sentiment_embeddings

December 11, 2020

If you are running this locally, please make sure you have Cuda Enabled, and Tensorflow 2.x and Pytorch 1.x installed with GPu support.

Whether locally or on Colab, if it's the first time running this notebook, run the following command to install all necessary dependencies (Pytorch and Tensorflow must be installed manually).

[ ]:
```
!pip3 install -r requirements.txt
```

1 Sentiment Analysis performance benchmark

Authors : *Ihab Bendidi*, *Yousra Bourkiche*, *Clément Siegrist*, *Kaouter Berrahal*

In general, documents with similar sentiments, would be close to each other in the embeddings feature space. This can become another method to judge the performance of sentiment analysis models.

In this work, we aim to perform a benchmark of recent sentiment analysis works and models, reproduce their results, and judge their performance in comparison to baseline methods.

This work has the following plan :

I - Processing & Exploratory Data Analysis - *Understanding the data* - *Text Preprocessing*

II - Sentiment classification models - *Bert Model* - *LSTM recurrent model* - *Baseline method : textblob*

III - Document Embeddings - *Training doc2vec* - *Doc2vec sentiment classifier*

IV - Model performance visualisation - *Bert model* - *LSTM model* - *Logreg model* - *Textblob*

Throughout this project, we are working with a clean Ubuntu 20.04 distribution, on Python 3.7. We are going to use libraries such as Pytorch 1.8 and Tensorflow 2, which would be using GPU. For installation of dependencis, you can install them using the `requirements.txt` file. More details can be found in the `README.md` file. You can find below the specs of the GPU we have been using for our experiments.

[2]:
```
!nvidia-smi
```

```
Thu Dec 10 15:36:35 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 455.45.01    Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
```

```
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   61C    P8    11W /  70W |      0MiB / 15079MiB |      0%      Default |
|                               |                      |                  ERR! |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

1.0.1 *Abstract*

Lorem Ipsum

[ ]:
```
import transformers
from transformers import BertModel, BertTokenizer, AdamW,␣
 ↪get_linear_schedule_with_warmup
import torch

import numpy as np
import pandas as pd
import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc
from mpl_toolkits.mplot3d import Axes3D


from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix,␣
 ↪classification_report,precision_score,accuracy_score,f1_score
from sklearn.linear_model import LogisticRegression
from sklearn import utils, decomposition, svm
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA

from wordcloud import WordCloud, STOPWORDS

from collections import defaultdict
from textwrap import wrap
from textblob import TextBlob
```

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential, Model, load_model
from keras.layers import Dense, Embedding, LSTM, SpatialDropout1D,Input
from keras.optimizers import Adam
from keras import backend as K
from keras.callbacks import ModelCheckpoint

from gensim.test.utils import common_texts
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
import nltk
from nltk.corpus import stopwords
from tqdm import tqdm
import multiprocessing



from torch import nn, optim
from torch.utils.data import Dataset, DataLoader
import torch.nn.functional as F

import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
%config InlineBackend.figure_format='retina'

sns.set(style='whitegrid', palette='muted', font_scale=1.2)

HAPPY_COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#ADFF02",␣
 ↪"#8F00FF"]

sns.set_palette(sns.color_palette(HAPPY_COLORS_PALETTE))

rcParams['figure.figsize'] = 12, 8

RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device

!python3 -m textblob.download_corpora
```

1.0.2 I - Processing & Exploratory Data Analysis

1 - Understanding the data We would be working throughout this experiment on a Twitter dataset of complains and reviews of people about airline companies. These reviews got a sentiment already labelled, even though its apparent it was also generated with another model. This might incur incorrect labels, but as long as the models we use are trained on the same dataset, it would still fit our comparison purpose, even though it won't have much use in production. While this dataset is limiting, especially in terms of size, it would also be a good baseline to start our work with.

We import out dataset in the form of a `.csv` file to put it in a Pandas Dataframe.

[113]:
```
df = pd.read_csv('tweets.csv')
```

[114]:
```
# The size of our dataset
df.shape
```

[114]:
```
(14640, 15)
```

As we explore the values of our dataset, we notice that much of the data is not pertinent for our task. As most of the data is for airlines to know the general opinions about their services. The most interesting columns for us are `airline_sentiment`, and the `text` columns.

[115]:
```
# Show the first five values of our dataframe
df.head()
```

[115]:

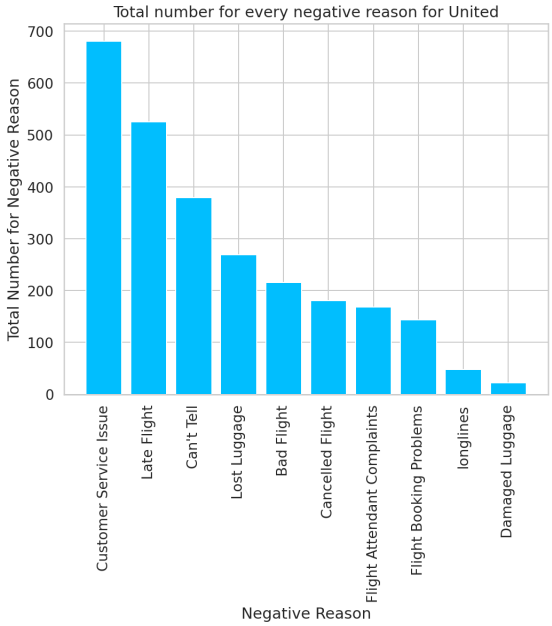
	tweet_id	…	user_timezone
0	570306133677760513	…	Eastern Time (US & Canada)
1	570301130888122368	…	Pacific Time (US & Canada)
2	570301083672813571	…	Central Time (US & Canada)
3	570301031407624196	…	Pacific Time (US & Canada)
4	570300817074462722	…	Pacific Time (US & Canada)

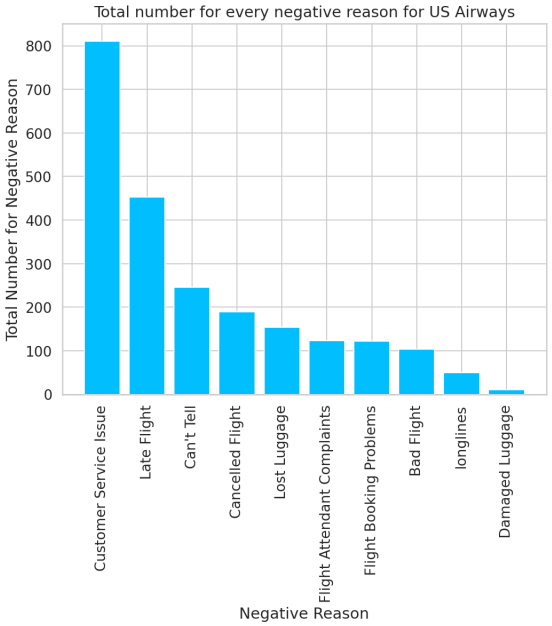
[5 rows x 15 columns]

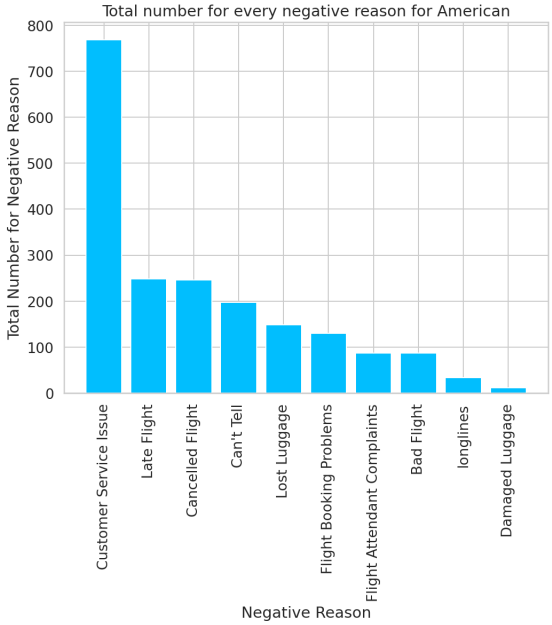
We'll begin with a cursory look through the dataset, to understand its various components. One such analysis is exploring the reasons users gave negative reviews to the airlines.

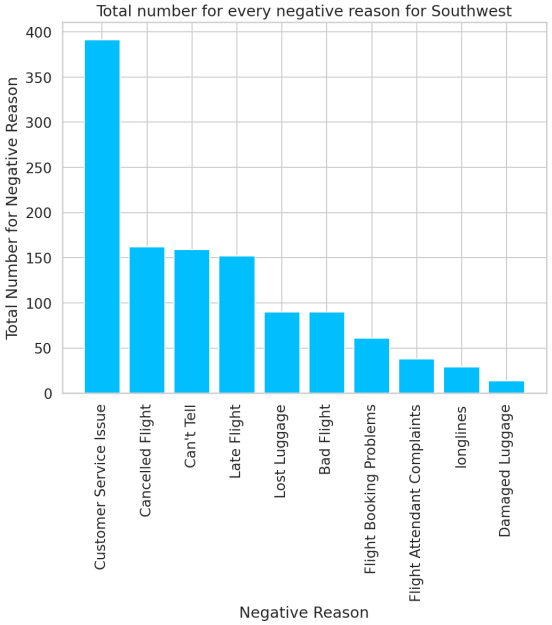
[117]:
```
def reason_each_flight(airline):
    data = df[df['airline'] == airline]
    data = data['negativereason']
    data_count = data.value_counts()
    List = data.value_counts().index.tolist()
    Index = range(1,(len(data.unique())))
    plt.bar(Index, data_count)
    plt.xlabel('Negative Reason')
    plt.ylabel('Total Number for Negative Reason')
    plt.title('Total number for every negative reason for ' + airline)
    plt.xticks(Index, List,rotation = 90)
```

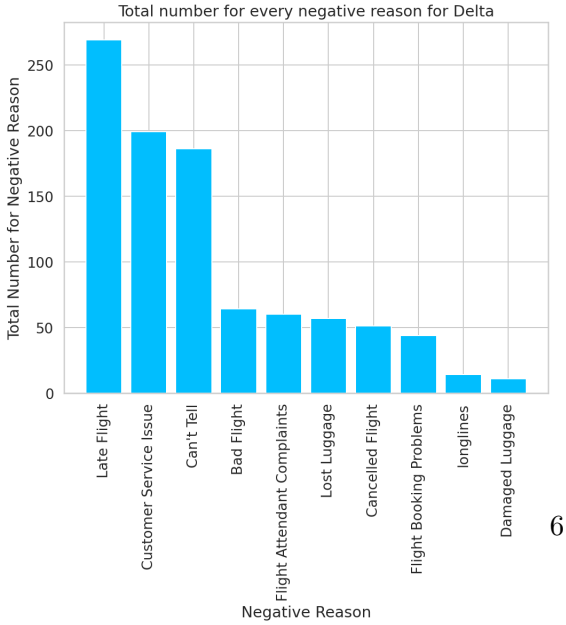
```
Air = df['airline'].value_counts().index.tolist()
plt.figure(1,figsize=(15, 25))
plt.subplot(321)
reason_each_flight(Air[0])
plt.subplot(322)
reason_each_flight(Air[1])
plt.subplot(323)
reason_each_flight(Air[2])
plt.subplot(324)
reason_each_flight(Air[3])
plt.subplot(325)
reason_each_flight(Air[4])
plt.subplot(326)
reason_each_flight(Air[5])
plt.tight_layout()
```

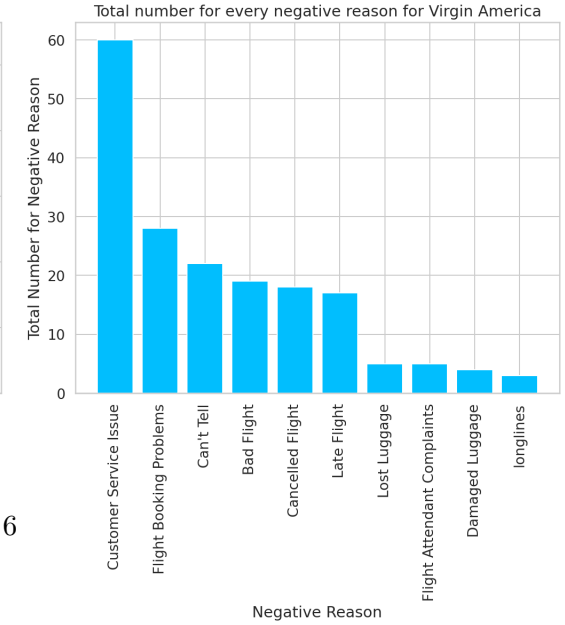

Total number for every negative reason for United
Total Number for Negative Reason
Customer Service Issue
Late Flight
Can't Tell
Lost Luggage
Bad Flight
Cancelled Flight
Flight Attendant Complaints
Flight Booking Problems
longlines
Damaged Luggage
Negative Reason


Total number for every negative reason for US Airways
Total Number for Negative Reason
Customer Service Issue
Late Flight
Can't Tell
Cancelled Flight
Lost Luggage
Flight Attendant Complaints
Flight Booking Problems
Bad Flight
longlines
Damaged Luggage
Negative Reason


Total number for every negative reason for American
Total Number for Negative Reason
Customer Service Issue
Late Flight
Cancelled Flight
Can't Tell
Lost Luggage
Flight Booking Problems
Flight Attendant Complaints
Bad Flight
longlines
Damaged Luggage
Negative Reason


Total number for every negative reason for Southwest
Total Number for Negative Reason
Customer Service Issue
Cancelled Flight
Can't Tell
Late Flight
Lost Luggage
Bad Flight
Flight Booking Problems
Flight Attendant Complaints
longlines
Damaged Luggage
Negative Reason


Total number for every negative reason for Delta
Total Number for Negative Reason
Late Flight
Customer Service Issue
Can't Tell
Bad Flight
Flight Attendant Complaints
Lost Luggage
Cancelled Flight
Flight Booking Problems
longlines
Damaged Luggage
Negative Reason


Total number for every negative reason for Virgin America
Total Number for Negative Reason
Customer Service Issue
Flight Booking Problems
Can't Tell
Bad Flight
Cancelled Flight
Late Flight
Lost Luggage
Flight Attendant Complaints
Damaged Luggage
longlines
Negative Reason

We notice that throughout most airline companies, the same distribution of issues appears, with some very slight variations. We can intuitively think that the model can learn to map such reasons if noticed in the text of the review, toward being of negative category.

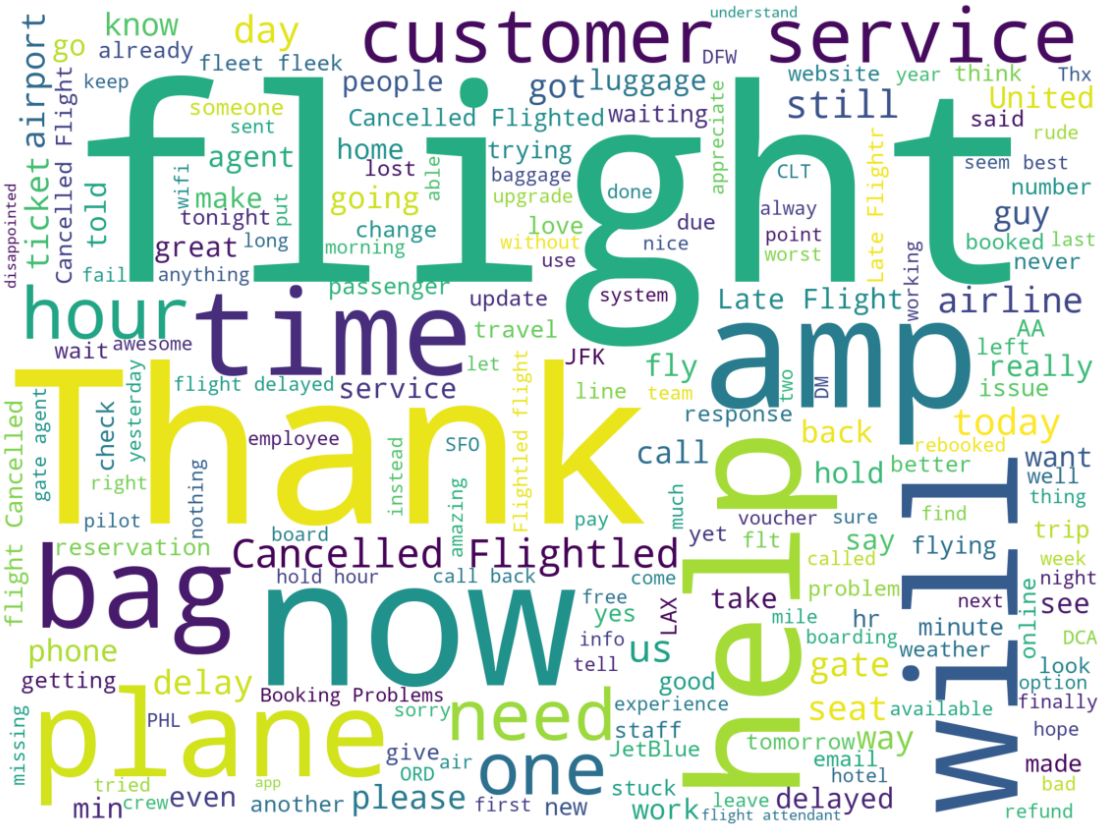
We follow this by an analysis of the word occurences in negative reviews :

[123]:
```
df_cloud = df[df['airline_sentiment']=='negative']
# join tweets to a single string
words = ' '.join(df['text'])
# remove URLs, RTs, and twitter handles
no_urls_no_tags = " ".join([word for word in words.split()
                            if 'http' not in word
                                and not word.startswith('@')
                                and word != 'RT'
                            ])

wordcloud = WordCloud(stopwords=STOPWORDS,
                      background_color='white',
                      width=2000,
                      height=1500
                     ).generate(no_urls_no_tags)

plt.figure(1,figsize=(10, 10))
plt.imshow(wordcloud)
plt.axis('off')
plt.show()
```

Similarly for the neutral category reviews :

[124]:
```
# let's make Word Cloud for neutral mood
df_cloud_neutral = df[df['airline_sentiment']=='neutral']
# join tweets to a single string
words = ' '.join(df['text'])
# remove URLs, RTs, and twitter handles
no_urls_no_tags = " ".join([word for word in words.split()
                            if 'http' not in word
                                and not word.startswith('@')
                                and word != 'RT'
                            ])

wordcloud = WordCloud(stopwords=STOPWORDS,
                      background_color='white',
                      width=2000,
                      height=1500
                     ).generate(no_urls_no_tags)
plt.figure(1,figsize=(10, 10))
plt.imshow(wordcloud)
plt.axis('off')
```

```
plt.show()
```



And finally for the positive category :

[125]:
```
df_cloud_positive = df[df['airline_sentiment']=='positive']
# join tweets to a single string
words = ' '.join(df['text'])
# remove URLs, RTs, and twitter handles
no_urls_no_tags = " ".join([word for word in words.split()
                            if 'http' not in word
                                and not word.startswith('@')
                                and word != 'RT'
                            ])

wordcloud = WordCloud(stopwords=STOPWORDS,
                      background_color='white',
                      width=2000,
                      height=1500
                     ).generate(no_urls_no_tags)
plt.figure(1,figsize=(10, 10))
plt.imshow(wordcloud)
```

```
plt.axis('off')
plt.show()
```

We notice that the the same general word distribution appears throughout the reviews of all three categories. This shows that we can't just use models based on word occurence in our model, but use model that can also take into account the context of the word in the review.

[8]:
```
# importing the interesting & pertinent data in our dataset into a new dataframe
tweets =␣
 ↪df[['tweet_id','text','airline_sentiment','airline_sentiment_confidence']]
tweets.head()
```

[8]:

	tweet_id	…	airline_sentiment_confidence
0	570306133677760513	…	1.0000
1	570301130888122368	…	0.3486
2	570301083672813571	…	0.6837
3	570301031407624196	…	1.0000
4	570300817074462722	…	1.0000

[5 rows x 4 columns]

To explore our data further, it is a must to check whether there are missing values. Though as seen below, it appears there is none in the data that matters.

[9]:
```
# The count of null or nan values in the dataset
tweets.isnull().sum()
```

[9]:
```
tweet_id                        0
text                            0
airline_sentiment               0
airline_sentiment_confidence    0
dtype: int64
```

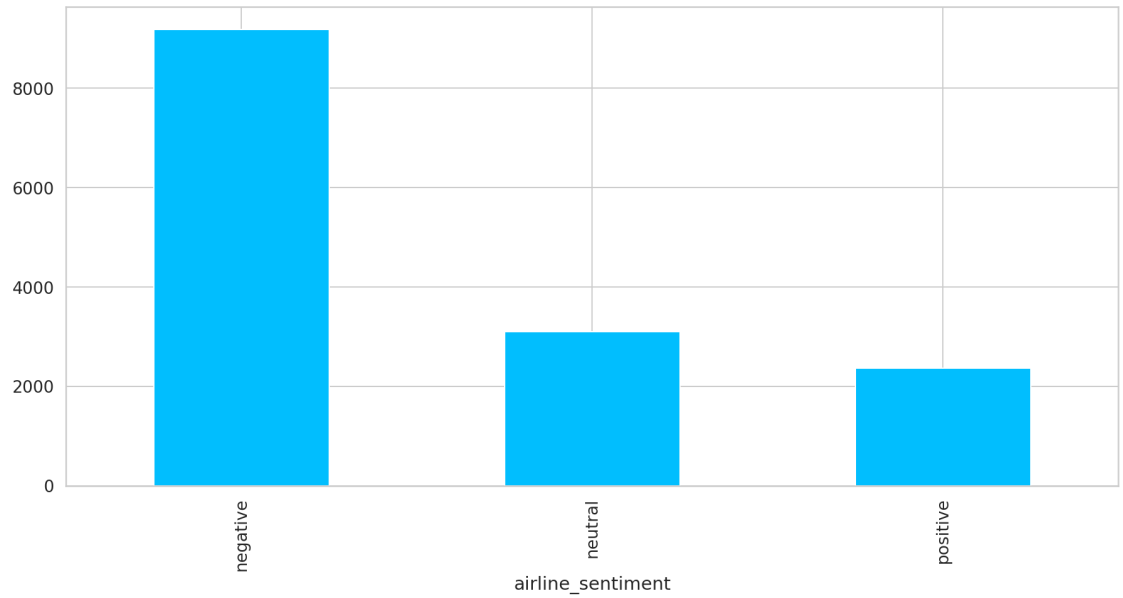
Our next focus would be to know the existing categories in the sentiment labels in `airline_sentiment`. We would then create a plot of their occurence in our data.

