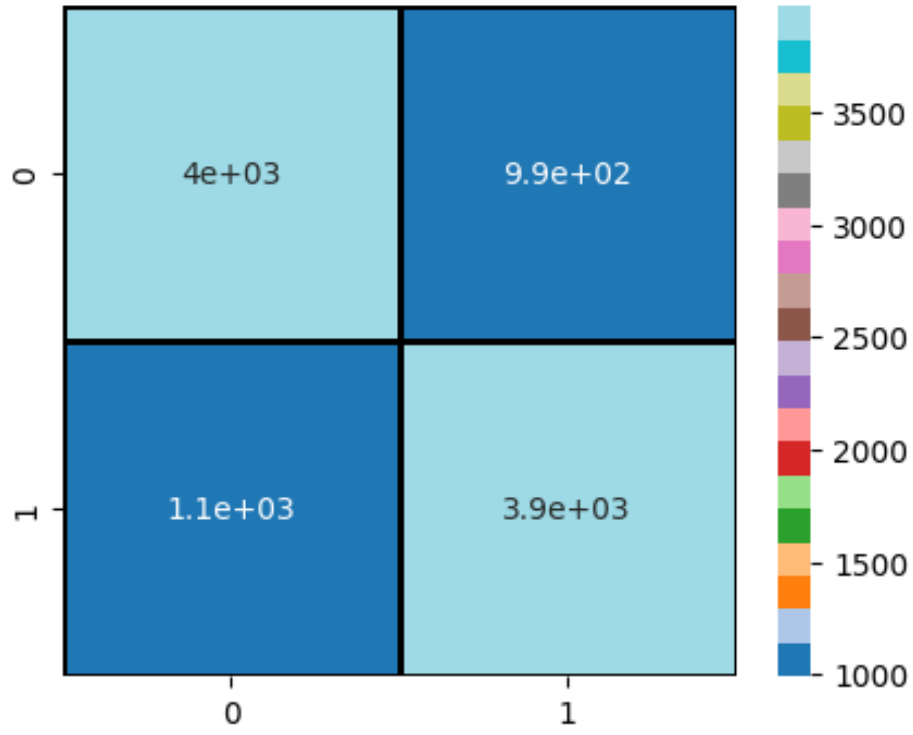
```
# calculating the accuracy score of the Word2Vec + GaussianNB model
accuracy_tfidf = accuracy_score(y_test_encoded, y_pred_tfidf)
print("Accuracy Score is ", accuracy_tfidf)
```

```
Accuracy Score is  0.789
```

```
cm = confusion_matrix(y_test_encoded, y_pred_tfidf)
#printing the confusion matrix to see corrrect and wrong predicted values
print(cm)
```

```
[[3972  989]
 [1121 3918]]
```

Confusion Matrix




```
precision1 = precision_score(y_test_encoded, y_pred_tfidf)
print("Precision Score is ", precision1)
```

```
Precision Score is  0.7984511921744447
```

```
recall1 = recall_score(y_test_encoded, y_pred_tfidf)
print("Recall Score is ", recall1)
```

```
Recall Score is  0.7775352252431038
```

```
f1score1 = f1_score(y_test_encoded, y_pred_tfidf)
print("F1 Score is ", f1score1)
```

```
F1 Score is  0.7878544138347076
```

b. Word2Vec (CBoW) + GausianNB

▾ Hyper-parameter used in Word2Vec Model

> vector_size: the dimensionality of the word vectors, set to 300.
>
> window: the maximum distance between the current and predicted word within a sentence, set to 5.
>
> min_count: the minimum frequency of a word to be included in the vocabulary, set to 1.
>
> workers: the number of worker threads used to train the model in parallel, set to 4.
>
> epochs: the number of training iterations over the corpus, set to 50.
>
> sg: it is used for skip gram by default is 1 which is skip gram . for cBOW it is set at 0

```
# building Word2Vec embeddings for train data
sentences = [review.split() for review in X_train]
word2vec_model = Word2Vec(sentences, vector_size=300, window=5,epochs= 50, min_count=1 , workers=4, sg=0)
```

```
X_train_w2v = np.array([np.mean([word2vec_model.wv[word] for word in review.split() if word in word2vec_model.wv.index_t
```

```
# building Word2Vec embeddings for test data
sentences = [review.split() for review in X_test]
word2vec_model = Word2Vec(sentences, vector_size=300, window=5,epochs= 50, min_count=1 , workers=4, sg=0)
```

```
X_test_w2v = np.array([np.mean([word2vec_model.wv[word] for word in review.split() if word in word2vec_model.wv.index_to
```

```
# building Gaussian Naive Bayes classifier using Word2Vec embeddings
nb_w2v = GaussianNB()
```