[10]:
```
# Unique values in a Series of the dataframe
tweets['airline_sentiment'].unique()
```

[10]:
```
array(['neutral', 'positive', 'negative'], dtype=object)
```

[11]:
```
tweets.groupby('airline_sentiment')['tweet_id'].count().plot.
↪bar(figsize=(15,7),grid=True)
```

[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc41da1ce10>
```



As it appears, our data is heavily imbalanced, as the `negative` category got up three times more occurences than the `neutral` or `positive` category. This is a matter that would need to be taken into account when judging the model performance, to avoid any bias in its predictions.

On another note, we take a peak at the format of the tweets and texts we'll be handling below.

[12]:
```
print(tweets['text'][1000])
```

```
@united how can you not put my bag on plane to Seattle. Flight 1212. Waiting  in
line to talk to someone about my bag. Status should matter.
```

2 - Text preprocessing In this part, we'll be handling the text data, cleaning it from discrepancies, and preparing it to be in a format that is friendly to processing by our models, that would come further on the line.

We can notice in the following tweets, and through other careful explorations of the dataset, that airline company names are often tagged at the beginning of every tweet.

[13]:
```
print(tweets['text'][10])
print(tweets['text'][1000])
```

```
@VirginAmerica did you know that suicide is the second leading cause of death
among teens 10-24
@united how can you not put my bag on plane to Seattle. Flight 1212. Waiting  in
line to talk to someone about my bag. Status should matter.
```

While this appears harmless, we could also forward a hypothesis that our models, further down the line, might overfit and learn to map company names tagged in the tweet directly to a sentiment category. That would be especially possible of some company names got some specific categories of sentiment review much more often than others.

To make certain of whether this hypothesis is unfounded or not, we search here for the occurences of the different categories of sentiments for each company name tagged.

To achieve that, we start by extracting the tags from every tweet.

[14]:
```
# This helper function returns the tags appearing mostly at the beginning of␣
↪each tweet
def get_tags(row):
    splits = row['text'].split(' ')
    # Some simple text processing to have all tags in the same format
    row['tags'] = ''.join(e for e in splits[0] if e.isalnum()).lower()
    # Correcting a very common typo
    if row['tags'] == 'jetblue':
        row['tags'] = 'jetblues'
    return row

# Extracting the tags (without deletion) from the tweets into their own columns
df = tweets.apply(get_tags,axis=1)
```

Our dataframe has a new Series inside, of the tags of the companies occuring in the tweets, in an unified format.

[15]:
```
df.head()
```

[15]:
```
             tweet_id  …           tags
0  570306133677760513  …  virginamerica
1  570301130888122368  …  virginamerica
2  570301083672813571  …  virginamerica
3  570301031407624196  …  virginamerica
4  570300817074462722  …  virginamerica

[5 rows x 5 columns]
```

Our next task would be to group the tags and sentiment categories together, to be able to count the occurences of each category in each tag.

[16]:
```
# Defining a helper function to count the occurences of each category of␣
 ↪sentiment for each tag
def count_tags(df, col, new_col):
    df[new_col] = df[col].count()
    return df

# Grouping all tags and sentiment categories together, in order to count␣
 ↪occurences of categories
# of sentiments for each category, then saving the resulting Series into our␣
 ↪DataFrame
df['tag_count'] = df.groupby(['tags','airline_sentiment']).
 ↪apply(count_tags,'tweet_id','count')['count']
```

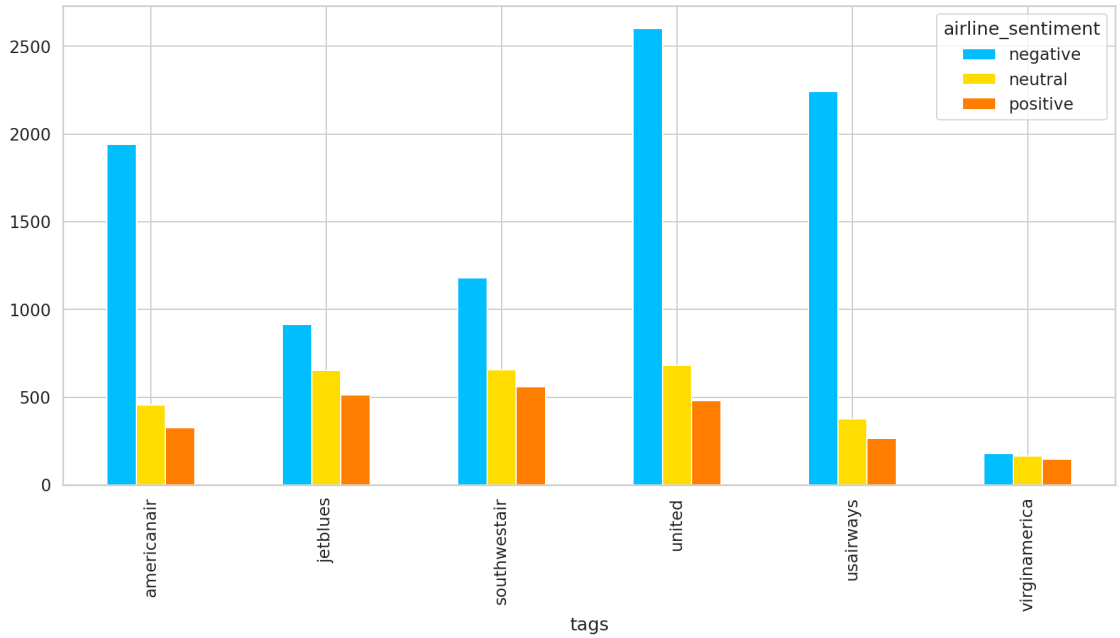
While this would enable us to have the exact count of occurences of each category of sentiment for each tag, we would find that there is some noise in our data. We would be filtering that noise by only keeping the tags whose count is beyond 15. The number of the filtered tags is negligible.

[17]:
```
temp_df = df[df['tag_count']>15].copy()
```

Following that, we create a cross table of the values of the count of occurences for each sentiment category, for each company tag, then we plot a bar plot to visualize the differences between the occurences.

[18]:
```
# Create a cross table of the count of the occurences of tags/sentiment classes
cross_df = pd.crosstab(temp_df.tags,temp_df.
 ↪airline_sentiment,values=temp_df['tag_count'],
                       margins=False,aggfunc='mean')

# Visualize our results in a bar plot
cross_df.plot.bar(figsize=(15,7),grid=True)
```

[18]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc41b57b358>
```



The earlier results in the plot have proven our hypothesis right : Some company tags got a very large ratio of negative sentiments. We believe these tags might make the model learn to map the presence of the tags to a sentiment category, mostly the **negative** category, as it is the major category for the tags **americanair**, **united** and **usairways**.

Our next task would involve deleting them from our tweets, to handle the text without bias.

[19]:
```
# A helper function that would remove the tags from the text and save it into a␣
↪new column
def remove_tags(row):
    splits = row['text'].split(' ')
    row['review'] = ' '.join(splits[1:])
    return row

# Removing the tags from the text and saving the cleaned up text in a new column
tweets = df.
↪apply(remove_tags,axis=1)[['tweet_id','text','review','airline_sentiment']]

tweets.head()
```

[19]:

	tweet_id	…	airline_sentiment
0	570306133677760513	…	neutral
1	570301130888122368	…	positive
2	570301083672813571	…	neutral
3	570301031407624196	…	negative
4	570300817074462722	…	negative

```
[5 rows x 4 columns]
```

Before going further, we will encode our labels, the `airline_sentiment` categories, to make it easier to work on this classification task.

[20]:
```
# Define a helper function for one hot encoding
def one_hot_encode(row):
    if row['airline_sentiment'] == "negative":
        row['target'] = 0
    if row['airline_sentiment'] == "neutral":
        row['target'] = 1
    if row['airline_sentiment'] == "positive":
        row['target'] = 2
    return row

# The class names : 0 -> negative, 1 -> neutral, 2 -> positive
class_names = ['negative', 'neutral', 'positive']

# One hot encode the labels of our dataset
tweets = tweets.
 ↪apply(one_hot_encode,axis=1)[['review','target','tweet_id','text','airline_sentiment']]

tweets.head()
```

[20]:

	review	…	airline_sentiment
0	What @dhepburn said.	…	neutral
1	plus you've added commercials to the experienc…	…	positive
2	I didn't today… Must mean I need to take ano…	…	neutral
3	it's really aggressive to blast obnoxious "ent…	…	negative
4	and it's a really big bad thing about it	…	negative

```
[5 rows x 5 columns]
```

Our following goal would be to preprocess further our text data for NLP. One of the models we would be working is BERT. BERT (introduced in this paper) stands for Bidirectional Encoder Representations from Transformers.

Machine Learning models don't work with raw text. You need to convert text to numbers (of some sort). BERT requires even more attention. Here are the requirements:

- Add special tokens to separate sentences and do classification
- Pass sequences of constant length (introduce padding)
- Create array of 0s (pad token) and 1s (real token) called attention mask

We can use a cased and uncased version of BERT and tokenizer. Intuitively, the cased version would work better, since "BAD" might convey more sentiment than "bad".

Different from classical text processing, which we would perform on the text data for another model later on, we would be using a a pre-trained BertTokenizer:

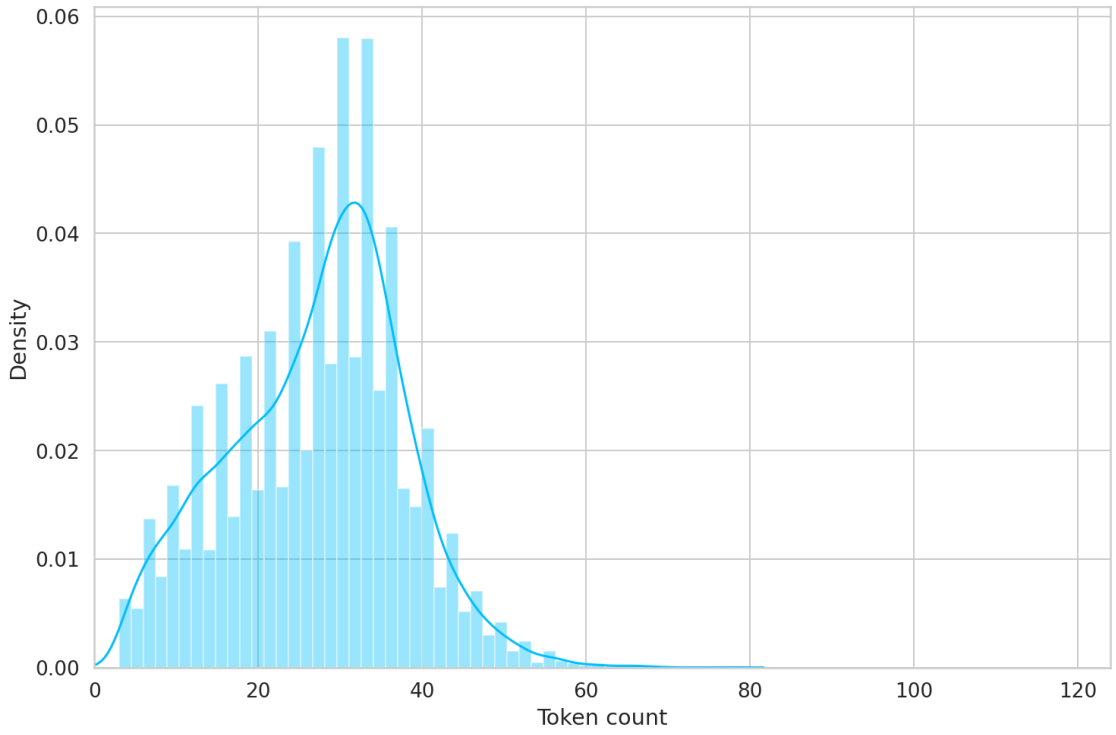
[21]:
```
PRE_TRAINED_MODEL_NAME = 'bert-base-cased'
tokenizer = BertTokenizer.from_pretrained(PRE_TRAINED_MODEL_NAME)
```

BERT works with fixed-length sequences. We'll use a simple strategy to choose the max length, so that we pad each token until it achieves the max length. Let's store the token length of each text :

[22]:
```
# List of the length of each generated token
token_lens = []

# Storing the length of the token of each text in our dataset
for txt in tweets.review:
    tokens = tokenizer.encode(txt, max_length=512)
    token_lens.append(len(tokens))

# Plotting the distribution of the lengths of the tokens in our dataset
sns.distplot(token_lens)
plt.xlim([0, 124])
plt.xlabel('Token count')
```

```
Truncation was not explicitly activated but `max_length` is provided a specific
value, please use `truncation=True` to explicitly truncate examples to max
length. Defaulting to 'longest_first' truncation strategy. If you encode pairs
of sequences (GLUE-style) with the tokenizer you can select this strategy more
precisely by providing a specific strategy to `truncation`.
```

[22]:
```
Text(0.5, 0, 'Token count')
```



Most of the texts contain between 20 and 40 token, but to stay on the safe side, we'll take a maximum length of 70 token.

[23]:
```
MAX_LEN = 70
```

Our following task is to manipulate our dataset to make it into a Pytorch dataset, that would make it simple to work on it to train our BERT model later on.

[24]:
```
class GPReviewDataset(Dataset):
    def __init__(self, reviews, targets, tokenizer, max_len):
        self.reviews = reviews
        self.targets = targets
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.reviews)

    def __getitem__(self, item):
        review = str(self.reviews[item])
        target = self.targets[item]

        # Tokenizing the texts, while also including special tokens
        # for start and end of the text, as well as padding
        encoding = self.tokenizer.encode_plus(
          review,
          add_special_tokens=True,
          max_length=self.max_len,
          return_token_type_ids=False,
          pad_to_max_length=True,
          return_attention_mask=True,
          return_tensors='pt', # We return here the data as Pytorch Tensor
        )

        return {
          'review_text': review,
          'input_ids': encoding['input_ids'].flatten(),
          'attention_mask': encoding['attention_mask'].flatten(),
          'targets': torch.tensor(target, dtype=torch.long)
        }
```

The tokenizer is doing most of the heavy lifting for us. We also return the review texts, so it'll be easier to evaluate the predictions from our model. Let's split the data, with 70% training, 10% validation set, and 20% test set :

[25]:
```
# Splitting the test data
df_temp, df_test = train_test_split(tweets, test_size=0.2,␣
 ↪random_state=RANDOM_SEED)

# Splitting the training and validation data
df_train, df_val = train_test_split(df_temp, test_size=0.1,␣
 ↪random_state=RANDOM_SEED)

print("The size of training set is : " + str(df_train.shape[0]))
print("The size of validation set is : " + str(df_val.shape[0]))
print("The size of test set is : " + str(df_test.shape[0]))
```

```
The size of training set is : 10540
The size of validation set is : 1172
The size of test set is : 2928
```

We also need to create a couple of data loaders for Pytorch :

[26]:
```
def create_data_loader(df, tokenizer, max_len, batch_size):
    ds = GPReviewDataset(
        reviews=df.review.to_numpy(),
        targets=df.target.to_numpy(),
        tokenizer=tokenizer,
        max_len=max_len
      )

    return DataLoader(
        ds,
        batch_size=batch_size,
        num_workers=4
      )


BATCH_SIZE = 16

train_data_loader = create_data_loader(df_train, tokenizer, MAX_LEN, BATCH_SIZE)
val_data_loader = create_data_loader(df_val, tokenizer, MAX_LEN, BATCH_SIZE)
test_data_loader = create_data_loader(df_test, tokenizer, MAX_LEN, BATCH_SIZE)
```

A simple exploration of an example batch of our data loader would show the following :

[27]:
```
# Loading a batch of 16 observation from the training set, tokenized and␣
 ↪processed
data = next(iter(train_data_loader))
print(data.keys())
```

```
dict_keys(['review_text', 'input_ids', 'attention_mask', 'targets'])
```

[28]:
```
# For each batch of 16 observations
print(data['input_ids'].shape)
print(data['attention_mask'].shape)
print(data['targets'].shape)
```

```
torch.Size([16, 70])
torch.Size([16, 70])
torch.Size([16])
```

1.0.3 II - Sentiment classification models

1 - Bert Model In this step, we'll be using a basic pretrained Bert Model, and build our sentiment classifier on top of it. We'll be importing it as follow :

[29]:
```
# When first executing this, it would take time to download the model from web␣
 ↪if it doesn't exist locally
# Make sure to have your internet activated the first time you run this cell␣
 ↪locally
bert_model = BertModel.from_pretrained(PRE_TRAINED_MODEL_NAME)
```

We would need to add classification layers at the top of our BERT model. But to do that, we would need to know the sizes of its current outputs. We can do that by testing it on the earlier batch of data :

[30]:
```
# Making test predictions using the raw BERT model on the batch data we made␣
 ↪earlier
model_outs = bert_model(
  input_ids=data['input_ids'],
  attention_mask=data['attention_mask']
)

# The last hidden layer in the raw BERT model
last_hidden_state = model_outs[0]
pooled_output = model_outs[1]
```

[31]:
```
last_hidden_state.shape
```

[31]:
```
torch.Size([16, 70, 768])
```

We have the hidden state for each of our 70 tokens (the length of our example sequence). 768 is the number of hidden units in the feedforward-networks.

We can now use this knowledge to create a classifier that uses the BERT model.

[32]:
```
# Defining our classifier that we would build on top of the raw bert model
class SentimentClassifier(nn.Module):

    def __init__(self, n_classes):
        super(SentimentClassifier, self).__init__()
```

```
        self.bert = BertModel.from_pretrained(PRE_TRAINED_MODEL_NAME)
        # Adding drop out, keeping 70% of the last neurons of the raw BERT model
        self.drop = nn.Dropout(p=0.3)
        # The last linear layer for multiclass classification
        self.out = nn.Linear(self.bert.config.hidden_size, n_classes)

    # Forward propagation function
    def forward(self, input_ids, attention_mask):
        model_outs = self.bert(
          input_ids=input_ids,
          attention_mask=attention_mask
        )
        last_hidden_state = model_outs[0]
        pooled_output = model_outs[1]
        output = self.drop(pooled_output)
        return self.out(output)
```

Our classifier delegates most of the heavy lifting to the BertModel. We use a dropout layer for some regularization and a fully-connected layer for our output. We're returning the raw output of the last layer since that is required for the cross-entropy loss function in PyTorch to work.

We now pass the instance of our classifier to GPU :

[33]:
```
# Initializing the classifier to output three class categories
model = SentimentClassifier(len(class_names))

# Running the classifier on GPU
model = model.to(device)
```

Similarly, we'll move the example batch of our training data to the GPU:

[34]:
```
# Saving the input data on GPU
input_ids = data['input_ids'].to(device)
attention_mask = data['attention_mask'].to(device)

print(input_ids.shape) # batch size x seq length
print(attention_mask.shape) # batch size x seq length
```

```
torch.Size([16, 70])
torch.Size([16, 70])
```

To get the predicted probabilities from our trained model, we'll apply the softmax function to the outputs:

[35]:
```
F.softmax(model(input_ids, attention_mask), dim=1)
```

[35]:
```
tensor([[0.4191, 0.1861, 0.3947],
        [0.5449, 0.1308, 0.3243],
        [0.2789, 0.1379, 0.5832],
```

```
        [0.5505, 0.1105, 0.3390],
        [0.5695, 0.1898, 0.2407],
        [0.3330, 0.1613, 0.5057],
        [0.4120, 0.0986, 0.4894],
        [0.2955, 0.2547, 0.4498],
        [0.3355, 0.1382, 0.5263],
        [0.2590, 0.2234, 0.5176],
        [0.3927, 0.1524, 0.4549],
        [0.4370, 0.1933, 0.3697],
        [0.5702, 0.0900, 0.3397],
        [0.4757, 0.1866, 0.3377],
        [0.3993, 0.1894, 0.4113],
        [0.3546, 0.1869, 0.4585]], device='cuda:0', grad_fn=<SoftmaxBackward>)
```

Following that, to reproduce the training procedure from the BERT paper, we'll use the AdamW optimizer provided by Hugging Face. It corrects weight decay, so it's similar to the original paper. We'll also use a linear scheduler with no warmup steps:

[36]:
```
# Number of epochs for training
EPOCHS = 10

# Adam optimizer
optimizer = AdamW(model.parameters(), lr=2e-5, correct_bias=False)
total_steps = len(train_data_loader) * EPOCHS

scheduler = get_linear_schedule_with_warmup(
  optimizer,
  num_warmup_steps=0,
  num_training_steps=total_steps
)

# Cross Entropy Loss Function
loss_fn = nn.CrossEntropyLoss().to(device)
```

The BERT authors have some recommendations for fine-tuning:

- Batch size: 16, 32
- Learning rate (Adam): 5e-5, 3e-5, 2e-5
- Number of epochs: 2, 3, 4

We're going to ignore the number of epochs recommendation but stick with the rest. Note that increasing the batch size reduces the training time significantly, but gives you lower accuracy.

We'll begin with writing a helper function for computing the precision of our model for each epoch, by computing the precision for each class, while taking label imbalance into account, and computing the weighted average precision, mode details here :

[37]:
```
# Computing multiclass precision for the outputs of the model
def compute_precision(outputs, labels):
```

```
  op = outputs.cpu()
  la = labels.cpu()
  _, preds = torch.max(op, dim=1)
  # We choose 'weighted' averaging of the precision of each label because it␣
↪takes into account the imbalance of labels in our tweets dataset
  # other viable averaging methods are 'micro'
  return torch.tensor(precision_score(la,preds, average='weighted'))
```

Let's continue with writing a helper function for training our model for one epoch:

[38]:
```
def train_epoch(
  model,
  data_loader,
  loss_fn,
  optimizer,
  device,
  scheduler,
  n_examples
):
    model = model.train()

    losses = []
    correct_predictions = 0
    precision = 0
    i = 0

    for d in data_loader: # d is our batch of observations
        i += 1

        #preparing inputs
        input_ids = d["input_ids"].to(device)
        attention_mask = d["attention_mask"].to(device)
        targets = d["targets"].to(device)

        # Running inference
        outputs = model(
          input_ids=input_ids,
          attention_mask=attention_mask
        )

        # softmax on the outputs to get the classes
        _, preds = torch.max(outputs, dim=1)

        # Computing loss
        loss = loss_fn(outputs, targets)

        # Counting the correct occurences
```

```
        correct_predictions += torch.sum(preds == targets)

        losses.append(loss.item())

        # Computing the precision (true positives/true positives + false␣
↪positives)
        # for each class and label, and find their average weighted by support
        precision += compute_precision(outputs,targets)

        # Backward propagation
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        scheduler.step()
        optimizer.zero_grad()

    return correct_predictions.double() / n_examples, np.mean(losses),␣
↪precision/i # Accuracy, loss, precision
```

The scheduler gets called every time a batch is fed to the model. We're avoiding exploding gradients by clipping the gradients of the model using clip_grad_norm_.

We'll then write another helper function that helps us evaluate the model on a given data loader:

[39]:
```
def eval_model(model, data_loader, loss_fn, device, n_examples):
    model = model.eval()

    losses = []
    correct_predictions = 0
    precision = 0
    i = 0

    with torch.no_grad():
        for d in data_loader:
            i += 1

            # Preparing inputs
            input_ids = d["input_ids"].to(device)
            attention_mask = d["attention_mask"].to(device)
            targets = d["targets"].to(device)

            # Running inference using the model
            outputs = model(
                input_ids=input_ids,
                attention_mask=attention_mask
              )

            # Running softmax on the outputs
```

```
            _, preds = torch.max(outputs, dim=1)

            # Computing loss function
            loss = loss_fn(outputs, targets)

            # Counting the correct occurences
            correct_predictions += torch.sum(preds == targets)

            # Computing the precision (true positives/true positives + false␣
↪positives)
            # for each class and label, and find their average weighted by␣
↪support
            precision += compute_precision(outputs,targets)

            losses.append(loss.item())

    return correct_predictions.double() / n_examples, np.mean(losses),␣
↪precision/i # Accuracy, loss, precision
```

Using those two, we can write our training loop. We'll also store the training history:

[39]:
```
%%time

# creating history object, to keep history of results
history = defaultdict(list)
best_accuracy = 0

# Iterating on epochs
for epoch in range(EPOCHS):

    print(f'Epoch {epoch + 1}/{EPOCHS}')
    print('-' * 10)

    # Running the training on the whole dataset (divided into batches)
    train_acc, train_loss, train_precision = train_epoch(
        model,
        train_data_loader,
        loss_fn,
        optimizer,
        device,
        scheduler,
        len(df_train)
      )

    # Results of training
    print(f'Train : loss {train_loss}, accuracy {train_acc}, precision␣
↪{train_precision}')
```

```
    # Running the validation on the whole dataset (divided into batches)
    val_acc, val_loss, val_precision = eval_model(
        model,
        val_data_loader,
        loss_fn,
        device,
        len(df_val)
      )

    # Results of validation
    print(f'Val : loss {val_loss}, accuracy {val_acc}, precision␣
↪{val_precision}')
    print()

    # Keeping all results into history
    history['train_acc'].append(train_acc)
    history['train_loss'].append(train_loss)
    history['train_precision'].append(train_precision)
    history['val_acc'].append(val_acc)
    history['val_loss'].append(val_loss)
    history['val_precision'].append(val_precision)

    # Saving the model with the best accuracy
    if val_acc > best_accuracy:
        torch.save(model.state_dict(), 'best_model_state.bin')
        best_accuracy = val_acc
```

```
Epoch 1/10
----------
Train : loss 0.5336638307410867, accuracy 0.7937381404174573, precision
0.8128075415774574
Val : loss 0.5097645931449291, accuracy 0.8225255972696246, precision
0.8327996585809085

Epoch 2/10
----------
Train : loss 0.28815961526127387, accuracy 0.8996204933586338, precision
0.9180582505506636
Val : loss 0.5173394468839507, accuracy 0.840443686006826, precision
0.8561103404853408

Epoch 3/10
----------
Train : loss 0.16401975514467562, accuracy 0.9564516129032258, precision
0.9654173358886948
Val : loss 0.7103318145272095, accuracy 0.8421501706484642, precision
```

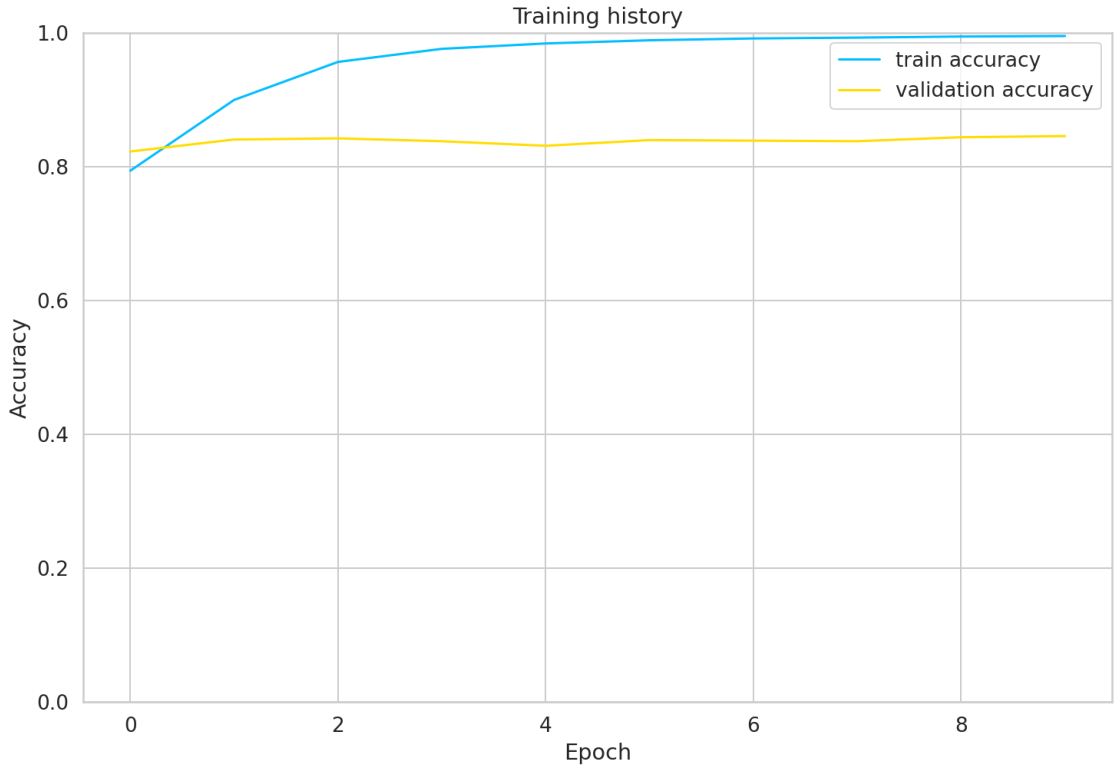
```
0.8633779977529982

Epoch 4/10
----------
Train : loss 0.09937565966243594, accuracy 0.9759962049335863, precision
0.9819180361242204
Val : loss 0.8404175414030466, accuracy 0.8378839590443686, precision
0.8533704705579709

Epoch 5/10
----------
Train : loss 0.06870397610728712, accuracy 0.9840607210626185, precision
0.9883391456798138
Val : loss 0.9713083968113957, accuracy 0.8310580204778157, precision
0.854324595730846

Epoch 6/10
----------
Train : loss 0.04804065153093716, accuracy 0.9888045540796964, precision
0.9918197537056608
Val : loss 1.005422699391823, accuracy 0.8395904436860069, precision
0.8619236314548818

Epoch 7/10
----------
Train : loss 0.037875482650134185, accuracy 0.9915559772296015, precision
0.9938949520488787
Val : loss 1.059087144507479, accuracy 0.8387372013651877, precision
0.8655118272305776

Epoch 8/10
----------
Train : loss 0.030661280900688648, accuracy 0.9927893738140418, precision
0.9948578482721125
Val : loss 1.1028201002250049, accuracy 0.8378839590443686, precision
0.8594707727520233

Epoch 9/10
----------
Train : loss 0.02341174374569288, accuracy 0.9944022770398482, precision
0.9960071302517245
Val : loss 1.072292411613366, accuracy 0.8438566552901023, precision
0.863226933539434

Epoch 10/10
----------
Train : loss 0.019540477724168562, accuracy 0.9951612903225806, precision
0.9967395485984258
```

```
Val : loss 1.084189734550278, accuracy 0.8455631399317406, precision
0.8639499272311778

CPU times: user 17min 20s, sys: 8min 46s, total: 26min 7s
Wall time: 26min 26s
```

Note that we're storing the state of the best model, indicated by the highest validation accuracy. The results would take some time. With our GPU, it took around 1 hour.

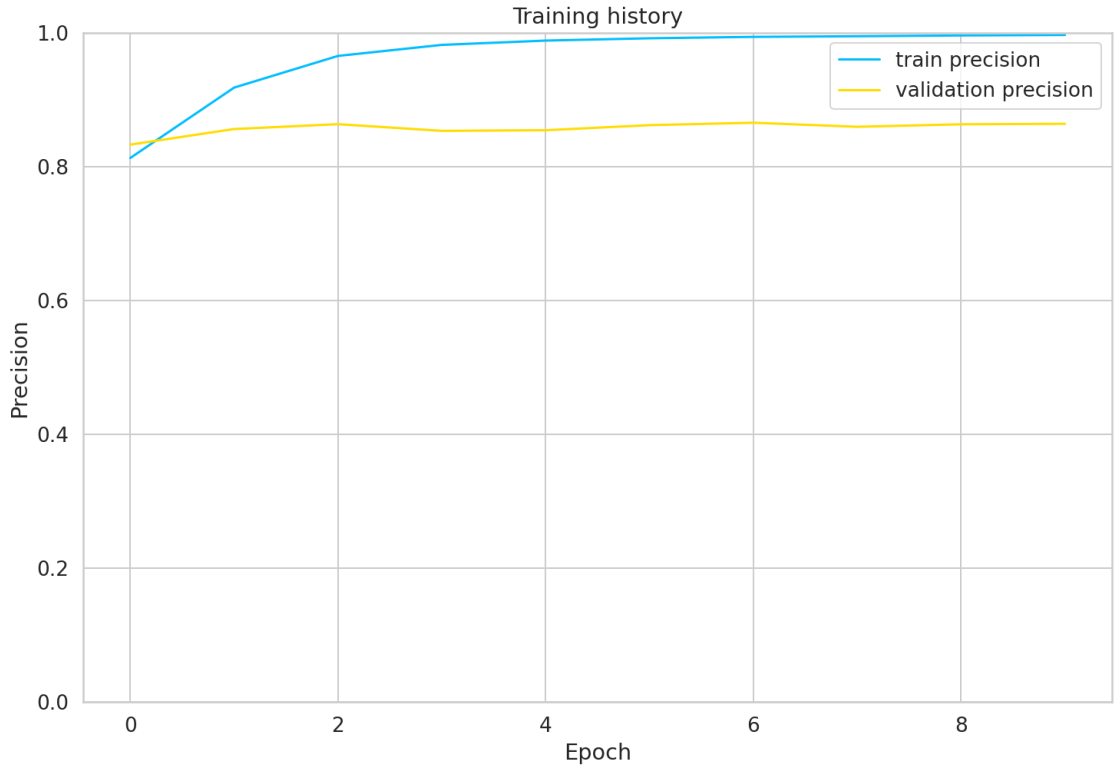
The following shows the result of the changes in accuracy and precision across epochs :

[40]:
```
# Plotting the results of the training and validation accuracy using the stored␣
↪information in history
plt.plot(history['train_acc'], label='train accuracy')
plt.plot(history['val_acc'], label='validation accuracy')

plt.title('Training history')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.ylim([0, 1]);
```



[41]:
```
# Plotting the results of the training and validation precision using the␣
↪stored information in history
```

```
plt.plot(history['train_precision'], label='train precision')
plt.plot(history['val_precision'], label='validation precision')

plt.title('Training history')
plt.ylabel('Precision')
plt.xlabel('Epoch')
plt.legend()
plt.ylim([0, 1]);
```



We read the best model of our training and store it into gpu for future usage :

[40]:
```
model = SentimentClassifier(len(class_names))
model.load_state_dict(torch.load('best_model_state.bin'))
model = model.to(device)
```

To get a good glimpse on how good our model is at predicting sentiment, we'll compute accuracu on the test data :

[41]:
```
test_acc, _, _ = eval_model(
  model,
  test_data_loader,
  loss_fn,
  device,
```

```
  len(df_test)
)

test_acc.item()
```

[41]: 0.8415300546448088

The accuracy is about the same on the test set than on validation set. Our model seems to generalize well.

We'll define a helper function to get the predictions from our model:

[42]:
```
def get_predictions(model, data_loader):
  model = model.eval()

  review_texts = []
  predictions = []
  prediction_probs = []
  real_values = []

  with torch.no_grad():
    for d in data_loader:

      texts = d["review_text"]
      input_ids = d["input_ids"].to(device)
      attention_mask = d["attention_mask"].to(device)
      targets = d["targets"].to(device)

      outputs = model(
        input_ids=input_ids,
        attention_mask=attention_mask
      )
      _, preds = torch.max(outputs, dim=1)

      probs = F.softmax(outputs, dim=1)

      review_texts.extend(texts)
      predictions.extend(preds)
      prediction_probs.extend(probs)
      real_values.extend(targets)

  predictions = torch.stack(predictions).cpu()
  prediction_probs = torch.stack(prediction_probs).cpu()
  real_values = torch.stack(real_values).cpu()
  return review_texts, predictions, prediction_probs, real_values
```

This is similar to the evaluation function, except that we're storing the text of the reviews and the predicted probabilities (by applying the softmax on the model outputs):

[43]:
```
y_review_texts, bert_y_pred, y_pred_probs, y_test = get_predictions(
  model,
  test_data_loader
)
```

Let's have a look at the classification report :

[44]:
```
print(classification_report(y_test, bert_y_pred, target_names=class_names))
```

```
              precision    recall  f1-score   support

    negative       0.90      0.92      0.91      1889
     neutral       0.69      0.62      0.66       580
    positive       0.77      0.79      0.78       459

    accuracy                           0.84      2928
   macro avg       0.79      0.78      0.78      2928
weighted avg       0.84      0.84      0.84      2928
```

Looks like it is really hard to classify neutral reviews. Even empirically, looking at many reviews, those are hard to classify.

We also observe very good results on the negative class, which understandable, knowing it to be the major class in our dataset with the most occurences.

Using the `weighted` average that takes into account the imbalances in the dataset and tries to correct them, we observe very good results in precision, recall and f1-score. Even in `macro` averaging, where we don't try to correct imbalances in the dataset, we still keep having good average precision, recall and f1-score.

We'll continue with the confusion matrix:

[45]:
```
def show_confusion_matrix(confusion_matrix):
  hmap = sns.heatmap(confusion_matrix, annot=True, fmt="d", cmap="Blues")
  hmap.yaxis.set_ticklabels(hmap.yaxis.get_ticklabels(), rotation=0, ha='right')
  hmap.xaxis.set_ticklabels(hmap.xaxis.get_ticklabels(), rotation=30,␣
 ↪ha='right')
  plt.ylabel('True sentiment')
  plt.xlabel('Predicted sentiment');

cm = confusion_matrix(y_test, bert_y_pred)
df_cm = pd.DataFrame(cm, index=class_names, columns=class_names)
show_confusion_matrix(df_cm)
```

This confirms that our model is having difficulty classifying neutral reviews. It mistakes those for negative with more frequency than the positive.

That's a good overview of the performance of our model. But let's have a look at an example from our test data:

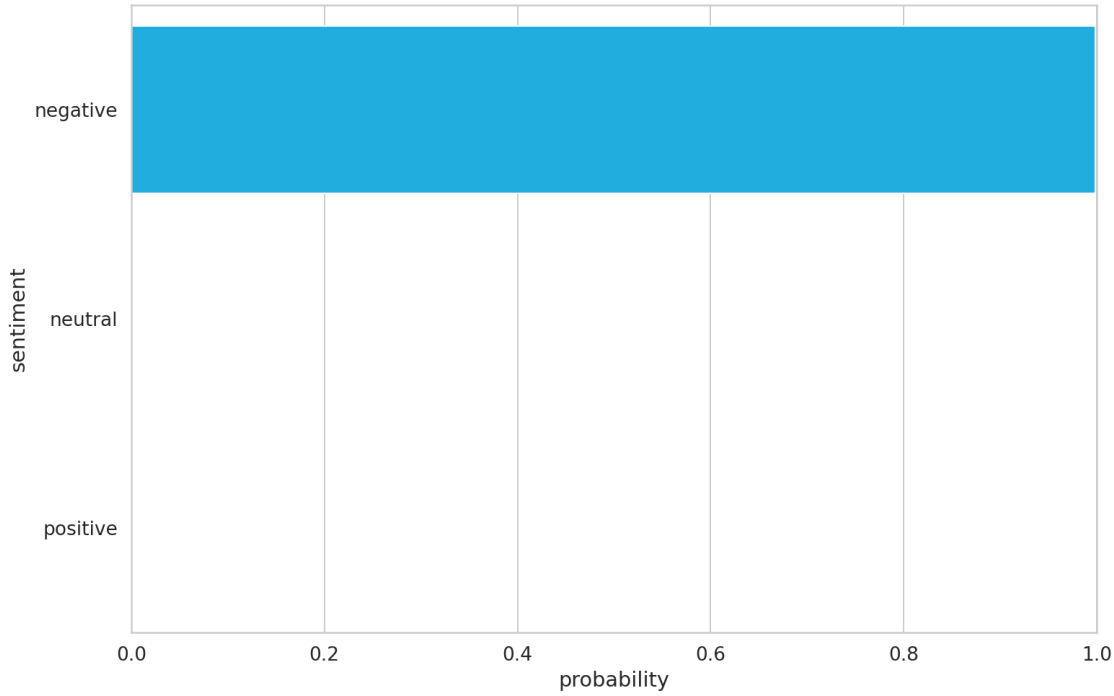
[46]:
```
# This number can be changed to have more data to test qualitatively with
idx = 999

review_text = y_review_texts[idx]
true_sentiment = y_test[idx]
pred_df = pd.DataFrame({
  'class_names': class_names,
  'values': y_pred_probs[idx]
})
```

[47]:
```
print("\n".join(wrap(review_text)))
print()
print(f'True sentiment: {class_names[true_sentiment]}')
```

```
bag in possession but no clothes in bag??

True sentiment: negative
```

Now we can look at the confidence of each sentiment of our model:

[48]:
```
sns.barplot(x='values', y='class_names', data=pred_df, orient='h')
plt.ylabel('sentiment')
plt.xlabel('probability')
plt.xlim([0, 1]);
```



2 - LSTM recurrent model We would also be testing another deep learning model, adapted for sentiment analysis and classification. We would be using Long Short Term Memoriy neurons with an embedding layer.

We begin by getting the features (text) and labels to prepare and preprocess them :

[49]:
```
# Getting the airplane reviews as labels
train_features_words = df_train['review']
val_features_words = df_val['review']
test_features_words = df_test['review']

features = tweets['review']

# Get dummies values of the three classes, meaning one 3 dimensional vector of␣
 ↪interger values for each line
train_labels = pd.get_dummies(df_train['airline_sentiment']).values
val_labels = pd.get_dummies(df_val['airline_sentiment']).values
test_labels = pd.get_dummies(df_test['airline_sentiment']).values
```

We then choose the max number of words in the text vocabulary we got, that we would consider as relevant. We would then fit a tokenizer on our data, and complete it with a padding of zero values :

[50]:
```
# Max features determine the max number of words in the entire dataset you will␣
 ↪consider relevant
max_features = 500

# Create a tokenizer object and pass the max features to it
tokenizer = Tokenizer(num_words=max_features, split=' ')

# Call the tokenizer method fit_on_texts and fit the values of your features to␣
 ↪it
tokenizer.fit_on_texts(features.values)
```

[51]:
```
# Convert each line of text to sequences of integer values where each value␣
 ↪correspond to one of the 100 max features
features_train = tokenizer.texts_to_sequences(train_features_words.values)
features_val = tokenizer.texts_to_sequences(val_features_words.values)
features_test = tokenizer.texts_to_sequences(test_features_words.values)

# Pad sequences is needed to homogenize the size of each line vector (if 100␣
 ↪max_features, then each line vector will be converted to 25dim vectors,␣
 ↪blank or no words
# will be filled with zero value
features_train = pad_sequences(features_train)
features_val = pad_sequences(features_val)
features_test = pad_sequences(features_test)
```

We'll transform our data into dataframes, for ease of manipulation, and resolve a padding issue in the test features :

[52]:
```
features_train = pd.DataFrame(features_train)
features_val = pd.DataFrame(features_val)
features_test = pd.DataFrame(features_test)



label_train = pd.DataFrame(train_labels)
label_val = pd.DataFrame(val_labels)
label_test = pd.DataFrame(test_labels)
```

[53]:
```
if features_test.shape[1] < features_train.shape[1] :
  temp = pd.DataFrame({0:[0]*features_test.shape[0]})
  for i in range(features_test.shape[1]):
    temp[i+1] = features_test[i]
  features_test = temp.copy()
```

The input size of our model would be our number of features, 29 in this case.

[54]:
```
input_len = features_train.shape[1]
```

[55]:
```
# Embedding layer size
embeding_dimension = 50

# LSTM layer size
lstm_size = 70
```

Our next step would be to define our model. We'll be using an embedding layer, with a 20% dropout to avoid overfitting, added to an LSTM. This would be completed by a dense layer with softmax activation, for the classification.

[56]:
```
from keras import activations
# We create the model graph that would return the model instance
def lstm_model(input_shape):

    X_input = Input(shape = input_shape)
    X = Embedding(max_features,embeding_dimension,␣
 ↪input_length=input_len)(X_input)
    X = SpatialDropout1D(0.2)(X)
    X = LSTM(lstm_size, dropout=0.2)(X)  #, recurrent_dropout=0.2

    X = Dense(3,activation='softmax')(X)

    model = Model(inputs = X_input, outputs = X)

    return model
```

[57]:
```
lstm_model = lstm_model(input_shape = input_len)
lstm_model.summary()
```

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 29)]              0
_________________________________________________________________
embedding (Embedding)        (None, 29, 50)            25000
_________________________________________________________________
spatial_dropout1d (SpatialDr (None, 29, 50)            0
_________________________________________________________________
lstm (LSTM)                  (None, 70)                33880
_________________________________________________________________
dense (Dense)                (None, 3)                 213
=================================================================
Total params: 59,093
Trainable params: 59,093
Non-trainable params: 0
```

We'll be working with the Adam optimizer, with a very small learning rate, as well as a categorical entropy.

[58]:
```
opt = Adam(lr=0.00001)

lstm_model.compile(loss='categorical_crossentropy', optimizer=opt,␣
↪metrics=["accuracy"])
```

Following that, we'll define a checkpoint where we save the best model with the best validation accuracy, to be able to keep the best model for usage after training.

[59]:
```
filepath='best_accuracy.h5'

checkpoint =␣
↪ModelCheckpoint(filepath,monitor='val_accuracy',mode='max',save_best_only=True,verbose=1)
callbacks_list = [checkpoint]
```

After some hyperparameter tuning, we choose to work with a batch size of 16 entries. We then train the model for 150 epoch using our training and validation data :

[62]:
```
history = lstm_model.fit(features_train, label_train, batch_size = 16,␣
↪epochs=150,validation_data=(features_val, label_val),callbacks=checkpoint)
```

```
Epoch 1/150
654/659 [============================>.] - ETA: 0s - loss: 1.0780 - accuracy:
0.5832
Epoch 00001: val_accuracy improved from -inf to 0.62287, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 6ms/step - loss: 1.0776 -
accuracy: 0.5836 - val_loss: 1.0386 - val_accuracy: 0.6229
Epoch 2/150
659/659 [==============================] - ETA: 0s - loss: 0.9588 - accuracy:
0.6223
Epoch 00002: val_accuracy did not improve from 0.62287
659/659 [==============================] - 3s 5ms/step - loss: 0.9588 -
accuracy: 0.6223 - val_loss: 0.9226 - val_accuracy: 0.6229
Epoch 3/150
656/659 [============================>.] - ETA: 0s - loss: 0.9040 - accuracy:
0.6223
Epoch 00003: val_accuracy did not improve from 0.62287
659/659 [==============================] - 3s 5ms/step - loss: 0.9040 -
accuracy: 0.6223 - val_loss: 0.8800 - val_accuracy: 0.6229
Epoch 4/150
654/659 [============================>.] - ETA: 0s - loss: 0.8652 - accuracy:
0.6215
Epoch 00004: val_accuracy did not improve from 0.62287
659/659 [==============================] - 3s 5ms/step - loss: 0.8643 -
```

```
accuracy: 0.6223 - val_loss: 0.8420 - val_accuracy: 0.6229
Epoch 5/150
656/659 [============================>.] - ETA: 0s - loss: 0.8418 - accuracy:
0.6238
Epoch 00005: val_accuracy did not improve from 0.62287
659/659 [==============================] - 3s 5ms/step - loss: 0.8425 -
accuracy: 0.6234 - val_loss: 0.8289 - val_accuracy: 0.6229
Epoch 6/150
658/659 [============================>.] - ETA: 0s - loss: 0.8312 - accuracy:
0.6243
Epoch 00006: val_accuracy did not improve from 0.62287
659/659 [==============================] - 3s 5ms/step - loss: 0.8316 -
accuracy: 0.6240 - val_loss: 0.8220 - val_accuracy: 0.6229
Epoch 7/150
651/659 [============================>.] - ETA: 0s - loss: 0.8273 - accuracy:
0.6264
Epoch 00007: val_accuracy did not improve from 0.62287
659/659 [==============================] - 3s 5ms/step - loss: 0.8260 -
accuracy: 0.6270 - val_loss: 0.8165 - val_accuracy: 0.6229
Epoch 8/150
653/659 [============================>.] - ETA: 0s - loss: 0.8201 - accuracy:
0.6274
Epoch 00008: val_accuracy did not improve from 0.62287
659/659 [==============================] - 3s 5ms/step - loss: 0.8195 -
accuracy: 0.6278 - val_loss: 0.8117 - val_accuracy: 0.6229
Epoch 9/150
649/659 [============================>.] - ETA: 0s - loss: 0.8138 - accuracy:
0.6283
Epoch 00009: val_accuracy improved from 0.62287 to 0.62799, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.8121 -
accuracy: 0.6292 - val_loss: 0.8072 - val_accuracy: 0.6280
Epoch 10/150
656/659 [============================>.] - ETA: 0s - loss: 0.8059 - accuracy:
0.6313
Epoch 00010: val_accuracy did not improve from 0.62799
659/659 [==============================] - 3s 5ms/step - loss: 0.8062 -
accuracy: 0.6313 - val_loss: 0.8020 - val_accuracy: 0.6271
Epoch 11/150
651/659 [============================>.] - ETA: 0s - loss: 0.8012 - accuracy:
0.6331
Epoch 00011: val_accuracy did not improve from 0.62799
659/659 [==============================] - 3s 5ms/step - loss: 0.8012 -
accuracy: 0.6332 - val_loss: 0.7962 - val_accuracy: 0.6280
Epoch 12/150
654/659 [============================>.] - ETA: 0s - loss: 0.7938 - accuracy:
0.6335
Epoch 00012: val_accuracy improved from 0.62799 to 0.63140, saving model to
```

```
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.7935 -
accuracy: 0.6338 - val_loss: 0.7908 - val_accuracy: 0.6314
Epoch 13/150
656/659 [============================>.] - ETA: 0s - loss: 0.7877 - accuracy:
0.6343
Epoch 00013: val_accuracy improved from 0.63140 to 0.63567, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.7875 -
accuracy: 0.6343 - val_loss: 0.7856 - val_accuracy: 0.6357
Epoch 14/150
657/659 [============================>.] - ETA: 0s - loss: 0.7783 - accuracy:
0.6383
Epoch 00014: val_accuracy improved from 0.63567 to 0.64505, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.7781 -
accuracy: 0.6385 - val_loss: 0.7800 - val_accuracy: 0.6451
Epoch 15/150
653/659 [============================>.] - ETA: 0s - loss: 0.7743 - accuracy:
0.6425
Epoch 00015: val_accuracy did not improve from 0.64505
659/659 [==============================] - 4s 5ms/step - loss: 0.7744 -
accuracy: 0.6425 - val_loss: 0.7739 - val_accuracy: 0.6442
Epoch 16/150
657/659 [============================>.] - ETA: 0s - loss: 0.7669 - accuracy:
0.6469
Epoch 00016: val_accuracy improved from 0.64505 to 0.65017, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.7665 -
accuracy: 0.6472 - val_loss: 0.7683 - val_accuracy: 0.6502
Epoch 17/150
657/659 [============================>.] - ETA: 0s - loss: 0.7593 - accuracy:
0.6475
Epoch 00017: val_accuracy improved from 0.65017 to 0.66297, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.7591 -
accuracy: 0.6475 - val_loss: 0.7630 - val_accuracy: 0.6630
Epoch 18/150
654/659 [============================>.] - ETA: 0s - loss: 0.7515 - accuracy:
0.6508
Epoch 00018: val_accuracy improved from 0.66297 to 0.66382, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.7516 -
accuracy: 0.6508 - val_loss: 0.7572 - val_accuracy: 0.6638
Epoch 19/150
659/659 [==============================] - ETA: 0s - loss: 0.7456 - accuracy:
0.6550
Epoch 00019: val_accuracy did not improve from 0.66382
```