Training the Model

```
[91] nb_w2v.fit(X_train_w2v, y_train_encoded)
```

▾ GaussianNB

GaussianNB()

Metrics

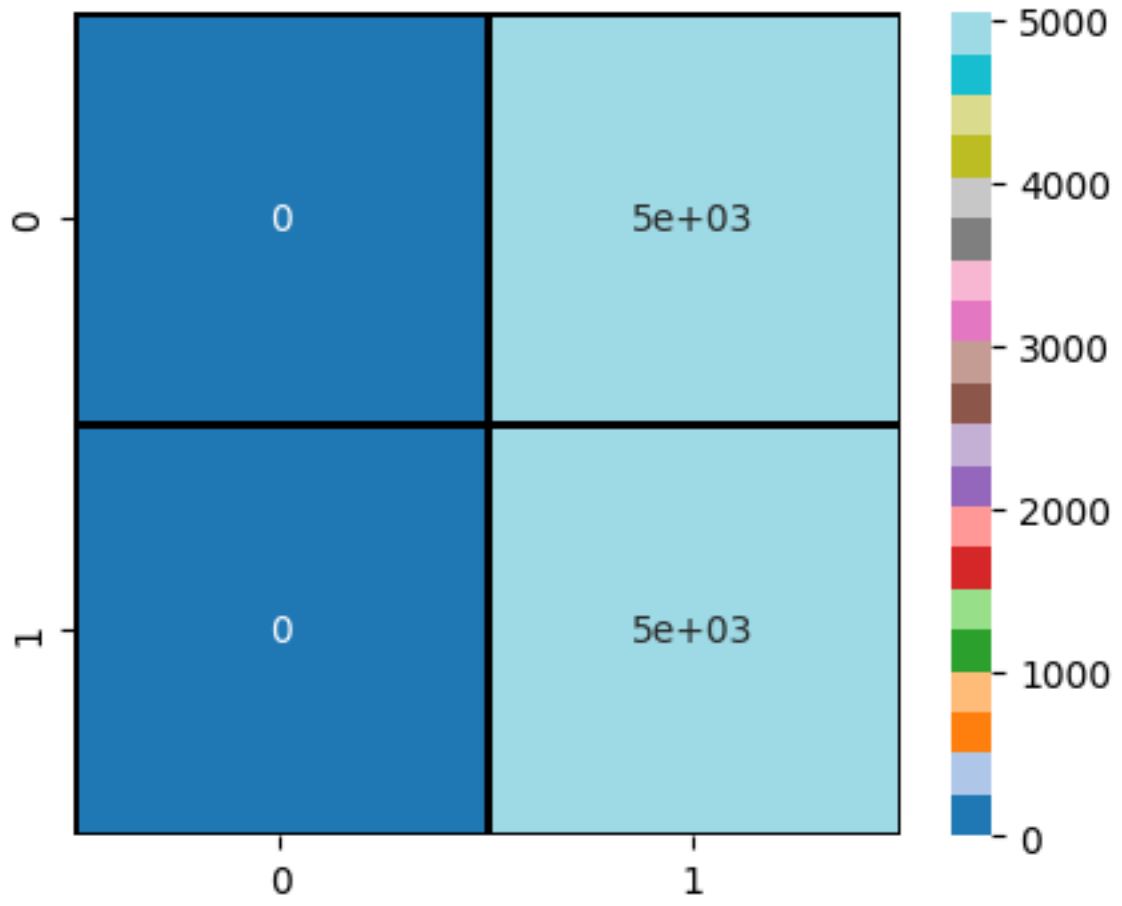
```
[195] # calculate the accuracy score of the Word2Vec + GaussianNB model
      accuracy_w2v = accuracy_score(y_test_encoded, y_pred_w2v)
      print("Accuracy Score is ", accuracy_w2v)
```

```
      Accuracy Score is  0.5039
```

```
[196] cm = confusion_matrix(y_test_encoded, y_pred_w2v)
      #printing the confusion matrix to see corrrect and wrong predicted values
      print(cm)
```

```
      [[   0 4961]
       [   0 5039]]
```