```
659/659 [==============================] - 3s 5ms/step - loss: 0.7456 -
accuracy: 0.6550 - val_loss: 0.7511 - val_accuracy: 0.6630
Epoch 20/150
657/659 [============================>.] - ETA: 0s - loss: 0.7383 - accuracy:
0.6550
Epoch 00020: val_accuracy improved from 0.66382 to 0.67150, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.7381 -
accuracy: 0.6551 - val_loss: 0.7460 - val_accuracy: 0.6715
Epoch 21/150
659/659 [==============================] - ETA: 0s - loss: 0.7307 - accuracy:
0.6592
Epoch 00021: val_accuracy did not improve from 0.67150
659/659 [==============================] - 3s 5ms/step - loss: 0.7307 -
accuracy: 0.6592 - val_loss: 0.7404 - val_accuracy: 0.6715
Epoch 22/150
656/659 [============================>.] - ETA: 0s - loss: 0.7254 - accuracy:
0.6641
Epoch 00022: val_accuracy improved from 0.67150 to 0.68089, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.7252 -
accuracy: 0.6642 - val_loss: 0.7354 - val_accuracy: 0.6809
Epoch 23/150
653/659 [============================>.] - ETA: 0s - loss: 0.7183 - accuracy:
0.6673
Epoch 00023: val_accuracy did not improve from 0.68089
659/659 [==============================] - 4s 5ms/step - loss: 0.7193 -
accuracy: 0.6667 - val_loss: 0.7301 - val_accuracy: 0.6792
Epoch 24/150
654/659 [============================>.] - ETA: 0s - loss: 0.7133 - accuracy:
0.6678
Epoch 00024: val_accuracy improved from 0.68089 to 0.68430, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.7128 -
accuracy: 0.6682 - val_loss: 0.7254 - val_accuracy: 0.6843
Epoch 25/150
659/659 [==============================] - ETA: 0s - loss: 0.7067 - accuracy:
0.6731
Epoch 00025: val_accuracy improved from 0.68430 to 0.68686, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.7067 -
accuracy: 0.6731 - val_loss: 0.7211 - val_accuracy: 0.6869
Epoch 26/150
653/659 [============================>.] - ETA: 0s - loss: 0.7023 - accuracy:
0.6740
Epoch 00026: val_accuracy improved from 0.68686 to 0.68857, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.7021 -
```

```
accuracy: 0.6743 - val_loss: 0.7150 - val_accuracy: 0.6886
Epoch 27/150
659/659 [==============================] - ETA: 0s - loss: 0.6947 - accuracy:
0.6788
Epoch 00027: val_accuracy improved from 0.68857 to 0.68942, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.6947 -
accuracy: 0.6788 - val_loss: 0.7128 - val_accuracy: 0.6894
Epoch 28/150
655/659 [============================>.] - ETA: 0s - loss: 0.6925 - accuracy:
0.6813
Epoch 00028: val_accuracy improved from 0.68942 to 0.69454, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.6919 -
accuracy: 0.6816 - val_loss: 0.7070 - val_accuracy: 0.6945
Epoch 29/150
652/659 [============================>.] - ETA: 0s - loss: 0.6839 - accuracy:
0.6850
Epoch 00029: val_accuracy did not improve from 0.69454
659/659 [==============================] - 3s 5ms/step - loss: 0.6837 -
accuracy: 0.6850 - val_loss: 0.7036 - val_accuracy: 0.6937
Epoch 30/150
653/659 [============================>.] - ETA: 0s - loss: 0.6798 - accuracy:
0.6886
Epoch 00030: val_accuracy did not improve from 0.69454
659/659 [==============================] - 3s 5ms/step - loss: 0.6799 -
accuracy: 0.6882 - val_loss: 0.7004 - val_accuracy: 0.6945
Epoch 31/150
651/659 [============================>.] - ETA: 0s - loss: 0.6756 - accuracy:
0.6911
Epoch 00031: val_accuracy improved from 0.69454 to 0.69625, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.6766 -
accuracy: 0.6909 - val_loss: 0.6961 - val_accuracy: 0.6962
Epoch 32/150
657/659 [============================>.] - ETA: 0s - loss: 0.6724 - accuracy:
0.6919
Epoch 00032: val_accuracy did not improve from 0.69625
659/659 [==============================] - 3s 5ms/step - loss: 0.6721 -
accuracy: 0.6922 - val_loss: 0.6929 - val_accuracy: 0.6962
Epoch 33/150
658/659 [============================>.] - ETA: 0s - loss: 0.6696 - accuracy:
0.6968
Epoch 00033: val_accuracy improved from 0.69625 to 0.69966, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.6695 -
accuracy: 0.6971 - val_loss: 0.6892 - val_accuracy: 0.6997
Epoch 34/150
```

```
653/659 [============================>.] - ETA: 0s - loss: 0.6634 - accuracy:
0.6984
Epoch 00034: val_accuracy did not improve from 0.69966
659/659 [==============================] - 3s 5ms/step - loss: 0.6636 -
accuracy: 0.6987 - val_loss: 0.6859 - val_accuracy: 0.6997
Epoch 35/150
656/659 [============================>.] - ETA: 0s - loss: 0.6625 - accuracy:
0.7047
Epoch 00035: val_accuracy improved from 0.69966 to 0.70137, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.6619 -
accuracy: 0.7049 - val_loss: 0.6828 - val_accuracy: 0.7014
Epoch 36/150
655/659 [============================>.] - ETA: 0s - loss: 0.6588 - accuracy:
0.7083
Epoch 00036: val_accuracy improved from 0.70137 to 0.70222, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.6585 -
accuracy: 0.7083 - val_loss: 0.6806 - val_accuracy: 0.7022
Epoch 37/150
654/659 [============================>.] - ETA: 0s - loss: 0.6544 - accuracy:
0.7102
Epoch 00037: val_accuracy improved from 0.70222 to 0.70990, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.6554 -
accuracy: 0.7099 - val_loss: 0.6765 - val_accuracy: 0.7099
Epoch 38/150
657/659 [============================>.] - ETA: 0s - loss: 0.6517 - accuracy:
0.7149
Epoch 00038: val_accuracy improved from 0.70990 to 0.71331, saving model to
best_accuracy.h5
659/659 [==============================] - 3s 5ms/step - loss: 0.6520 -
accuracy: 0.7148 - val_loss: 0.6744 - val_accuracy: 0.7133
Epoch 39/150
658/659 [============================>.] - ETA: 0s - loss: 0.6491 - accuracy:
0.7173
Epoch 00039: val_accuracy did not improve from 0.71331
659/659 [==============================] - 3s 5ms/step - loss: 0.6492 -
accuracy: 0.7174 - val_loss: 0.6710 - val_accuracy: 0.7125
Epoch 40/150
653/659 [============================>.] - ETA: 0s - loss: 0.6438 - accuracy:
0.7210
Epoch 00040: val_accuracy did not improve from 0.71331
659/659 [==============================] - 3s 5ms/step - loss: 0.6436 -
accuracy: 0.7206 - val_loss: 0.6700 - val_accuracy: 0.7116
Epoch 41/150
658/659 [============================>.] - ETA: 0s - loss: 0.6420 - accuracy:
0.7201
```

```
Epoch 00041: val_accuracy improved from 0.71331 to 0.72526, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.6420 -
accuracy: 0.7201 - val_loss: 0.6671 - val_accuracy: 0.7253
Epoch 42/150
655/659 [============================>.] - ETA: 0s - loss: 0.6409 - accuracy:
0.7256
Epoch 00042: val_accuracy improved from 0.72526 to 0.72611, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.6415 -
accuracy: 0.7256 - val_loss: 0.6654 - val_accuracy: 0.7261
Epoch 43/150
656/659 [============================>.] - ETA: 0s - loss: 0.6388 - accuracy:
0.7291
Epoch 00043: val_accuracy did not improve from 0.72611
659/659 [==============================] - 4s 5ms/step - loss: 0.6388 -
accuracy: 0.7292 - val_loss: 0.6653 - val_accuracy: 0.7201
Epoch 44/150
649/659 [============================>.] - ETA: 0s - loss: 0.6332 - accuracy:
0.7305
Epoch 00044: val_accuracy improved from 0.72611 to 0.73123, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.6352 -
accuracy: 0.7296 - val_loss: 0.6615 - val_accuracy: 0.7312
Epoch 45/150
655/659 [============================>.] - ETA: 0s - loss: 0.6341 - accuracy:
0.7322
Epoch 00045: val_accuracy did not improve from 0.73123
659/659 [==============================] - 4s 5ms/step - loss: 0.6343 -
accuracy: 0.7325 - val_loss: 0.6591 - val_accuracy: 0.7304
Epoch 46/150
654/659 [============================>.] - ETA: 0s - loss: 0.6314 - accuracy:
0.7381
Epoch 00046: val_accuracy did not improve from 0.73123
659/659 [==============================] - 4s 5ms/step - loss: 0.6317 -
accuracy: 0.7380 - val_loss: 0.6582 - val_accuracy: 0.7312
Epoch 47/150
659/659 [==============================] - ETA: 0s - loss: 0.6305 - accuracy:
0.7426
Epoch 00047: val_accuracy improved from 0.73123 to 0.73464, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.6305 -
accuracy: 0.7426 - val_loss: 0.6556 - val_accuracy: 0.7346
Epoch 48/150
656/659 [============================>.] - ETA: 0s - loss: 0.6273 - accuracy:
0.7426
Epoch 00048: val_accuracy did not improve from 0.73464
659/659 [==============================] - 3s 5ms/step - loss: 0.6266 -
```

```
accuracy: 0.7429 - val_loss: 0.6548 - val_accuracy: 0.7346
Epoch 49/150
657/659 [============================>.] - ETA: 0s - loss: 0.6217 - accuracy:
0.7451
Epoch 00049: val_accuracy did not improve from 0.73464
659/659 [==============================] - 3s 5ms/step - loss: 0.6220 -
accuracy: 0.7450 - val_loss: 0.6533 - val_accuracy: 0.7329
Epoch 50/150
653/659 [============================>.] - ETA: 0s - loss: 0.6217 - accuracy:
0.7442
Epoch 00050: val_accuracy improved from 0.73464 to 0.73805, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.6219 -
accuracy: 0.7443 - val_loss: 0.6504 - val_accuracy: 0.7381
Epoch 51/150
651/659 [============================>.] - ETA: 0s - loss: 0.6209 - accuracy:
0.7498
Epoch 00051: val_accuracy did not improve from 0.73805
659/659 [==============================] - 4s 5ms/step - loss: 0.6211 -
accuracy: 0.7497 - val_loss: 0.6505 - val_accuracy: 0.7346
Epoch 52/150
659/659 [==============================] - ETA: 0s - loss: 0.6163 - accuracy:
0.7491
Epoch 00052: val_accuracy improved from 0.73805 to 0.74147, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 6ms/step - loss: 0.6163 -
accuracy: 0.7491 - val_loss: 0.6485 - val_accuracy: 0.7415
Epoch 53/150
654/659 [============================>.] - ETA: 0s - loss: 0.6178 - accuracy:
0.7493
Epoch 00053: val_accuracy did not improve from 0.74147
659/659 [==============================] - 4s 6ms/step - loss: 0.6175 -
accuracy: 0.7491 - val_loss: 0.6466 - val_accuracy: 0.7381
Epoch 54/150
653/659 [============================>.] - ETA: 0s - loss: 0.6153 - accuracy:
0.7518
Epoch 00054: val_accuracy did not improve from 0.74147
659/659 [==============================] - 4s 6ms/step - loss: 0.6145 -
accuracy: 0.7516 - val_loss: 0.6453 - val_accuracy: 0.7398
Epoch 55/150
652/659 [============================>.] - ETA: 0s - loss: 0.6104 - accuracy:
0.7546
Epoch 00055: val_accuracy improved from 0.74147 to 0.74403, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 6ms/step - loss: 0.6101 -
accuracy: 0.7543 - val_loss: 0.6441 - val_accuracy: 0.7440
Epoch 56/150
651/659 [============================>.] - ETA: 0s - loss: 0.6067 - accuracy:
```

```
0.7536
Epoch 00056: val_accuracy did not improve from 0.74403
659/659 [==============================] - 4s 5ms/step - loss: 0.6081 -
accuracy: 0.7529 - val_loss: 0.6437 - val_accuracy: 0.7415
Epoch 57/150
656/659 [============================>.] - ETA: 0s - loss: 0.6096 - accuracy:
0.7544
Epoch 00057: val_accuracy improved from 0.74403 to 0.74744, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.6094 -
accuracy: 0.7545 - val_loss: 0.6419 - val_accuracy: 0.7474
Epoch 58/150
652/659 [============================>.] - ETA: 0s - loss: 0.6090 - accuracy:
0.7524
Epoch 00058: val_accuracy improved from 0.74744 to 0.74915, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.6087 -
accuracy: 0.7525 - val_loss: 0.6405 - val_accuracy: 0.7491
Epoch 59/150
657/659 [============================>.] - ETA: 0s - loss: 0.6020 - accuracy:
0.7560
Epoch 00059: val_accuracy did not improve from 0.74915
659/659 [==============================] - 3s 5ms/step - loss: 0.6023 -
accuracy: 0.7559 - val_loss: 0.6404 - val_accuracy: 0.7466
Epoch 60/150
658/659 [============================>.] - ETA: 0s - loss: 0.6008 - accuracy:
0.7549
Epoch 00060: val_accuracy improved from 0.74915 to 0.75085, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.6007 -
accuracy: 0.7549 - val_loss: 0.6379 - val_accuracy: 0.7509
Epoch 61/150
650/659 [============================>.] - ETA: 0s - loss: 0.6016 - accuracy:
0.7571
Epoch 00061: val_accuracy improved from 0.75085 to 0.75256, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.6005 -
accuracy: 0.7574 - val_loss: 0.6368 - val_accuracy: 0.7526
Epoch 62/150
650/659 [============================>.] - ETA: 0s - loss: 0.5975 - accuracy:
0.7583
Epoch 00062: val_accuracy improved from 0.75256 to 0.75341, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.5979 -
accuracy: 0.7579 - val_loss: 0.6356 - val_accuracy: 0.7534
Epoch 63/150
658/659 [============================>.] - ETA: 0s - loss: 0.5961 - accuracy:
0.7581
```

```
Epoch 00063: val_accuracy did not improve from 0.75341
659/659 [==============================] - 3s 5ms/step - loss: 0.5960 -
accuracy: 0.7582 - val_loss: 0.6337 - val_accuracy: 0.7509
Epoch 64/150
651/659 [============================>.] - ETA: 0s - loss: 0.5956 - accuracy:
0.7608
Epoch 00064: val_accuracy did not improve from 0.75341
659/659 [==============================] - 4s 5ms/step - loss: 0.5956 -
accuracy: 0.7607 - val_loss: 0.6328 - val_accuracy: 0.7526
Epoch 65/150
655/659 [============================>.] - ETA: 0s - loss: 0.5952 - accuracy:
0.7589
Epoch 00065: val_accuracy did not improve from 0.75341
659/659 [==============================] - 3s 5ms/step - loss: 0.5948 -
accuracy: 0.7588 - val_loss: 0.6319 - val_accuracy: 0.7509
Epoch 66/150
653/659 [============================>.] - ETA: 0s - loss: 0.5905 - accuracy:
0.7629
Epoch 00066: val_accuracy did not improve from 0.75341
659/659 [==============================] - 4s 5ms/step - loss: 0.5898 -
accuracy: 0.7636 - val_loss: 0.6309 - val_accuracy: 0.7517
Epoch 67/150
653/659 [============================>.] - ETA: 0s - loss: 0.5906 - accuracy:
0.7602
Epoch 00067: val_accuracy did not improve from 0.75341
659/659 [==============================] - 4s 5ms/step - loss: 0.5895 -
accuracy: 0.7607 - val_loss: 0.6323 - val_accuracy: 0.7500
Epoch 68/150
658/659 [============================>.] - ETA: 0s - loss: 0.5901 - accuracy:
0.7615
Epoch 00068: val_accuracy did not improve from 0.75341
659/659 [==============================] - 4s 6ms/step - loss: 0.5900 -
accuracy: 0.7615 - val_loss: 0.6293 - val_accuracy: 0.7526
Epoch 69/150
653/659 [============================>.] - ETA: 0s - loss: 0.5864 - accuracy:
0.7660
Epoch 00069: val_accuracy did not improve from 0.75341
659/659 [==============================] - 4s 6ms/step - loss: 0.5869 -
accuracy: 0.7657 - val_loss: 0.6266 - val_accuracy: 0.7491
Epoch 70/150
659/659 [==============================] - ETA: 0s - loss: 0.5837 - accuracy:
0.7646
Epoch 00070: val_accuracy improved from 0.75341 to 0.75427, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.5837 -
accuracy: 0.7646 - val_loss: 0.6261 - val_accuracy: 0.7543
Epoch 71/150
654/659 [============================>.] - ETA: 0s - loss: 0.5832 - accuracy:
```

```
0.7647
Epoch 00071: val_accuracy did not improve from 0.75427
659/659 [==============================] - 4s 5ms/step - loss: 0.5833 -
accuracy: 0.7646 - val_loss: 0.6254 - val_accuracy: 0.7526
Epoch 72/150
654/659 [============================>.] - ETA: 0s - loss: 0.5818 - accuracy:
0.7642
Epoch 00072: val_accuracy did not improve from 0.75427
659/659 [==============================] - 3s 5ms/step - loss: 0.5815 -
accuracy: 0.7642 - val_loss: 0.6234 - val_accuracy: 0.7534
Epoch 73/150
651/659 [============================>.] - ETA: 0s - loss: 0.5782 - accuracy:
0.7682
Epoch 00073: val_accuracy did not improve from 0.75427
659/659 [==============================] - 4s 5ms/step - loss: 0.5776 -
accuracy: 0.7685 - val_loss: 0.6232 - val_accuracy: 0.7534
Epoch 74/150
650/659 [============================>.] - ETA: 0s - loss: 0.5775 - accuracy:
0.7654
Epoch 00074: val_accuracy did not improve from 0.75427
659/659 [==============================] - 4s 5ms/step - loss: 0.5775 -
accuracy: 0.7654 - val_loss: 0.6230 - val_accuracy: 0.7526
Epoch 75/150
651/659 [============================>.] - ETA: 0s - loss: 0.5763 - accuracy:
0.7650
Epoch 00075: val_accuracy did not improve from 0.75427
659/659 [==============================] - 4s 5ms/step - loss: 0.5757 -
accuracy: 0.7653 - val_loss: 0.6212 - val_accuracy: 0.7534
Epoch 76/150
655/659 [============================>.] - ETA: 0s - loss: 0.5720 - accuracy:
0.7720
Epoch 00076: val_accuracy improved from 0.75427 to 0.75768, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.5729 -
accuracy: 0.7717 - val_loss: 0.6204 - val_accuracy: 0.7577
Epoch 77/150
650/659 [============================>.] - ETA: 0s - loss: 0.5753 - accuracy:
0.7680
Epoch 00077: val_accuracy did not improve from 0.75768
659/659 [==============================] - 4s 5ms/step - loss: 0.5754 -
accuracy: 0.7679 - val_loss: 0.6180 - val_accuracy: 0.7534
Epoch 78/150
654/659 [============================>.] - ETA: 0s - loss: 0.5744 - accuracy:
0.7656
Epoch 00078: val_accuracy improved from 0.75768 to 0.76195, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 6ms/step - loss: 0.5744 -
accuracy: 0.7660 - val_loss: 0.6177 - val_accuracy: 0.7619
```

```
Epoch 79/150
658/659 [============================>.] - ETA: 0s - loss: 0.5728 - accuracy:
0.7703
Epoch 00079: val_accuracy did not improve from 0.76195
659/659 [==============================] - 4s 5ms/step - loss: 0.5727 -
accuracy: 0.7704 - val_loss: 0.6161 - val_accuracy: 0.7611
Epoch 80/150
650/659 [============================>.] - ETA: 0s - loss: 0.5714 - accuracy:
0.7702
Epoch 00080: val_accuracy improved from 0.76195 to 0.76365, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.5717 -
accuracy: 0.7699 - val_loss: 0.6154 - val_accuracy: 0.7637
Epoch 81/150
657/659 [============================>.] - ETA: 0s - loss: 0.5694 - accuracy:
0.7693
Epoch 00081: val_accuracy did not improve from 0.76365
659/659 [==============================] - 4s 5ms/step - loss: 0.5696 -
accuracy: 0.7692 - val_loss: 0.6150 - val_accuracy: 0.7611
Epoch 82/150
652/659 [============================>.] - ETA: 0s - loss: 0.5674 - accuracy:
0.7700
Epoch 00082: val_accuracy did not improve from 0.76365
659/659 [==============================] - 4s 5ms/step - loss: 0.5672 -
accuracy: 0.7697 - val_loss: 0.6143 - val_accuracy: 0.7594
Epoch 83/150
653/659 [============================>.] - ETA: 0s - loss: 0.5685 - accuracy:
0.7697
Epoch 00083: val_accuracy improved from 0.76365 to 0.76706, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.5681 -
accuracy: 0.7699 - val_loss: 0.6135 - val_accuracy: 0.7671
Epoch 84/150
654/659 [============================>.] - ETA: 0s - loss: 0.5659 - accuracy:
0.7688
Epoch 00084: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 5ms/step - loss: 0.5654 -
accuracy: 0.7689 - val_loss: 0.6145 - val_accuracy: 0.7534
Epoch 85/150
651/659 [============================>.] - ETA: 0s - loss: 0.5624 - accuracy:
0.7721
Epoch 00085: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 5ms/step - loss: 0.5620 -
accuracy: 0.7721 - val_loss: 0.6113 - val_accuracy: 0.7619
Epoch 86/150
657/659 [============================>.] - ETA: 0s - loss: 0.5648 - accuracy:
0.7724
Epoch 00086: val_accuracy did not improve from 0.76706
```

```
659/659 [==============================] - 4s 5ms/step - loss: 0.5644 -
accuracy: 0.7725 - val_loss: 0.6106 - val_accuracy: 0.7619
Epoch 87/150
658/659 [============================>.] - ETA: 0s - loss: 0.5636 - accuracy:
0.7733
Epoch 00087: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 5ms/step - loss: 0.5636 -
accuracy: 0.7731 - val_loss: 0.6100 - val_accuracy: 0.7637
Epoch 88/150
655/659 [============================>.] - ETA: 0s - loss: 0.5623 - accuracy:
0.7708
Epoch 00088: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 5ms/step - loss: 0.5622 -
accuracy: 0.7709 - val_loss: 0.6086 - val_accuracy: 0.7662
Epoch 89/150
650/659 [============================>.] - ETA: 0s - loss: 0.5590 - accuracy:
0.7773
Epoch 00089: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 5ms/step - loss: 0.5595 -
accuracy: 0.7769 - val_loss: 0.6091 - val_accuracy: 0.7611
Epoch 90/150
655/659 [============================>.] - ETA: 0s - loss: 0.5563 - accuracy:
0.7777
Epoch 00090: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 6ms/step - loss: 0.5560 -
accuracy: 0.7779 - val_loss: 0.6093 - val_accuracy: 0.7602
Epoch 91/150
654/659 [============================>.] - ETA: 0s - loss: 0.5600 - accuracy:
0.7745
Epoch 00091: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 6ms/step - loss: 0.5601 -
accuracy: 0.7744 - val_loss: 0.6057 - val_accuracy: 0.7637
Epoch 92/150
654/659 [============================>.] - ETA: 0s - loss: 0.5536 - accuracy:
0.7773
Epoch 00092: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 6ms/step - loss: 0.5544 -
accuracy: 0.7773 - val_loss: 0.6048 - val_accuracy: 0.7645
Epoch 93/150
656/659 [============================>.] - ETA: 0s - loss: 0.5555 - accuracy:
0.7760
Epoch 00093: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 5ms/step - loss: 0.5558 -
accuracy: 0.7759 - val_loss: 0.6049 - val_accuracy: 0.7628
Epoch 94/150
649/659 [============================>.] - ETA: 0s - loss: 0.5509 - accuracy:
0.7795
Epoch 00094: val_accuracy did not improve from 0.76706
```

```
659/659 [==============================] - 4s 5ms/step - loss: 0.5519 -
accuracy: 0.7788 - val_loss: 0.6045 - val_accuracy: 0.7628
Epoch 95/150
653/659 [============================>.] - ETA: 0s - loss: 0.5547 - accuracy:
0.7779
Epoch 00095: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 5ms/step - loss: 0.5538 -
accuracy: 0.7782 - val_loss: 0.6028 - val_accuracy: 0.7645
Epoch 96/150
652/659 [============================>.] - ETA: 0s - loss: 0.5546 - accuracy:
0.7766
Epoch 00096: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 5ms/step - loss: 0.5540 -
accuracy: 0.7767 - val_loss: 0.6005 - val_accuracy: 0.7662
Epoch 97/150
659/659 [==============================] - ETA: 0s - loss: 0.5498 - accuracy:
0.7771
Epoch 00097: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 5ms/step - loss: 0.5498 -
accuracy: 0.7771 - val_loss: 0.5999 - val_accuracy: 0.7654
Epoch 98/150
653/659 [============================>.] - ETA: 0s - loss: 0.5483 - accuracy:
0.7788
Epoch 00098: val_accuracy did not improve from 0.76706
659/659 [==============================] - 4s 5ms/step - loss: 0.5482 -
accuracy: 0.7788 - val_loss: 0.5987 - val_accuracy: 0.7662
Epoch 99/150
653/659 [============================>.] - ETA: 0s - loss: 0.5490 - accuracy:
0.7781
Epoch 00099: val_accuracy improved from 0.76706 to 0.76792, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 5ms/step - loss: 0.5490 -
accuracy: 0.7782 - val_loss: 0.5970 - val_accuracy: 0.7679
Epoch 100/150
655/659 [============================>.] - ETA: 0s - loss: 0.5466 - accuracy:
0.7806
Epoch 00100: val_accuracy did not improve from 0.76792
659/659 [==============================] - 4s 5ms/step - loss: 0.5467 -
accuracy: 0.7804 - val_loss: 0.5997 - val_accuracy: 0.7637
Epoch 101/150
649/659 [============================>.] - ETA: 0s - loss: 0.5439 - accuracy:
0.7834
Epoch 00101: val_accuracy did not improve from 0.76792
659/659 [==============================] - 4s 5ms/step - loss: 0.5433 -
accuracy: 0.7832 - val_loss: 0.5957 - val_accuracy: 0.7662
Epoch 102/150
656/659 [============================>.] - ETA: 0s - loss: 0.5450 - accuracy:
0.7813
```

```
Epoch 00102: val_accuracy did not improve from 0.76792
659/659 [==============================] - 4s 5ms/step - loss: 0.5446 -
accuracy: 0.7814 - val_loss: 0.5947 - val_accuracy: 0.7662
Epoch 103/150
656/659 [============================>.] - ETA: 0s - loss: 0.5431 - accuracy:
0.7827
Epoch 00103: val_accuracy improved from 0.76792 to 0.76962, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 6ms/step - loss: 0.5425 -
accuracy: 0.7829 - val_loss: 0.5942 - val_accuracy: 0.7696
Epoch 104/150
659/659 [==============================] - ETA: 0s - loss: 0.5406 - accuracy:
0.7836
Epoch 00104: val_accuracy did not improve from 0.76962
659/659 [==============================] - 4s 6ms/step - loss: 0.5406 -
accuracy: 0.7836 - val_loss: 0.5944 - val_accuracy: 0.7671
Epoch 105/150
653/659 [============================>.] - ETA: 0s - loss: 0.5379 - accuracy:
0.7868
Epoch 00105: val_accuracy did not improve from 0.76962
659/659 [==============================] - 4s 5ms/step - loss: 0.5370 -
accuracy: 0.7873 - val_loss: 0.5913 - val_accuracy: 0.7688
Epoch 106/150
657/659 [============================>.] - ETA: 0s - loss: 0.5421 - accuracy:
0.7795
Epoch 00106: val_accuracy did not improve from 0.76962
659/659 [==============================] - 4s 5ms/step - loss: 0.5416 -
accuracy: 0.7798 - val_loss: 0.5916 - val_accuracy: 0.7696
Epoch 107/150
657/659 [============================>.] - ETA: 0s - loss: 0.5378 - accuracy:
0.7831
Epoch 00107: val_accuracy did not improve from 0.76962
659/659 [==============================] - 4s 6ms/step - loss: 0.5372 -
accuracy: 0.7835 - val_loss: 0.5894 - val_accuracy: 0.7696
Epoch 108/150
652/659 [============================>.] - ETA: 0s - loss: 0.5380 - accuracy:
0.7856
Epoch 00108: val_accuracy did not improve from 0.76962
659/659 [==============================] - 4s 6ms/step - loss: 0.5381 -
accuracy: 0.7858 - val_loss: 0.5885 - val_accuracy: 0.7679
Epoch 109/150
651/659 [============================>.] - ETA: 0s - loss: 0.5391 - accuracy:
0.7824
Epoch 00109: val_accuracy did not improve from 0.76962
659/659 [==============================] - 4s 6ms/step - loss: 0.5391 -
accuracy: 0.7824 - val_loss: 0.5881 - val_accuracy: 0.7679
Epoch 110/150
658/659 [============================>.] - ETA: 0s - loss: 0.5349 - accuracy:
```

```
0.7826
Epoch 00110: val_accuracy improved from 0.76962 to 0.77218, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 6ms/step - loss: 0.5349 -
accuracy: 0.7825 - val_loss: 0.5869 - val_accuracy: 0.7722
Epoch 111/150
657/659 [============================>.] - ETA: 0s - loss: 0.5346 - accuracy:
0.7864
Epoch 00111: val_accuracy did not improve from 0.77218
659/659 [==============================] - 4s 6ms/step - loss: 0.5344 -
accuracy: 0.7865 - val_loss: 0.5858 - val_accuracy: 0.7722
Epoch 112/150
651/659 [============================>.] - ETA: 0s - loss: 0.5356 - accuracy:
0.7857
Epoch 00112: val_accuracy improved from 0.77218 to 0.77389, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 6ms/step - loss: 0.5356 -
accuracy: 0.7856 - val_loss: 0.5859 - val_accuracy: 0.7739
Epoch 113/150
653/659 [============================>.] - ETA: 0s - loss: 0.5342 - accuracy:
0.7839
Epoch 00113: val_accuracy improved from 0.77389 to 0.77560, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 6ms/step - loss: 0.5354 -
accuracy: 0.7833 - val_loss: 0.5843 - val_accuracy: 0.7756
Epoch 114/150
650/659 [============================>.] - ETA: 0s - loss: 0.5283 - accuracy:
0.7876
Epoch 00114: val_accuracy did not improve from 0.77560
659/659 [==============================] - 4s 6ms/step - loss: 0.5284 -
accuracy: 0.7876 - val_loss: 0.5830 - val_accuracy: 0.7705
Epoch 115/150
653/659 [============================>.] - ETA: 0s - loss: 0.5293 - accuracy:
0.7883
Epoch 00115: val_accuracy did not improve from 0.77560
659/659 [==============================] - 4s 6ms/step - loss: 0.5292 -
accuracy: 0.7886 - val_loss: 0.5856 - val_accuracy: 0.7730
Epoch 116/150
657/659 [============================>.] - ETA: 0s - loss: 0.5292 - accuracy:
0.7900
Epoch 00116: val_accuracy improved from 0.77560 to 0.77986, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 6ms/step - loss: 0.5290 -
accuracy: 0.7898 - val_loss: 0.5812 - val_accuracy: 0.7799
Epoch 117/150
657/659 [============================>.] - ETA: 0s - loss: 0.5304 - accuracy:
0.7863
Epoch 00117: val_accuracy did not improve from 0.77986
```

```
659/659 [==============================] - 4s 6ms/step - loss: 0.5310 -
accuracy: 0.7861 - val_loss: 0.5811 - val_accuracy: 0.7739
Epoch 118/150
652/659 [============================>.] - ETA: 0s - loss: 0.5257 - accuracy:
0.7896
Epoch 00118: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 5ms/step - loss: 0.5261 -
accuracy: 0.7889 - val_loss: 0.5798 - val_accuracy: 0.7765
Epoch 119/150
654/659 [============================>.] - ETA: 0s - loss: 0.5274 - accuracy:
0.7872
Epoch 00119: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 5ms/step - loss: 0.5276 -
accuracy: 0.7873 - val_loss: 0.5788 - val_accuracy: 0.7747
Epoch 120/150
654/659 [============================>.] - ETA: 0s - loss: 0.5265 - accuracy:
0.7882
Epoch 00120: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 6ms/step - loss: 0.5264 -
accuracy: 0.7886 - val_loss: 0.5804 - val_accuracy: 0.7739
Epoch 121/150
657/659 [============================>.] - ETA: 0s - loss: 0.5266 - accuracy:
0.7906
Epoch 00121: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 6ms/step - loss: 0.5260 -
accuracy: 0.7911 - val_loss: 0.5773 - val_accuracy: 0.7790
Epoch 122/150
655/659 [============================>.] - ETA: 0s - loss: 0.5253 - accuracy:
0.7906
Epoch 00122: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 5ms/step - loss: 0.5249 -
accuracy: 0.7910 - val_loss: 0.5762 - val_accuracy: 0.7782
Epoch 123/150
653/659 [============================>.] - ETA: 0s - loss: 0.5218 - accuracy:
0.7902
Epoch 00123: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 5ms/step - loss: 0.5222 -
accuracy: 0.7898 - val_loss: 0.5780 - val_accuracy: 0.7773
Epoch 124/150
658/659 [============================>.] - ETA: 0s - loss: 0.5203 - accuracy:
0.7915
Epoch 00124: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 6ms/step - loss: 0.5203 -
accuracy: 0.7916 - val_loss: 0.5755 - val_accuracy: 0.7765
Epoch 125/150
650/659 [============================>.] - ETA: 0s - loss: 0.5195 - accuracy:
0.7919
Epoch 00125: val_accuracy did not improve from 0.77986
```

```
659/659 [==============================] - 4s 6ms/step - loss: 0.5190 -
accuracy: 0.7917 - val_loss: 0.5749 - val_accuracy: 0.7799
Epoch 126/150
652/659 [============================>.] - ETA: 0s - loss: 0.5205 - accuracy:
0.7895
Epoch 00126: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 6ms/step - loss: 0.5202 -
accuracy: 0.7896 - val_loss: 0.5737 - val_accuracy: 0.7799
Epoch 127/150
652/659 [============================>.] - ETA: 0s - loss: 0.5199 - accuracy:
0.7895
Epoch 00127: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 6ms/step - loss: 0.5207 -
accuracy: 0.7891 - val_loss: 0.5746 - val_accuracy: 0.7782
Epoch 128/150
651/659 [============================>.] - ETA: 0s - loss: 0.5163 - accuracy:
0.7911
Epoch 00128: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 6ms/step - loss: 0.5167 -
accuracy: 0.7907 - val_loss: 0.5765 - val_accuracy: 0.7756
Epoch 129/150
652/659 [============================>.] - ETA: 0s - loss: 0.5166 - accuracy:
0.7901
Epoch 00129: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 6ms/step - loss: 0.5172 -
accuracy: 0.7905 - val_loss: 0.5733 - val_accuracy: 0.7773
Epoch 130/150
651/659 [============================>.] - ETA: 0s - loss: 0.5195 - accuracy:
0.7891
Epoch 00130: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 6ms/step - loss: 0.5189 -
accuracy: 0.7891 - val_loss: 0.5721 - val_accuracy: 0.7782
Epoch 131/150
654/659 [============================>.] - ETA: 0s - loss: 0.5156 - accuracy:
0.7946
Epoch 00131: val_accuracy did not improve from 0.77986
659/659 [==============================] - 4s 6ms/step - loss: 0.5159 -
accuracy: 0.7945 - val_loss: 0.5716 - val_accuracy: 0.7790
Epoch 132/150
655/659 [============================>.] - ETA: 0s - loss: 0.5126 - accuracy:
0.7966
Epoch 00132: val_accuracy improved from 0.77986 to 0.78072, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 6ms/step - loss: 0.5128 -
accuracy: 0.7964 - val_loss: 0.5721 - val_accuracy: 0.7807
Epoch 133/150
654/659 [============================>.] - ETA: 0s - loss: 0.5122 - accuracy:
0.7950
```

```
Epoch 00133: val_accuracy improved from 0.78072 to 0.78157, saving model to
best_accuracy.h5
659/659 [==============================] - 4s 6ms/step - loss: 0.5125 -
accuracy: 0.7949 - val_loss: 0.5718 - val_accuracy: 0.7816
Epoch 134/150
655/659 [============================>.] - ETA: 0s - loss: 0.5162 - accuracy:
0.7958
Epoch 00134: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5155 -
accuracy: 0.7960 - val_loss: 0.5712 - val_accuracy: 0.7790
Epoch 135/150
659/659 [==============================] - ETA: 0s - loss: 0.5133 - accuracy:
0.7907
Epoch 00135: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5133 -
accuracy: 0.7907 - val_loss: 0.5705 - val_accuracy: 0.7773
Epoch 136/150
653/659 [============================>.] - ETA: 0s - loss: 0.5090 - accuracy:
0.7935
Epoch 00136: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5109 -
accuracy: 0.7928 - val_loss: 0.5710 - val_accuracy: 0.7799
Epoch 137/150
659/659 [==============================] - ETA: 0s - loss: 0.5111 - accuracy:
0.7932
Epoch 00137: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5111 -
accuracy: 0.7932 - val_loss: 0.5702 - val_accuracy: 0.7782
Epoch 138/150
657/659 [============================>.] - ETA: 0s - loss: 0.5081 - accuracy:
0.7956
Epoch 00138: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5076 -
accuracy: 0.7957 - val_loss: 0.5705 - val_accuracy: 0.7807
Epoch 139/150
658/659 [============================>.] - ETA: 0s - loss: 0.5073 - accuracy:
0.7955
Epoch 00139: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5076 -
accuracy: 0.7954 - val_loss: 0.5686 - val_accuracy: 0.7816
Epoch 140/150
653/659 [============================>.] - ETA: 0s - loss: 0.5063 - accuracy:
0.7961
Epoch 00140: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5062 -
accuracy: 0.7964 - val_loss: 0.5691 - val_accuracy: 0.7799
Epoch 141/150
657/659 [============================>.] - ETA: 0s - loss: 0.5081 - accuracy:
```

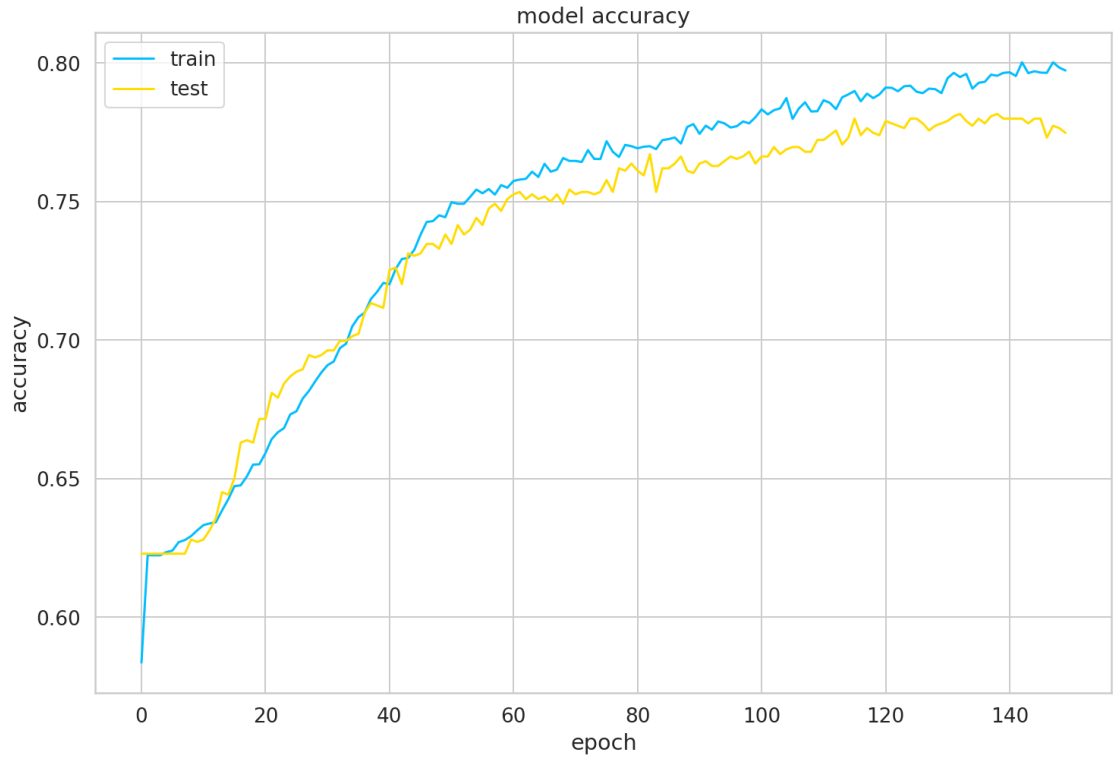
```
0.7968
Epoch 00141: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5085 -
accuracy: 0.7966 - val_loss: 0.5691 - val_accuracy: 0.7799
Epoch 142/150
657/659 [============================>.] - ETA: 0s - loss: 0.5071 - accuracy:
0.7954
Epoch 00142: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5071 -
accuracy: 0.7953 - val_loss: 0.5683 - val_accuracy: 0.7799
Epoch 143/150
657/659 [============================>.] - ETA: 0s - loss: 0.5044 - accuracy:
0.7999
Epoch 00143: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5038 -
accuracy: 0.8002 - val_loss: 0.5677 - val_accuracy: 0.7799
Epoch 144/150
657/659 [============================>.] - ETA: 0s - loss: 0.5065 - accuracy:
0.7962
Epoch 00144: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5068 -
accuracy: 0.7963 - val_loss: 0.5683 - val_accuracy: 0.7782
Epoch 145/150
657/659 [============================>.] - ETA: 0s - loss: 0.5055 - accuracy:
0.7968
Epoch 00145: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5055 -
accuracy: 0.7970 - val_loss: 0.5671 - val_accuracy: 0.7799
Epoch 146/150
653/659 [============================>.] - ETA: 0s - loss: 0.5046 - accuracy:
0.7964
Epoch 00146: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5043 -
accuracy: 0.7965 - val_loss: 0.5686 - val_accuracy: 0.7799
Epoch 147/150
654/659 [============================>.] - ETA: 0s - loss: 0.5048 - accuracy:
0.7963
Epoch 00147: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5040 -
accuracy: 0.7964 - val_loss: 0.5689 - val_accuracy: 0.7730
Epoch 148/150
659/659 [==============================] - ETA: 0s - loss: 0.5013 - accuracy:
0.8002
Epoch 00148: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5013 -
accuracy: 0.8002 - val_loss: 0.5662 - val_accuracy: 0.7773
Epoch 149/150
654/659 [============================>.] - ETA: 0s - loss: 0.5027 - accuracy:
```

```
0.7985
Epoch 00149: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5031 -
accuracy: 0.7983 - val_loss: 0.5681 - val_accuracy: 0.7765
Epoch 150/150
656/659 [============================>.] - ETA: 0s - loss: 0.5025 - accuracy:
0.7974
Epoch 00150: val_accuracy did not improve from 0.78157
659/659 [==============================] - 4s 6ms/step - loss: 0.5025 -
accuracy: 0.7972 - val_loss: 0.5681 - val_accuracy: 0.7747
```

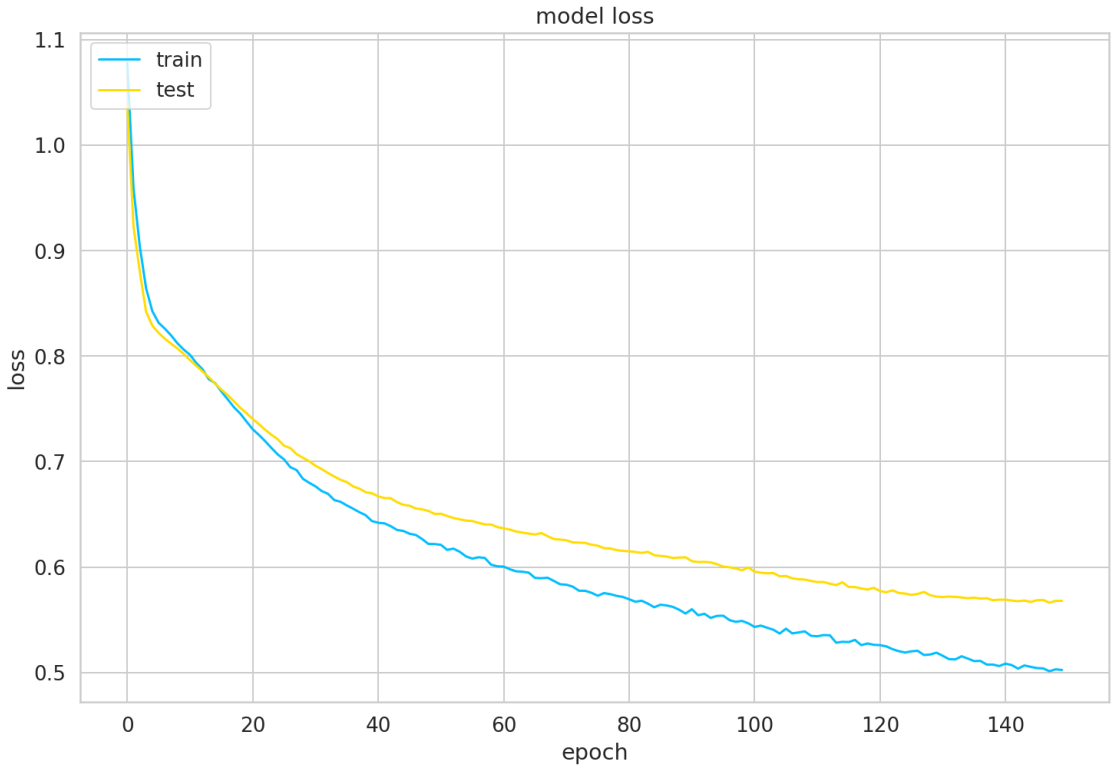
It is worth mentionning that it takes around 4 hours to train with the above mentionned GPU.

We then visualize the model accuracy and loss. We can notice that the performance of the model grows in the validation set until epoch 120.

[63]:
```
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



[64]:
```
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



We of course load the best model performance wise, so that we can use it to judge the performance and specs of this approach :

[60]:
```
# Loading the best model
lstm_model = load_model("./best_accuracy.h5")
```

We run an evaluation on the test dataset. As it appears, there is no drop in accuracy, which means that the model is able to generalize fairly well and did not overfit.

[61]:
```
lstm_model.evaluate(features_test, label_test)
```