Confusion Matrix



```
precision2 = precision_score(y_test_encoded, y_pred_w2v)
print("Precision Score is ", precision2)
```

```
Precision Score is  0.5039
```

```
recall2 = recall_score(y_test_encoded, y_pred_w2v)
print("Recall Score is ", recall2)
```

```
Recall Score is  1.0
```

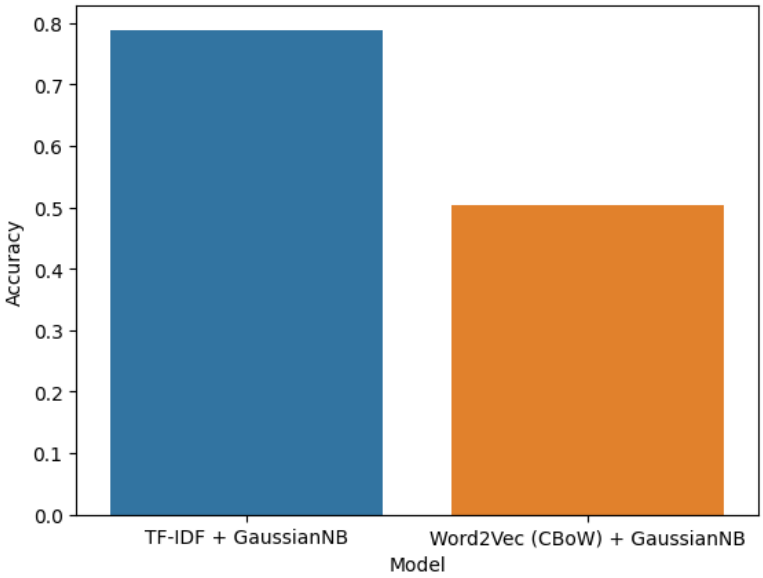
```
f1score2 = f1_score(y_test_encoded, y_pred_w2v)
print("F1 Score is ", f1score2)
```