```
92/92 [==============================] - 0s 3ms/step - loss: 0.5482 - accuracy:
0.7804
```

[61]: [0.5481827259063721, 0.7803961634635925]

To understand better the performance of our model, we'll extract the ground truth labels and the predictions on the test dataset :

[62]:
```
# Getting the index ot the best prediction for each entry, to keep in the same␣
↪format of ground truths
lstm_labels_pred = np.argmax(lstm_model.predict(features_test), axis=1).tolist()
```

[63]:
```
# reverse one hot encoding
lstm_label_truth = []
for index, row in label_test.iterrows():
  if row[0] == 1:
    value = 0
  elif row[1] == 1:
    value = 1
  else :
    value = 2
  lstm_label_truth.append(value)
```

A detailled report of our model shows the following :

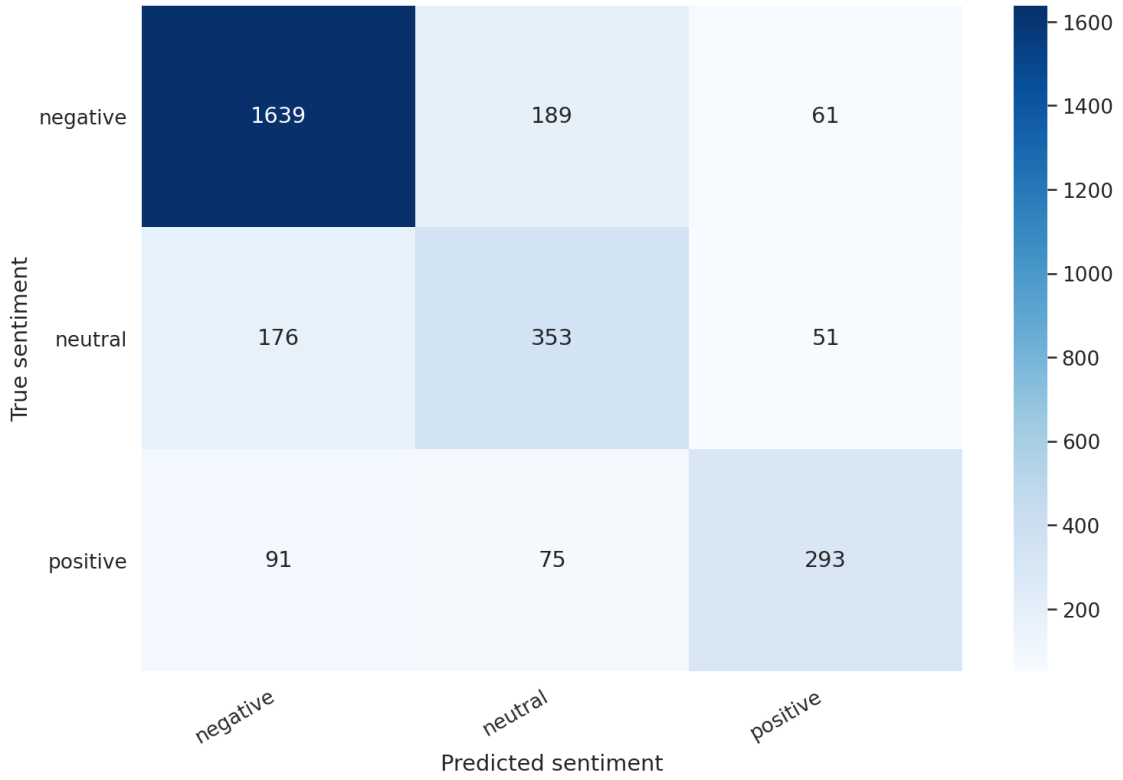
[64]:
```
print(classification_report(lstm_label_truth, lstm_labels_pred,␣
↪target_names=class_names))
```

```
              precision    recall  f1-score   support

    negative       0.86      0.87      0.86      1889
     neutral       0.57      0.61      0.59       580
    positive       0.72      0.64      0.68       459

    accuracy                           0.78      2928
   macro avg       0.72      0.70      0.71      2928
weighted avg       0.78      0.78      0.78      2928
```

The confusion matrix of the model shows that the performance ...

[65]:
```
def show_confusion_matrix(confusion_matrix):
  hmap = sns.heatmap(confusion_matrix, annot=True, fmt="d", cmap="Blues")
  hmap.yaxis.set_ticklabels(hmap.yaxis.get_ticklabels(), rotation=0, ha='right')
  hmap.xaxis.set_ticklabels(hmap.xaxis.get_ticklabels(), rotation=30,␣
↪ha='right')
  plt.ylabel('True sentiment')
  plt.xlabel('Predicted sentiment');

cm = confusion_matrix(lstm_label_truth, lstm_labels_pred)
df_cm = pd.DataFrame(cm, index=class_names, columns=class_names)
```

```
show_confusion_matrix(df_cm)
```



3 - Baseline method : Textblob In this step, we'll be working with a ready to use library for sentiment analysis, TextBlob. As it is used widely in industry, we'll use it as a baseline to compare with our models.

As the library is ready to use, we'll be using it directly on the test dataset.

[66]:
```
df_test.head()
```

[66]:

	review	…	airline_sentiment
4794	you're my early frontrunner for best airline! …	…	positive
10480	how is it that my flt to EWR was Cancelled Fli…	…	negative
8067	what is going on with your BDL to DCA flights …	…	negative
8880	do they have to depart from Washington, D.C.??	…	neutral
8292	I can probably find some of them. Are the tick…	…	negative

[5 rows x 5 columns]

We define a function for prediction, with threshold parameters chosen using web recommendations of users of the library. As the sentiment prediction is made on sentences, we take the mean of the total sentiment score for a review.

[67]:
```
def textblob_predict(text):
  blob = TextBlob(text)
  sentiment = 0
  count_sentences = 0
  for sentence in blob.sentences:
    count_sentences += 1
    sentiment += sentence.sentiment.polarity
  sentiment = sentiment / count_sentences
  if sentiment > 0.1 :
    category = 2
  elif sentiment < -0.1 :
    category = 0
  else :
    category = 1
  return category
```

We then wrangle our ground truth and predictions into a proper format, to present the classification report results.

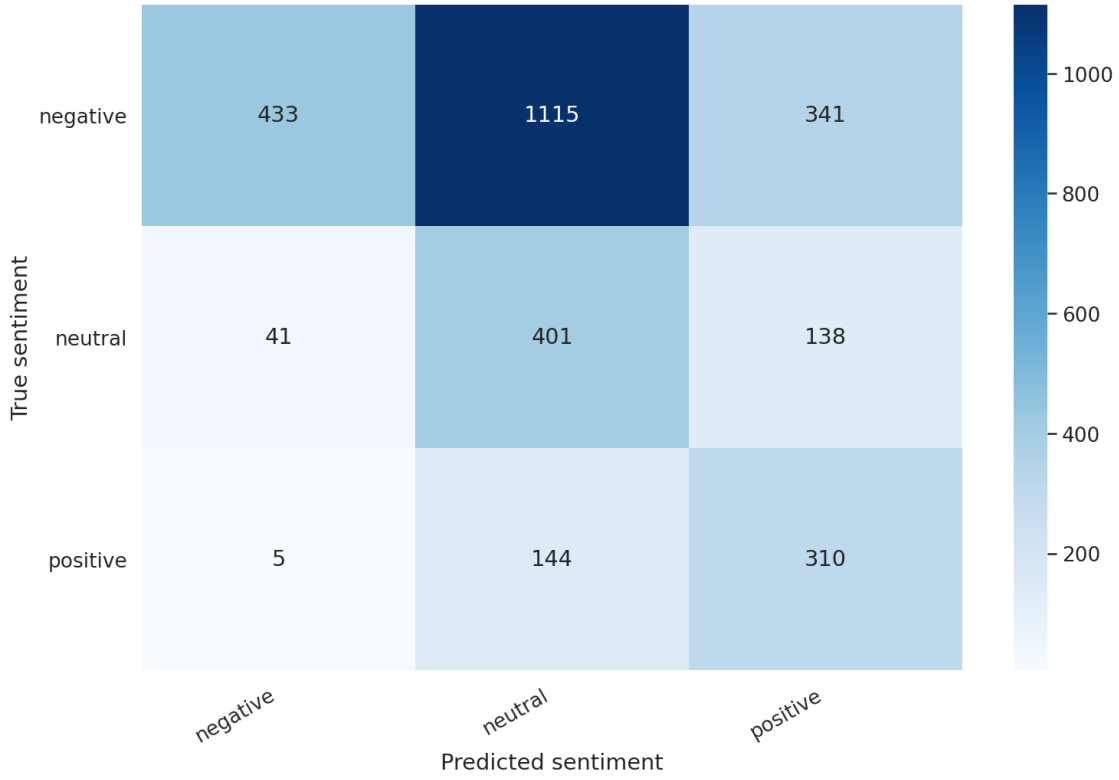
[68]:
```
textblob_label_truth = []
textblob_label_pred = []
for index, row in df_test.iterrows():
  textblob_label_pred.append(textblob_predict(row['review']))
  textblob_label_truth.append(row['target'])
```

[69]:
```
print(classification_report(textblob_label_truth, textblob_label_pred,␣
 ↪target_names=class_names))
```

```
              precision    recall  f1-score   support

    negative       0.90      0.23      0.37      1889
     neutral       0.24      0.69      0.36       580
    positive       0.39      0.68      0.50       459

    accuracy                           0.39      2928
   macro avg       0.51      0.53      0.41      2928
weighted avg       0.69      0.39      0.38      2928
```

The library shows a huge deficiency in performance. The confusion matrix that follows only proves it further, as it sees most observations as either neutral or positive, with a bias against the negative.

[70]:
```
cm = confusion_matrix(textblob_label_truth, textblob_label_pred)
df_cm = pd.DataFrame(cm, index=class_names, columns=class_names)
show_confusion_matrix(df_cm)
```



1.0.4 III - Document embeddings

In this section, we'll be having a big focus on representing our reviews by vectors of features, using Doc2vec. Doc2vec is an unsupervised approach build on top of word2vec, that unlike word2vec that can make vector representations of words while taking into account context, doc2vec can make vector representations of documents.

These vector embeddings can then be used as input for various NLP models, or used for visualisation of document similarity. In our task, after training the doc2vec on our data, we'll be using it for sentiment classification.

1 - Training doc2vec We first begin by defining a tokenization function that uses `nltk` at its core.

[71]:
```
def tokenize_text(text):
  tokens = []
  for sent in nltk.sent_tokenize(text):
      for word in nltk.word_tokenize(sent):
          if len(word) < 2:
              continue
          tokens.append(word.lower())
  return tokens
```

We will then separate our training and testing dataset (same test dataset throughout the notebook), to create the TaggedDocument object we'll be running through the doc2vec.

[72]:
```
train_documents = []
test_documents = []

tags_index = {'negative': 0 , 'neutral': 1, 'positive': 2}
```

[73]:
```
for index, row in df_temp.iterrows():
  train_documents.append(TaggedDocument(words=tokenize_text(row['review']),␣
 ↪tags=[tags_index.get(row['airline_sentiment'], 8)]))


for index, row in df_test.iterrows():
  test_documents.append(TaggedDocument(words=tokenize_text(row['review']),␣
 ↪tags=[tags_index.get(row['airline_sentiment'], 8)]))
```

[74]:
```
print(train_documents[0])
```

```
TaggedDocument(['you', 'are', 'offering', 'us', 'rooms', 'for', '32', 'people',
'fail'], [0])
```

Next step is getting the document feature vectors. We'll be using multiprocessing in order to speed up the training. Each process and core of the device would be used.

[75]:
```
cores = multiprocessing.cpu_count()
```

While there are two types of doc2vec models, we decide to use Distributed bag of words (PV-DBOW), as it is more adapted and shows more performance. After a number of finetuning steps, we define the model to have an output of three features vector that can encodes the information of the document.

[76]:
```
model_dbow = Doc2Vec(dm=0, vector_size=3, negative=6,window=10, hs=0,␣
 ↪min_count=8, sample = 0, workers=cores, alpha=0.025, min_alpha=0.001)
```

We then build the vocabulary of our model, using the words that are in our training dataset.

[77]:
```
model_dbow.build_vocab([x for x in tqdm(train_documents)])
```

```
100%|      | 11712/11712 [00:00<00:00, 1470460.93it/s]
```

And after shuffling the data, we launch training of the model for 30 epoch, and then save the model that results.

[78]:
```
train_documents  = utils.shuffle(train_documents)
```

[79]:
```
model_dbow.train(train_documents,total_examples=len(train_documents), epochs=30)
```

[80]:
```
model_dbow.save('./movieModel.d2v')
```

After training, we define a helper function that helps us encode documents into their feature vectors. We then use it to encode our training and test documents, for usage in sentiment classification later.

[81]:
```
def vector_for_learning(model, input_docs):
  sents = input_docs
  targets, feature_vectors = zip(*[(doc.tags[0], model.infer_vector(doc.words,␣
 ↪steps=20)) for doc in sents])
  return targets, feature_vectors
```

[82]:
```
y_train, X_train = vector_for_learning(model_dbow, train_documents)
y_test, X_test = vector_for_learning(model_dbow, test_documents)
```

[83]:
```
# Size of the feature encoding for each document
len("feature encoding size is " + str(X_train[0]))
```

[83]:
```
62
```

We transform our test data feature vectors into a dataframe, and add to it its ground truth labels.

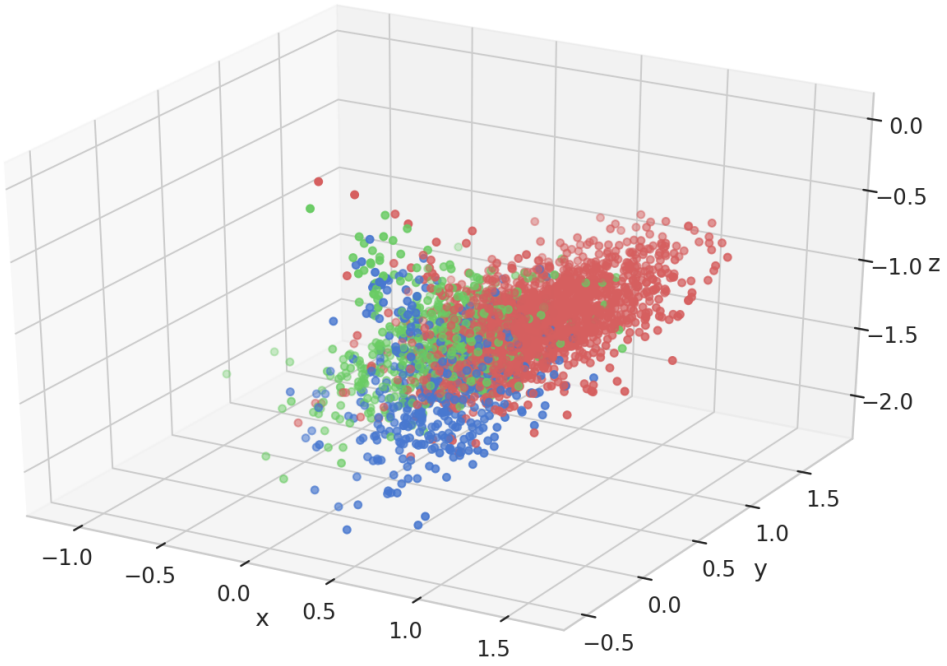
[84]:
```
df_pca_test = pd.DataFrame(X_test)
df_pca_test['y_test'] = pd.DataFrame(y_test)
```

The goal of this next step is to visualize all our test documents in the feature space of documents, highlighted by their sentiment. As our doc2vec model only worked in training data, we'll be able to test how well it generalizes.

As further seen below, we can see that the three sentiment classes are very fuzzily separate in the feature space of our test documents, which proves thatour doc2vec model can generalize on unseen data.

[85]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.scatter(df_pca_test[0].to_list(), df_pca_test[1].to_list(),df_pca_test[2].
 ↪to_list(),c=colormap[df_pca_test['y_test']])
```

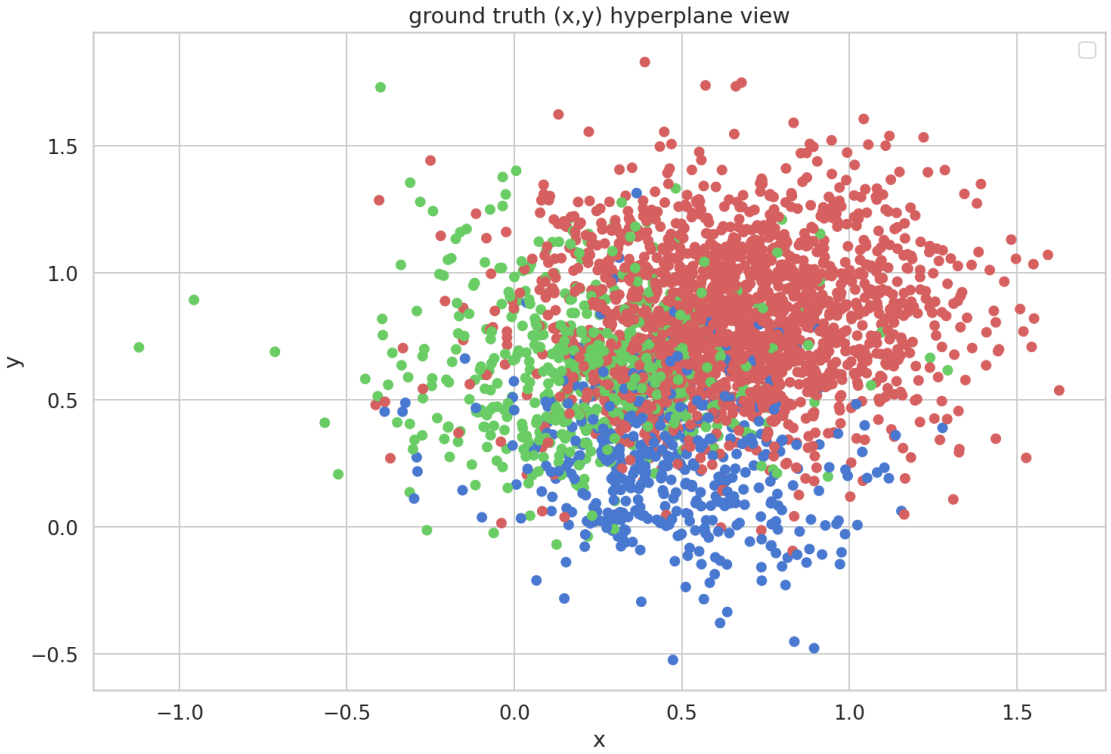
[85]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7fc2ed0ea6d8>
```



The following also shows views of each hyperplane of our feature space :

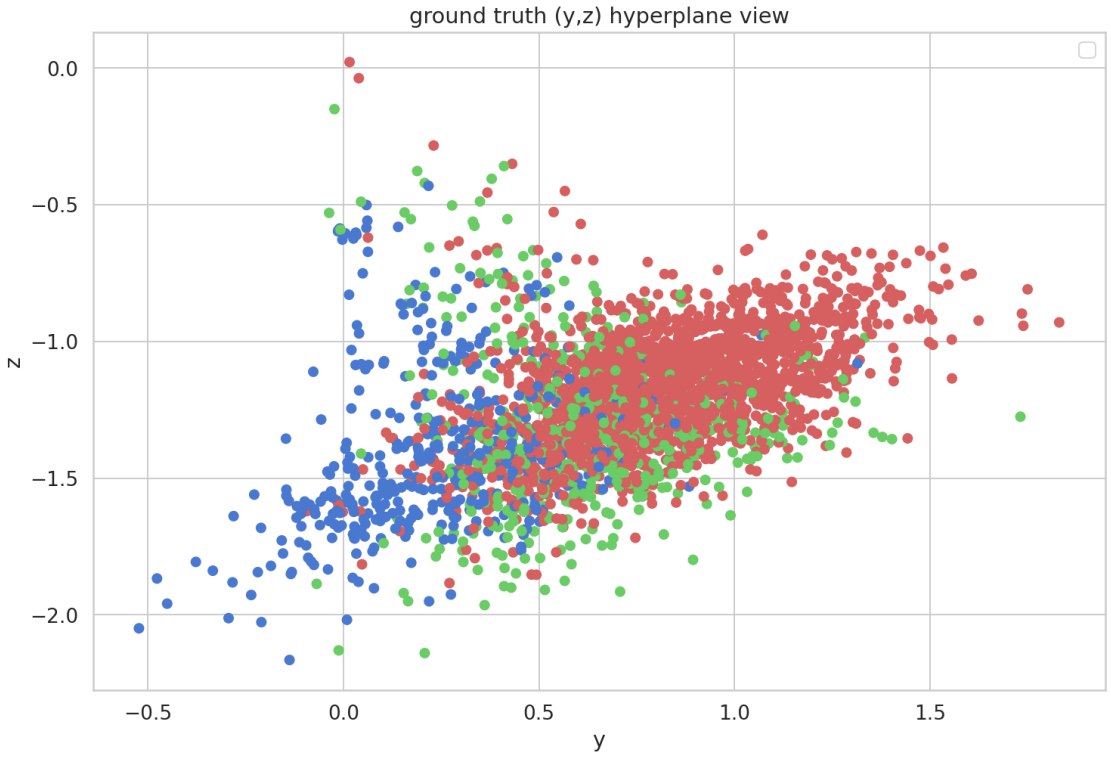
[86]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.title("ground truth (x,y) hyperplane view")
plt.scatter(df_pca_test[0], df_pca_test[1], c=colormap[df_pca_test['y_test']])
plt.show()
```

```
No handles with labels found to put in legend.
```



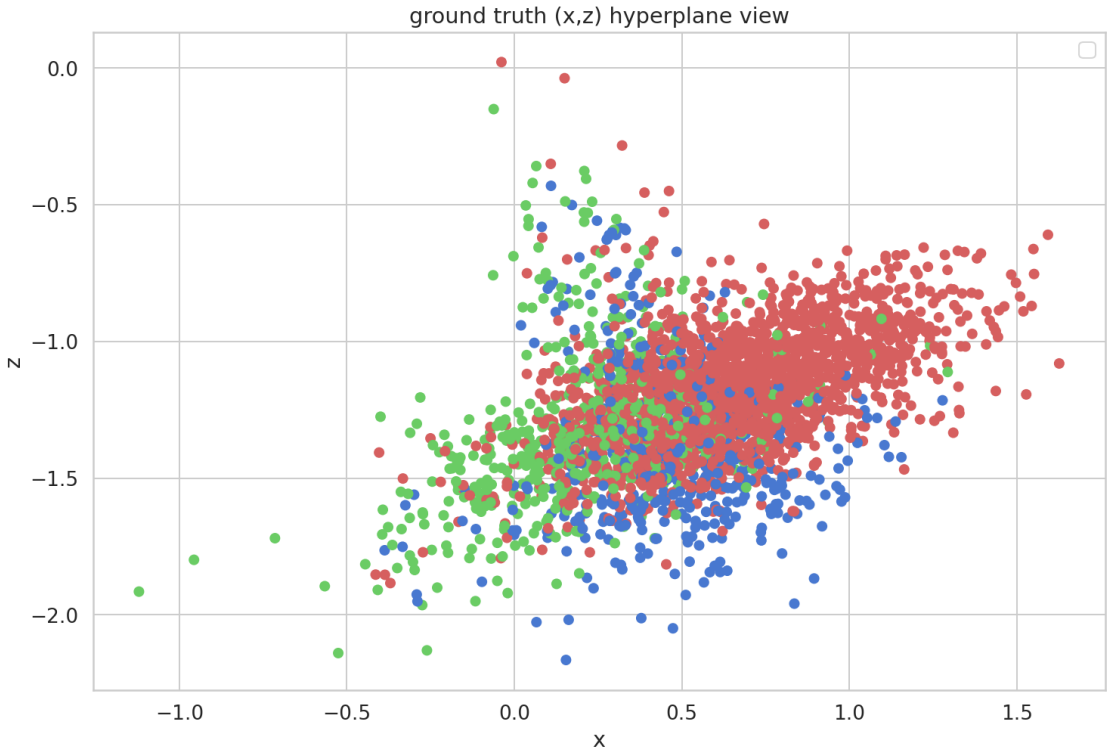
[87]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("y")
plt.ylabel("z")
plt.legend()
plt.title("ground truth (y,z) hyperplane view")
plt.scatter(df_pca_test[1], df_pca_test[2], c=colormap[df_pca_test['y_test']])
plt.show()
```

```
No handles with labels found to put in legend.
```



[88]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("x")
plt.ylabel("z")
plt.legend()
plt.title("ground truth (x,z) hyperplane view")
plt.scatter(df_pca_test[0], df_pca_test[2], c=colormap[df_pca_test['y_test']])
plt.show()
```

```
No handles with labels found to put in legend.
```



2 - Doc2vec Sentiment classifier Using the above encoded features of documents, we will aim to classify the sentiments of our documents using a logistic regression model.