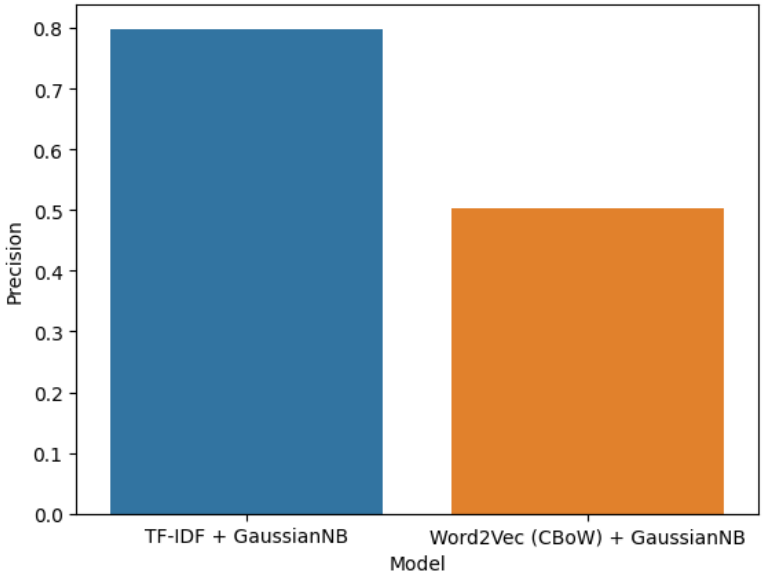
```
F1 Score is  0.6701243433738946
```

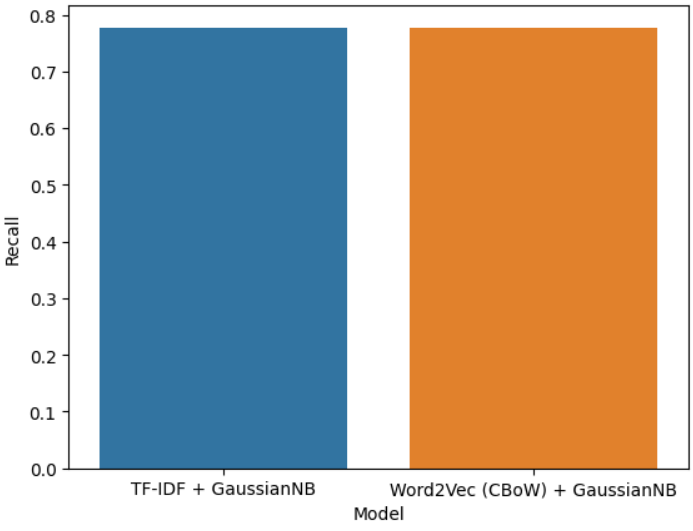
Perfomance Comparison

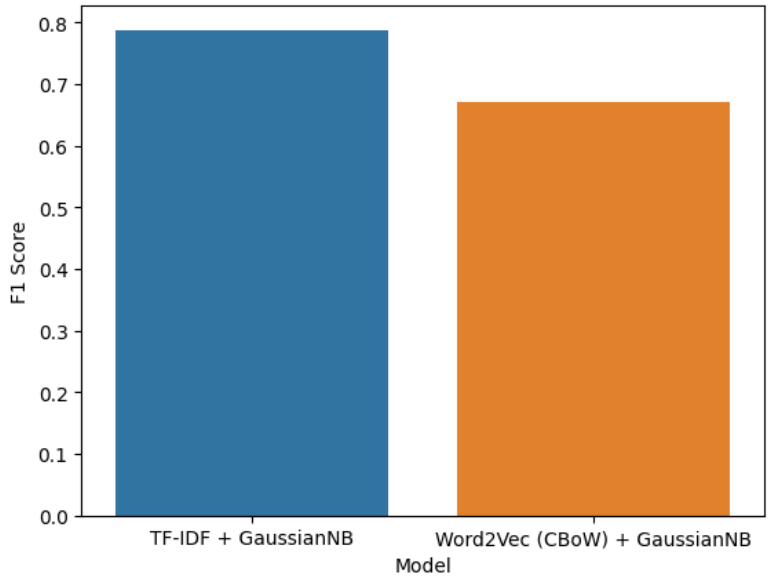
```
# create a performance comparison table
performance_table = pd.DataFrame({
    'Model': ['TF-IDF + GaussianNB', 'Word2Vec (CBoW) + GaussianNB'],
    'Accuracy': [accuracy_tfidf, accuracy_w2v],
    'Precision':[precision1, precision2],
    'Recall':[recall1, recall1],
    'F1 Score':[f1score1, f1score2]
})

print(performance_table)
```

```
                         Model  Accuracy  Precision    Recall  F1 Score
0           TF-IDF + GaussianNB    0.7890   0.798451  0.777535  0.787854
1  Word2Vec (CBoW) + GaussianNB    0.5039   0.503900  0.777535  0.670124
```


0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Accuracy
TF-IDF + GaussianNB
Word2Vec (CBoW) + GaussianNB
Model


0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Precision
TF-IDF + GaussianNB
Word2Vec (CBoW) + GaussianNB
Model


0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Recall
TF-IDF + GaussianNB
Word2Vec (CBoW) + GaussianNB
Model



Results

From the Gaussian Naive Baye's Model we can see that when words were tranformed into numeric form using TFIDF accuracy was 79% and when words were transformed using word2vec accuracy was 50%. However, recall score for both the models were approximately similar whereas for precision, f1 score model with tfidf performed better.