[89]:
```
logreg = LogisticRegression(n_jobs=1, C=1e5)
logreg.fit(X_train, y_train)
logreg_label_pred = logreg.predict(X_test)
print('Testing accuracy for movie plots%s' % accuracy_score(y_test,␣
 ↪logreg_label_pred))
print('Testing precision score for movie plots: {}'.
 ↪format(precision_score(y_test, logreg_label_pred, average='weighted')))
print('Testing F1 score for movie plots: {}'.format(f1_score(y_test,␣
 ↪logreg_label_pred, average='weighted')))
```

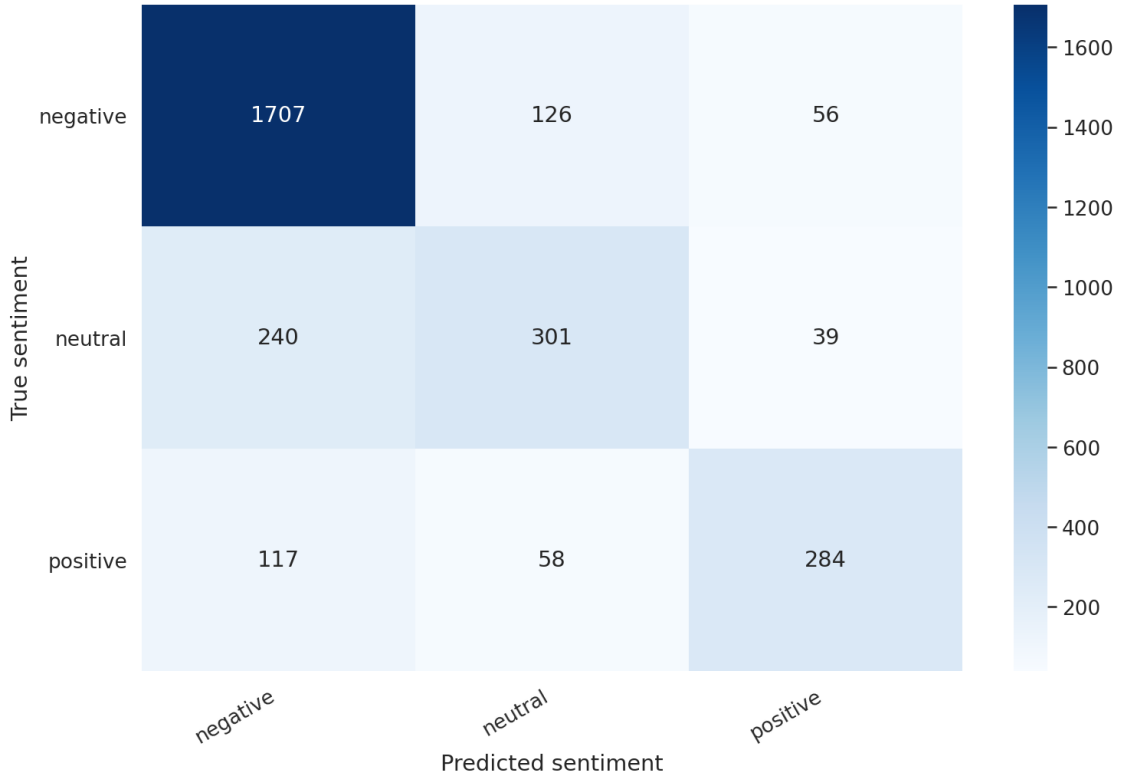
```
Testing accuracy for movie plots0.7827868852459017
Testing precision score for movie plots: 0.7739668332605486
Testing F1 score for movie plots: 0.7754073596780778
```

The model shows great performance, though similarly to the BERT model it shows difficulty recognizing `neutral` categories for their ambiguity. This can be observed through the classification report and the confusion matrix below.

[90]:
```
print(classification_report(y_test, logreg_label_pred,␣
 ↪target_names=class_names))
```

```
              precision    recall  f1-score   support

    negative       0.83      0.90      0.86      1889
     neutral       0.62      0.52      0.57       580
    positive       0.75      0.62      0.68       459

    accuracy                           0.78      2928
   macro avg       0.73      0.68      0.70      2928
weighted avg       0.77      0.78      0.78      2928
```

[91]:
```
def show_confusion_matrix(confusion_matrix):
  hmap = sns.heatmap(confusion_matrix, annot=True, fmt="d", cmap="Blues")
  hmap.yaxis.set_ticklabels(hmap.yaxis.get_ticklabels(), rotation=0, ha='right')
  hmap.xaxis.set_ticklabels(hmap.xaxis.get_ticklabels(), rotation=30,␣
 ↪ha='right')
  plt.ylabel('True sentiment')
  plt.xlabel('Predicted sentiment');

cm = confusion_matrix(y_test, logreg_label_pred)
df_cm = pd.DataFrame(cm, index=class_names, columns=class_names)
show_confusion_matrix(df_cm)
```



1.0.5 IV - Model performance visualisation

In this last step, we'll be comparing the performance of our various models by visualizing their sentiment prediction in the feature space for each document. Document of the same sentiment should be close to each other, similarly to the visualization of the ground truth above.

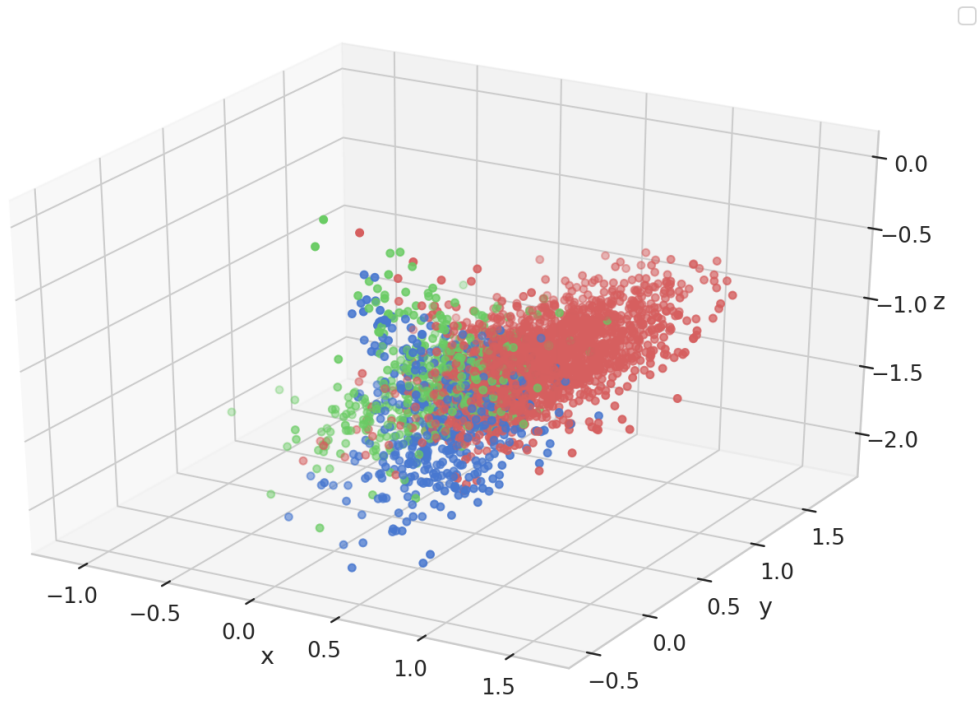
1 - Bert model

[92]:
```
df_pca_test['bert_y_pred'] = pd.DataFrame(bert_y_pred)
```

[93]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.legend()
ax.grid(True)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.scatter(df_pca_test[0].to_list(), df_pca_test[1].to_list(),df_pca_test[2].
 ↪to_list(),c=colormap[df_pca_test['bert_y_pred']])
```

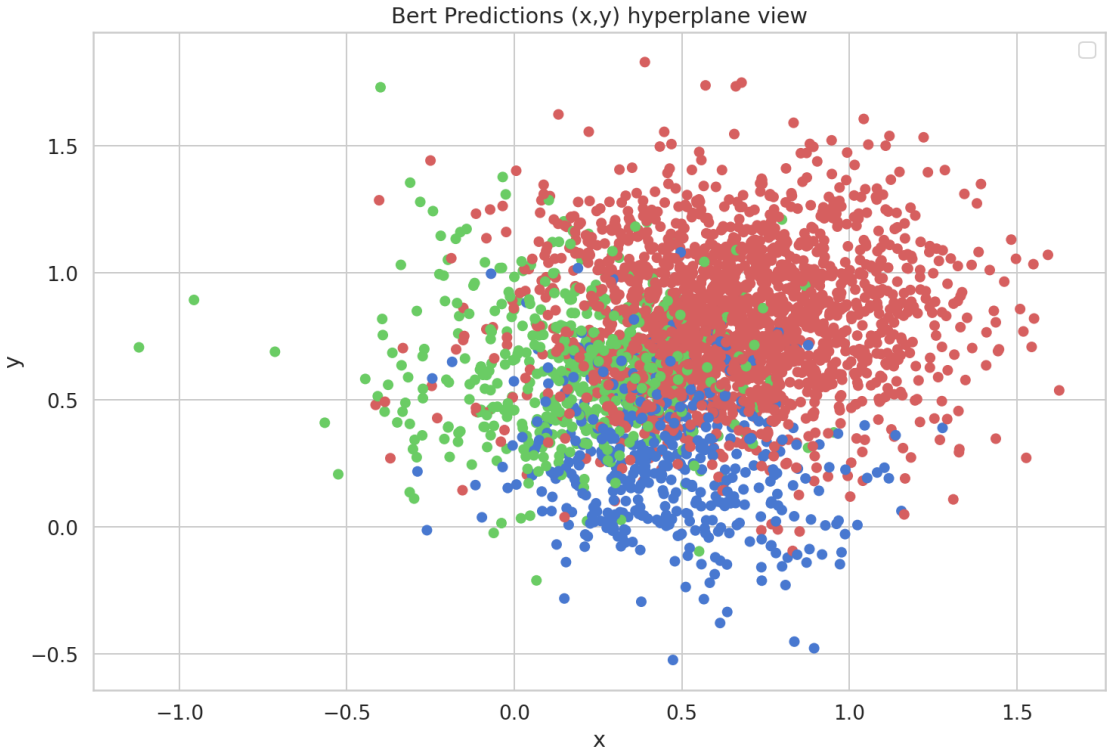
```
No handles with labels found to put in legend.
```

[93]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7fc2ececf710>
```



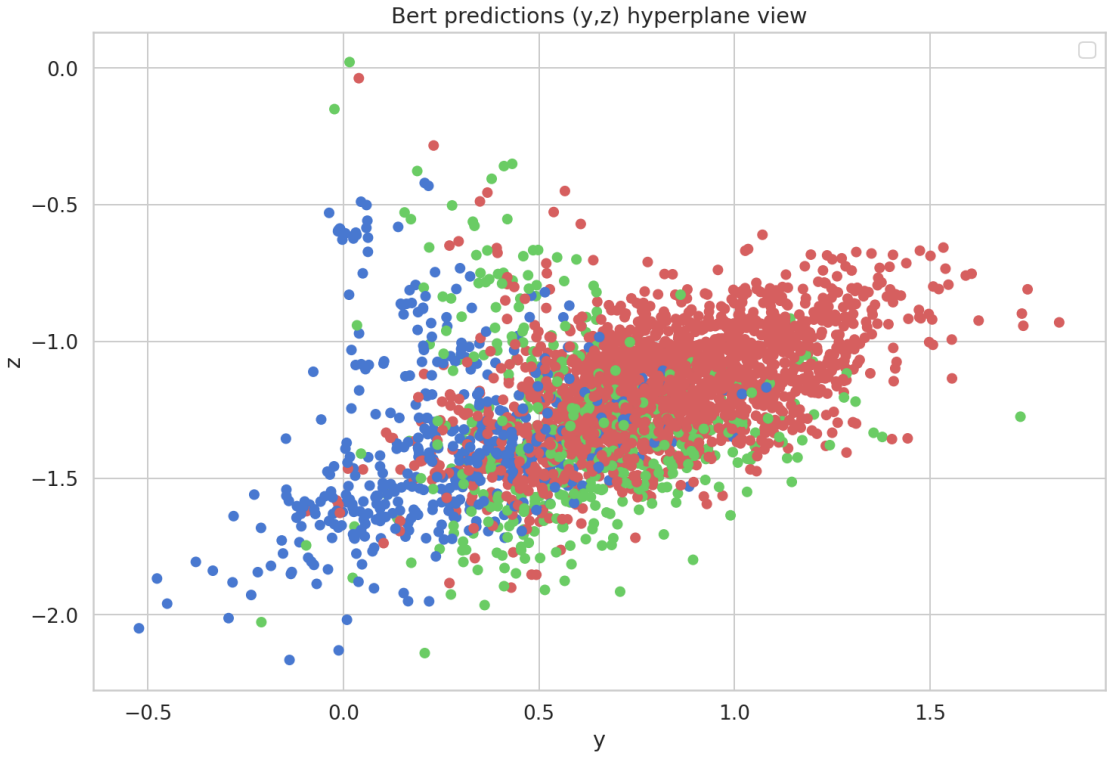
[94]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.title("Bert Predictions (x,y) hyperplane view")
plt.scatter(df_pca_test[0], df_pca_test[1],␣
 ↪c=colormap[df_pca_test['bert_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



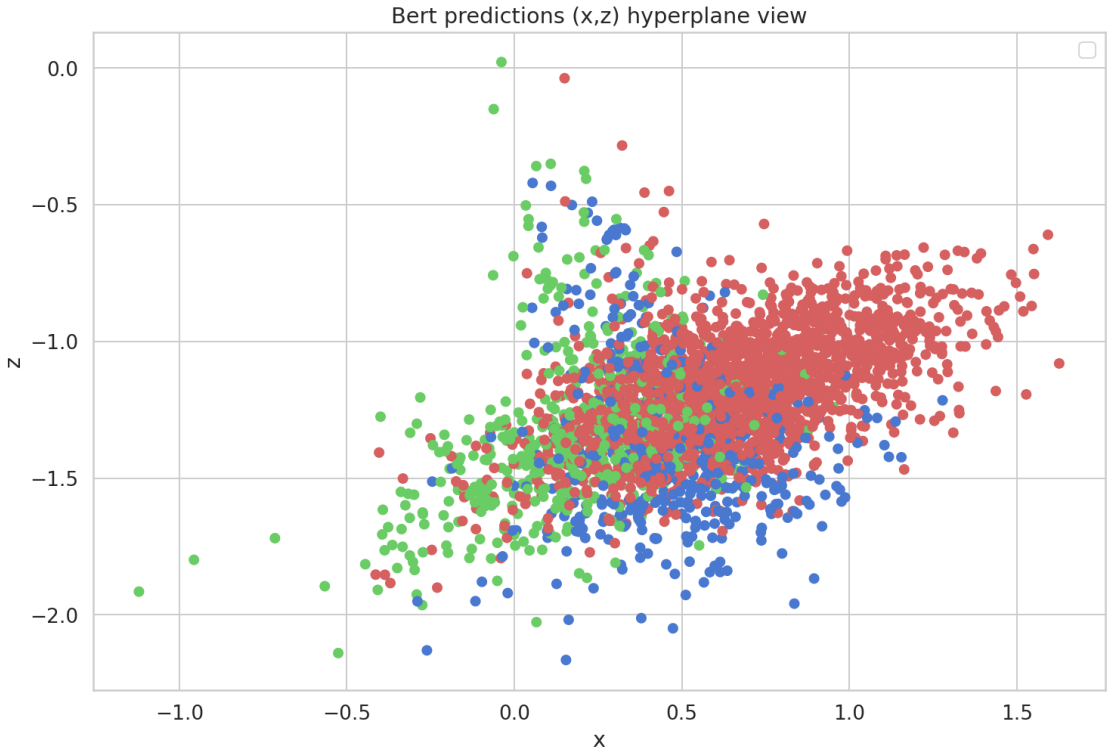
[95]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("y")
plt.ylabel("z")
plt.legend()
plt.title("Bert predictions (y,z) hyperplane view")
plt.scatter(df_pca_test[1], df_pca_test[2],␣
 ↪c=colormap[df_pca_test['bert_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



[96]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("x")
plt.ylabel("z")
plt.legend()
plt.title("Bert predictions (x,z) hyperplane view")
plt.scatter(df_pca_test[0], df_pca_test[2],␣
 ↪c=colormap[df_pca_test['bert_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



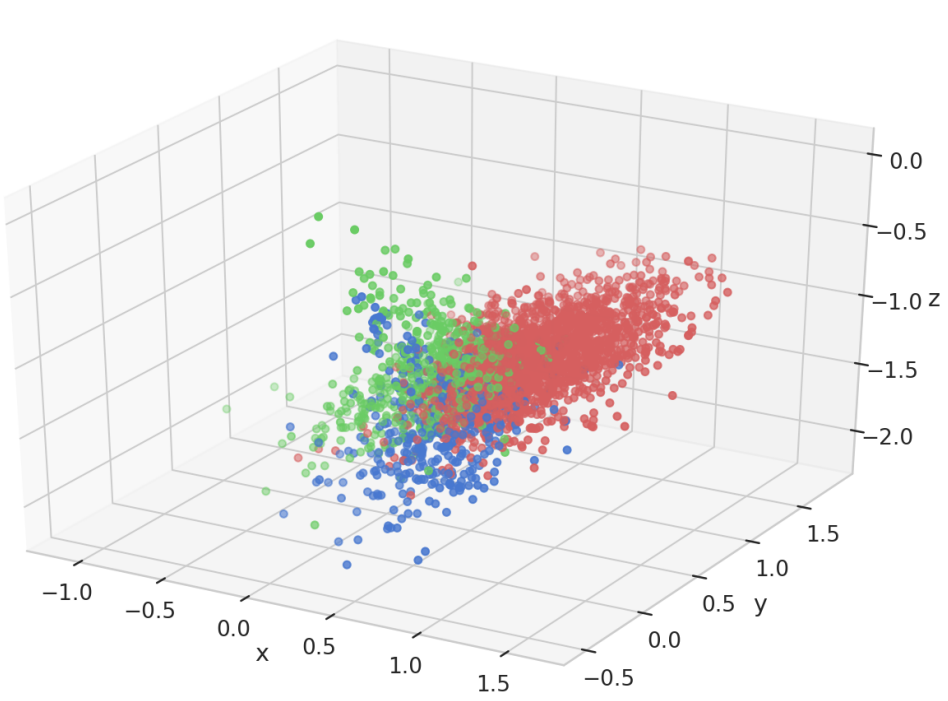
2 - LSTM model

[97]:
```
df_pca_test['lstm_y_pred'] = pd.DataFrame(lstm_labels_pred)
```

[98]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.legend()
ax.grid(True)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.scatter(df_pca_test[0].to_list(), df_pca_test[1].to_list(),df_pca_test[2].
 ↪to_list(),c=colormap[df_pca_test['lstm_y_pred']])
```

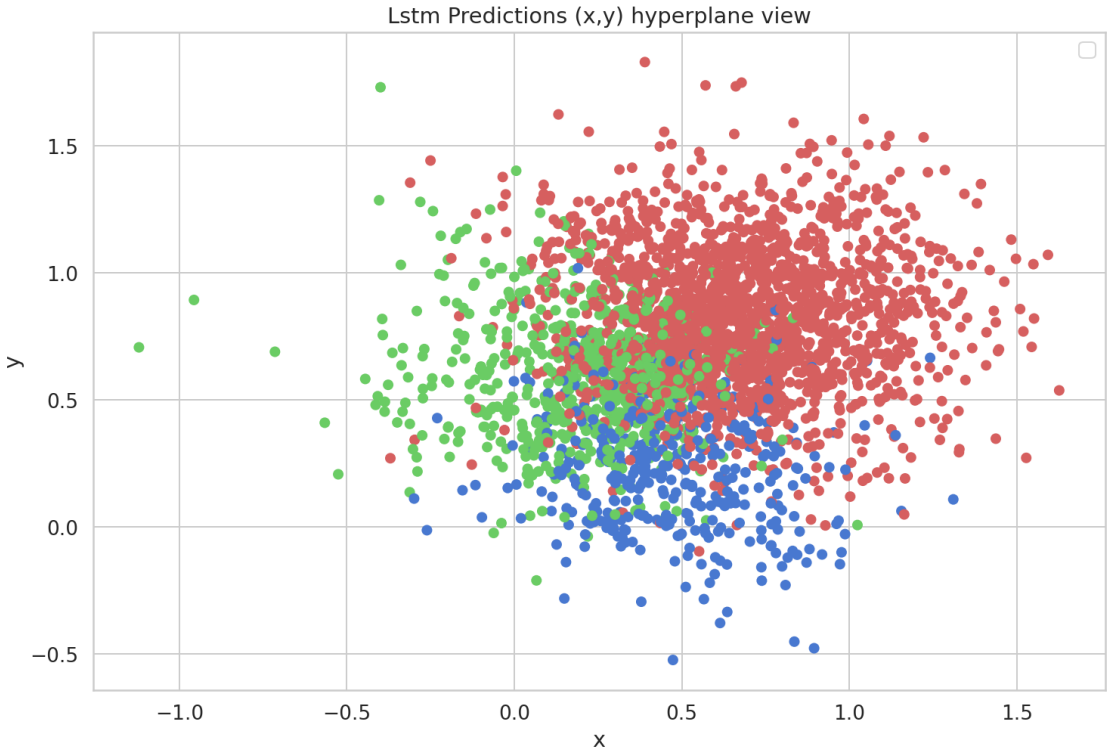
```
No handles with labels found to put in legend.
```

[98]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7fc2ecd95160>
```



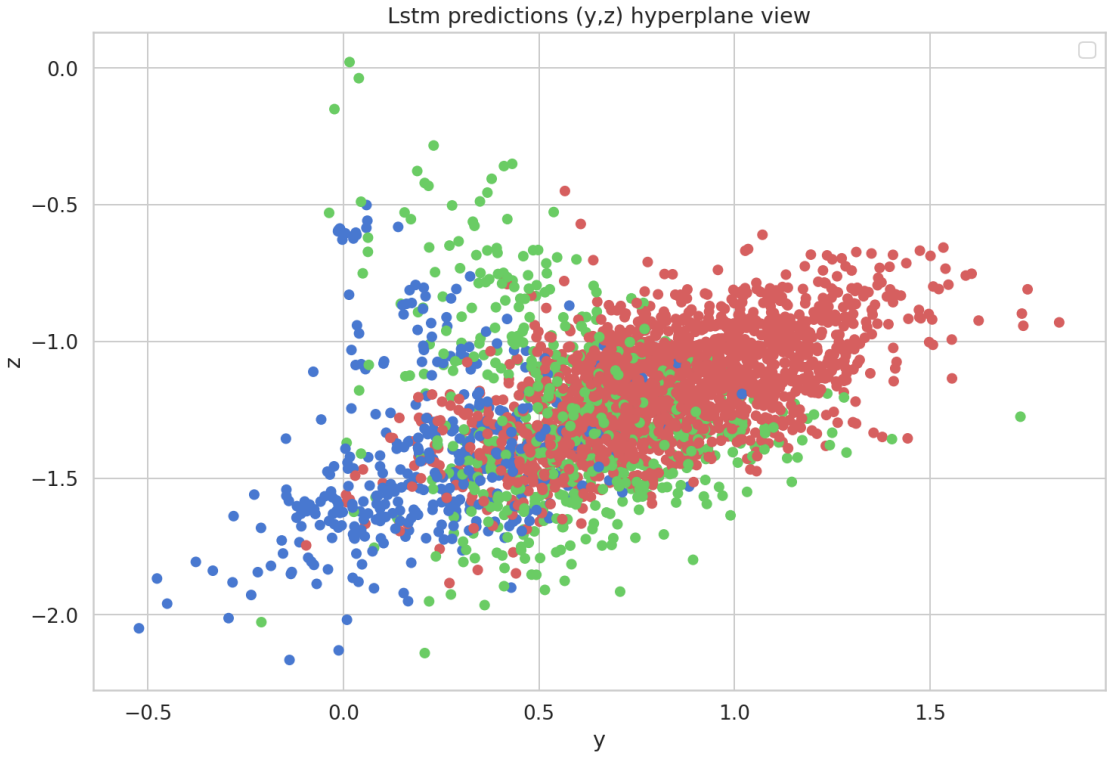
[99]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.title("Lstm Predictions (x,y) hyperplane view")
plt.scatter(df_pca_test[0], df_pca_test[1],␣
 ↪c=colormap[df_pca_test['lstm_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



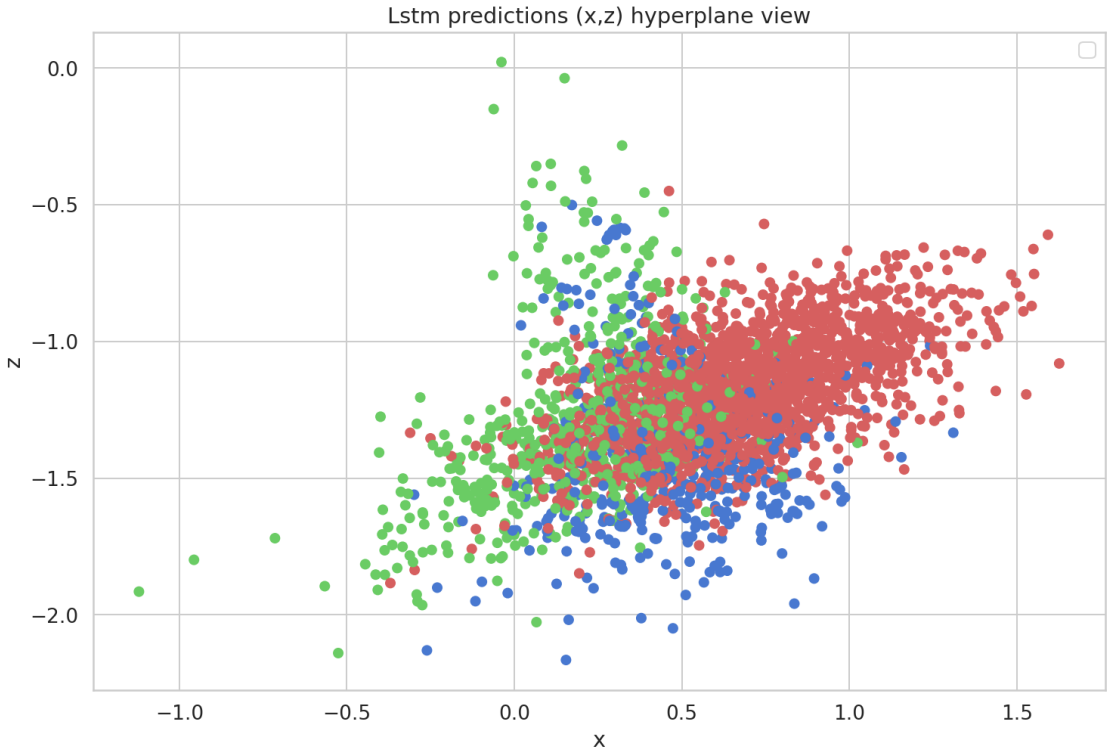
[100]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("y")
plt.ylabel("z")
plt.legend()
plt.title("Lstm predictions (y,z) hyperplane view")
plt.scatter(df_pca_test[1], df_pca_test[2],␣
 ↪c=colormap[df_pca_test['lstm_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



[101]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("x")
plt.ylabel("z")
plt.legend()
plt.title("Lstm predictions (x,z) hyperplane view")
plt.scatter(df_pca_test[0], df_pca_test[2],␣
 ↪c=colormap[df_pca_test['lstm_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



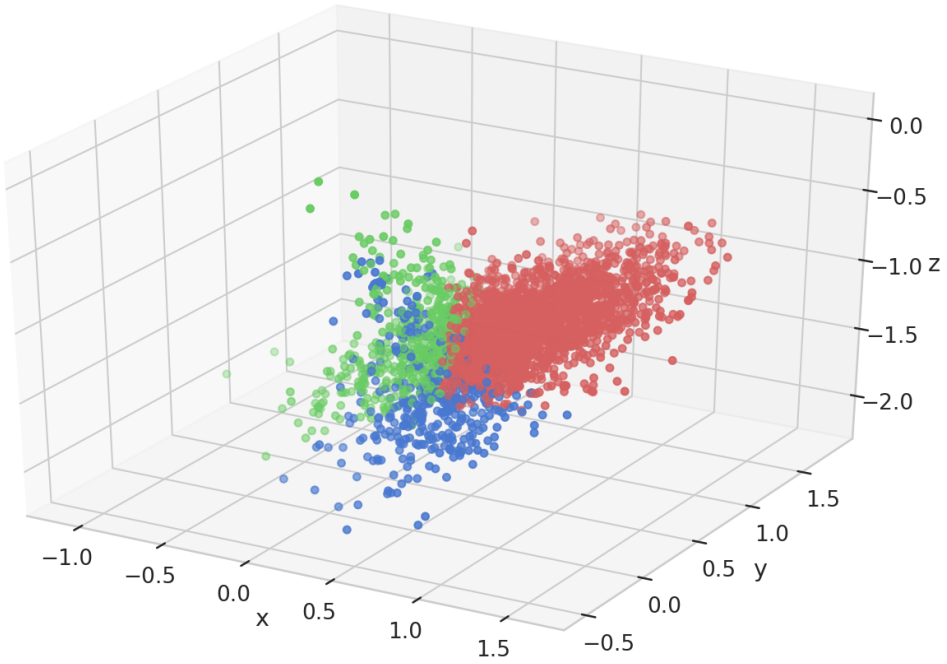
3 - Logreg model

[102]:
```
df_pca_test['logreg_y_pred'] = pd.DataFrame(logreg_label_pred)
```

[103]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.legend()
ax.grid(True)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.scatter(df_pca_test[0].to_list(), df_pca_test[1].to_list(),df_pca_test[2].
 ↪to_list(),c=colormap[df_pca_test['logreg_y_pred']])
```

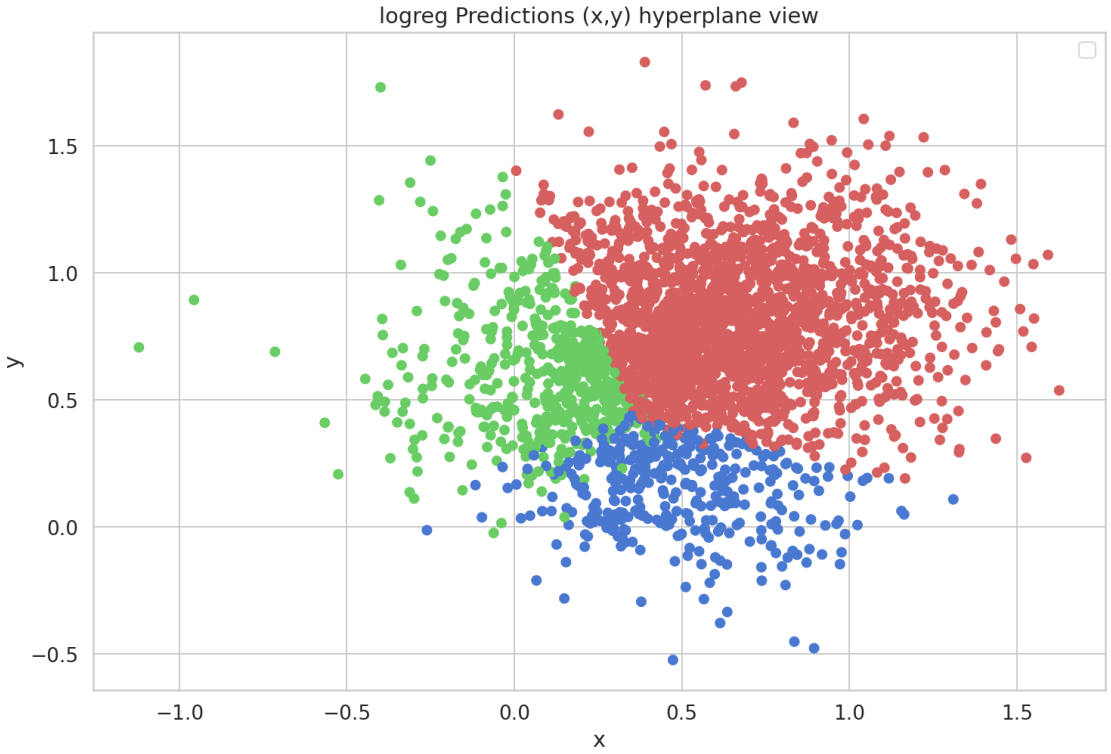
```
No handles with labels found to put in legend.
```

[103]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7fc2ecc4d908>
```



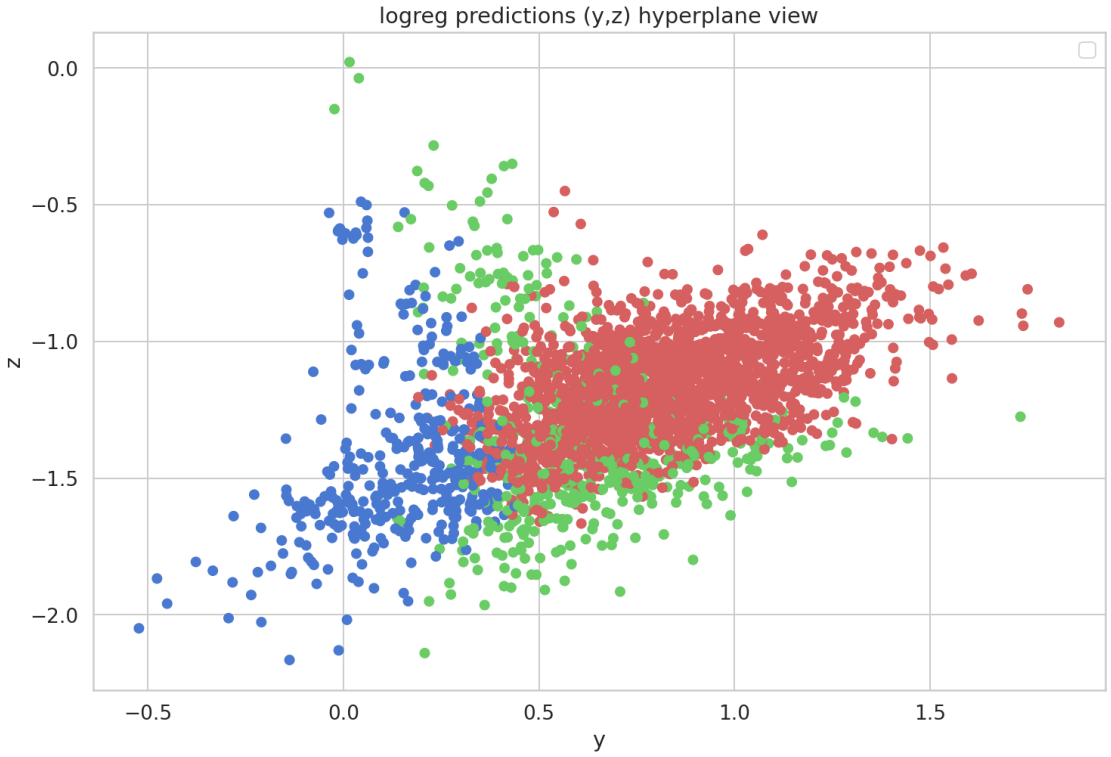
[104]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.title("logreg Predictions (x,y) hyperplane view")
plt.scatter(df_pca_test[0], df_pca_test[1],␣
 ↪c=colormap[df_pca_test['logreg_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



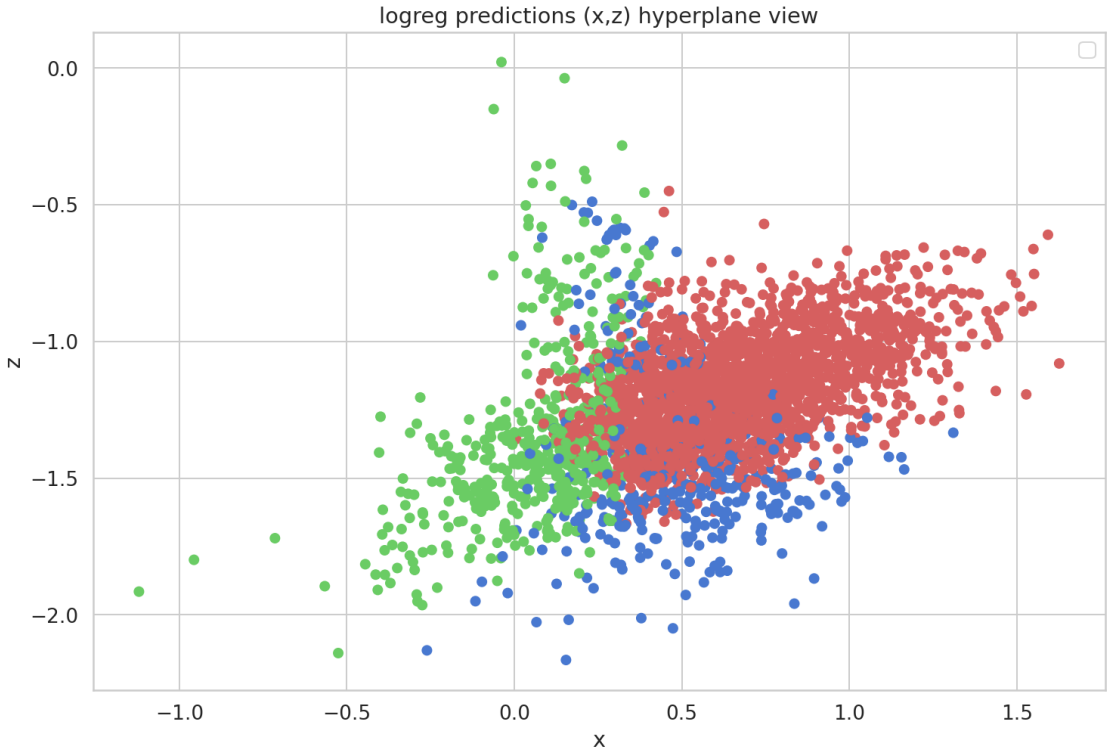
[105]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("y")
plt.ylabel("z")
plt.legend()
plt.title("logreg predictions (y,z) hyperplane view")
plt.scatter(df_pca_test[1], df_pca_test[2],␣
 ↪c=colormap[df_pca_test['logreg_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



[106]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("x")
plt.ylabel("z")
plt.legend()
plt.title("logreg predictions (x,z) hyperplane view")
plt.scatter(df_pca_test[0], df_pca_test[2],␣
 ↪c=colormap[df_pca_test['logreg_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



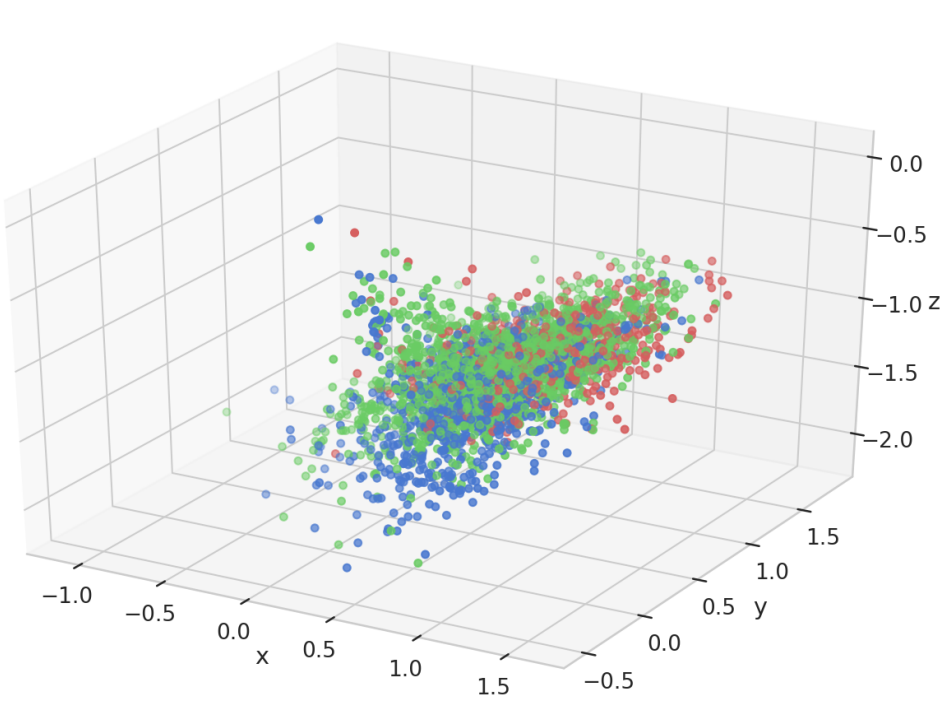
4 - Textblob

[107]:
```
df_pca_test['textblob_y_pred'] = pd.DataFrame(textblob_label_pred)
```

[108]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.legend()
ax.grid(True)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.scatter(df_pca_test[0].to_list(), df_pca_test[1].to_list(),df_pca_test[2].
 ↪to_list(),c=colormap[df_pca_test['textblob_y_pred']])
```

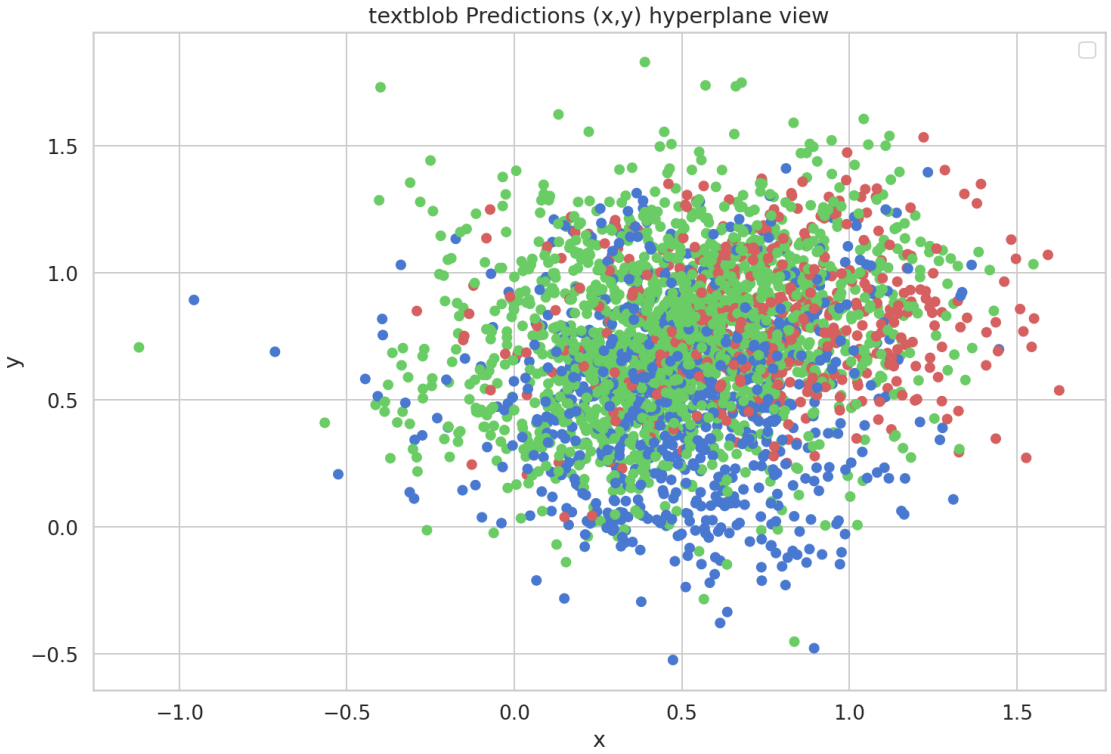
```
No handles with labels found to put in legend.
```

[108]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7fc2ecb04b00>
```



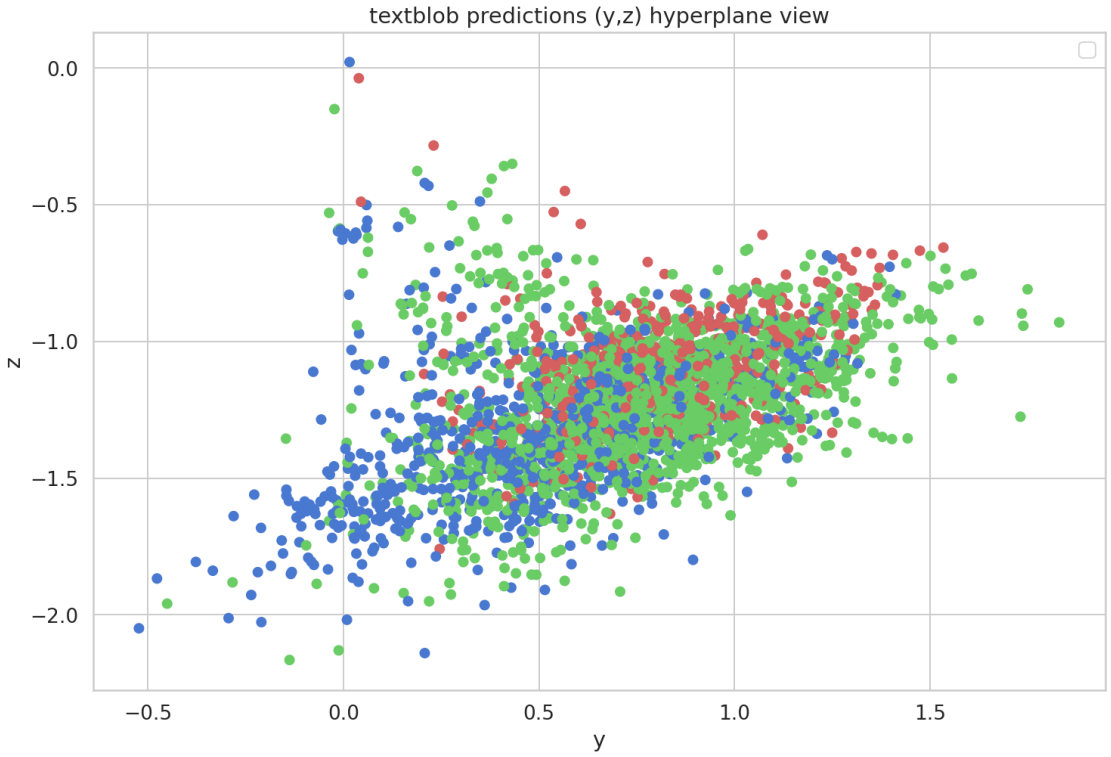
[109]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.title("textblob Predictions (x,y) hyperplane view")
plt.scatter(df_pca_test[0], df_pca_test[1],␣
 ↪c=colormap[df_pca_test['textblob_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



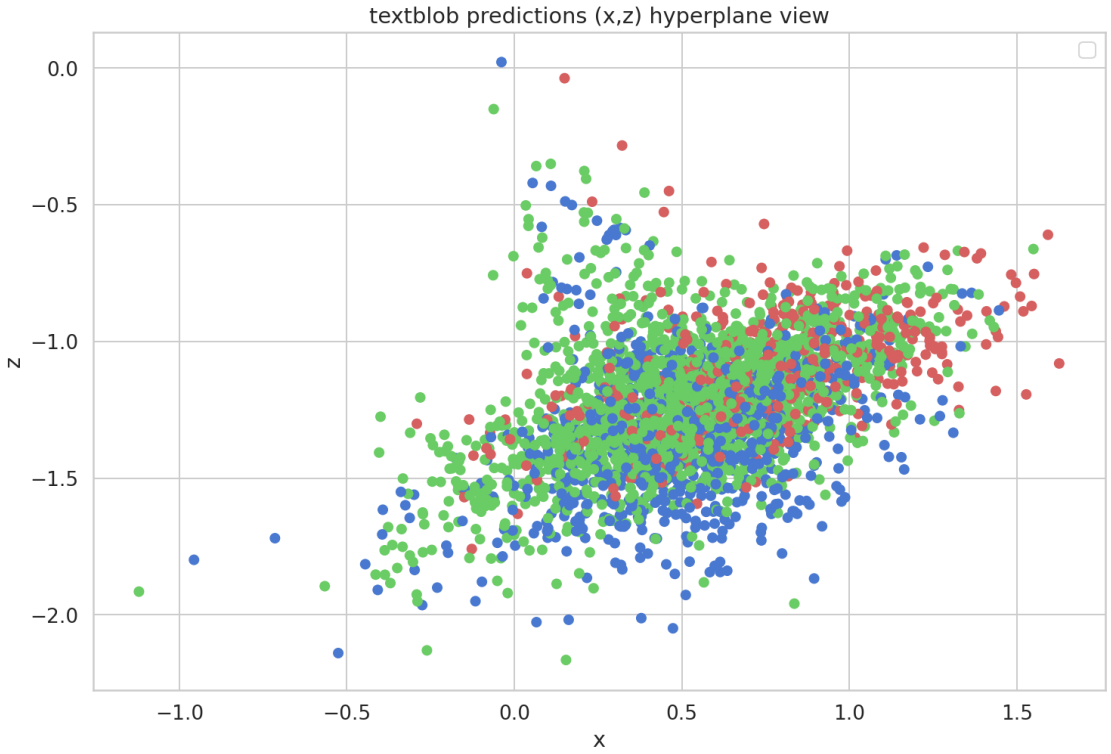
[110]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("y")
plt.ylabel("z")
plt.legend()
plt.title("textblob predictions (y,z) hyperplane view")
plt.scatter(df_pca_test[1], df_pca_test[2],␣
 ↪c=colormap[df_pca_test['textblob_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



[111]:
```
colormap = np.array(['r', 'g', 'b'])
# Red -> negative, Green -> neutral, Blue -> positive
plt.xlabel("x")
plt.ylabel("z")
plt.legend()
plt.title("textblob predictions (x,z) hyperplane view")
plt.scatter(df_pca_test[0], df_pca_test[2],␣
 ↪c=colormap[df_pca_test['textblob_y_pred']])
plt.show()
```

```
No handles with labels found to put in legend.
```



The result show that….

[116]: