Sentiment Analysis Project

August 6, 2023

# 1 Sentiment analysis project

## 1.1 Dataset

In this notebook, I will be working on a dataset of Amazon baby product customer reviews and ratings. Feel free to download the dataset from this lien.

The dataset is composed of two features :

- **Sentence** : customer opinions on products (customer feelings about products)
- **label** : the ratings (0 means negative feeling and 1 means positive feeling)

[1]:
```
#Important librairies
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

[2]:
```
# Read dataset
df = pd.read_csv("amazon_cells_labelled.txt", names =["sentence", "label"], sep␣
↪= "\t")
df.head()
```

[2]:
```
                                            sentence  label
0  So there is no way for me to plug it in here i…      0
1                        Good case, Excellent value.      1
2                             Great for the jawbone.      1
3  Tied to charger for conversations lasting more…      0
4                                  The mic is great.      1
```

[3]:
```
#Count the classes of the label
df["label"].value_counts()
```

[3]:
```
0    500
1    500
Name: label, dtype: int64
```

[69]:
```
#dataset details
df.info()
```

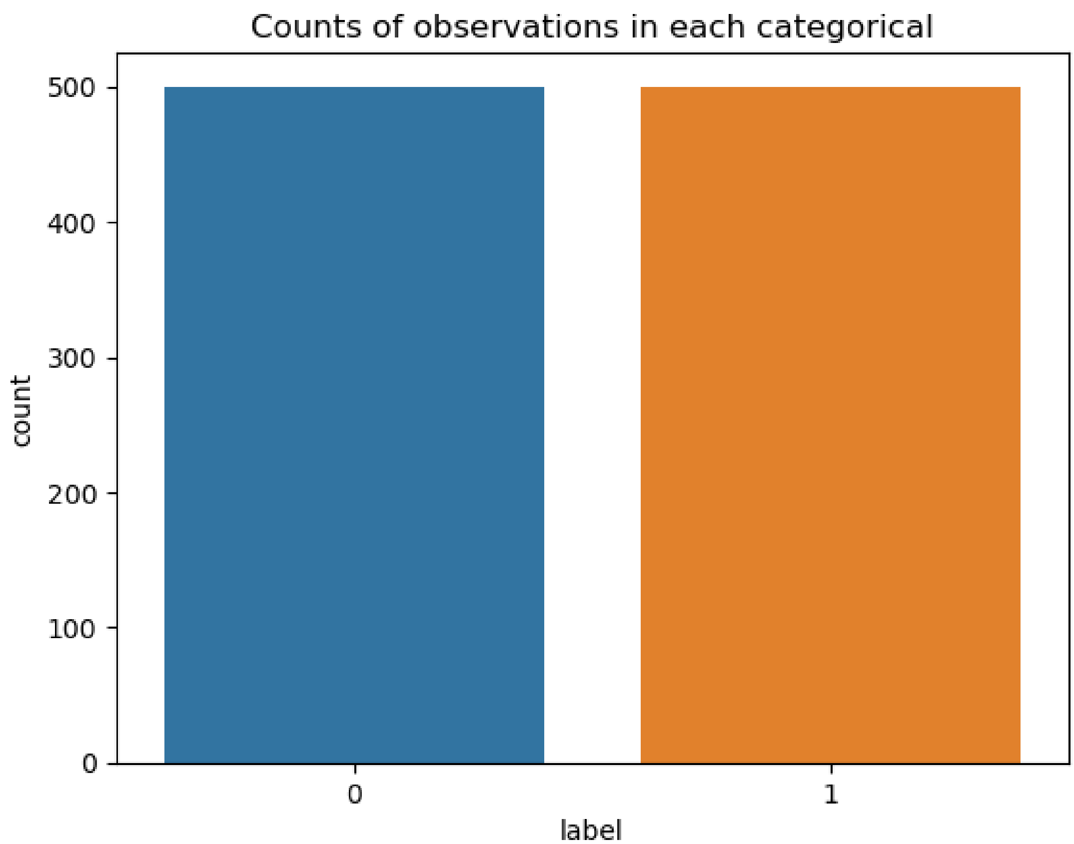
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   sentence  1000 non-null   object
 1   label     1000 non-null   int64
dtypes: int64(1), object(1)
memory usage: 15.8+ KB
```

[70]:
```
#Dataset shape
df.shape
```

[70]:
```
(1000, 2)
```

[44]:
```
#Countplot of the label
ax =plt.axes()
sns.countplot(x = "label", data =df)
ax.set_title('Counts of observations in each categorical')
```

[44]:
```
Text(0.5, 1.0, 'Counts of observations in each categorical')
```




We find that the classes are balanced. This is a good sign for model prediction.

### 1.2 Data processing

[5]:
```
# Separate label from features
X = df.drop("label", axis =1)
y = df["label"]
```

[6]:
```
#Split the dataset into training set and test setet
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
      X, y, test_size=0.30, random_state=101)
```

[7]:
```
X_train.shape
```

[7]: (700, 1)

[8]:
```
X_test.shape
```

[8]: (300, 1)

### 1.2.1 Tokenization : Numeric transformation of text

As we know, when we deal with text data, we have to convert it to numbers before feeding any machine learning model including neural networks. Tokenization then consists of cutting a text composed of several paragraphs into small units, also called tokens. Tokenization is the first step in many natural language processing projects, since it is the basis for developing good models and helping to better understand the text we have.

We use the tensorflow library to transform our text into numeric values:

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

But be aware that there are several libraries capable of transforming text into numeric values such as the following libraries: :

**CountVectorizer** : from sklearn.feature_extraction.text import CountVectorizer

**word_tokenize** : from nltk.tokenize import word_tokenize

**Tokenizer** : from spacy.tokenizer import Tokenizer

[9]:
```
#Add the text to a list to make it easier to use with Tokenizer
X_train_sentences =[]
for i in X_train["sentence"] :
    X_train_sentences.append(i)

print("Show first 2 rows of training set :\n\n", X_train_sentences[:2], "\n")

X_test_sentences = []

for j in X_test["sentence"] :
    X_test_sentences.append(j)

print("Show first 2 rows of test set :\n\n", X_test_sentences[:2])
```

```
Show first 2 rows of training set :

 ['Great Phone.', 'The look of it is very sharp and the screen is nice and
clear, with great graphics.']

Show first 2 rows of test set :

 ["No shifting, no bubbling, no peeling, not even a scratch, NOTHING!I couldn't
be more happier with my new one for the Droid.", 'Customer service was
terrible.']
```

[10]:
```
#Important librairies
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

[11]:
```
#important settings for Tokenizer and Embedding Layer
vocab_size = 500
max_length = 100
trunc_type = 'post'
oov_tok = '<OOV>'
padding_type = 'post'
```

- **vocab_size** = 500 : means that we will take 500 unique words to form the tokinzer.
- **Max_length** = 100 : represents the length of each review. If the original review is longer than 150 words, it will be truncated.
- **trunc_type** = **'post'** : means the review will be truncated at the end when a review is longer than 100 words.
- **padding_type** = **'post'** : means the padding will be applied at the end, not at the beginning.

Tokenize words now

[93]:
```
#Initialize the tokenizer

tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_tok)

#Fit the tokenizer
tokenizer.fit_on_texts(X_train_sentences)
word_index = tokenizer.word_index

#Convert text to numeric values
training_sequences = tokenizer.texts_to_sequences(X_train_sentences)
testing_sequences = tokenizer.texts_to_sequences(X_test_sentences)


print("The first 25 words of the word index dictionary with their number␣
 ↪indices in the text\n\n", list(word_index.items())[:25], "\n\n")

print("The training set showing the first 5 sentences transformed into␣
 ↪numerical values\n\n", training_sequences[:5], "\n")

print("The testing set showing the first 5 sentences transformed into numerical␣
 ↪values\n\n", testing_sequences[:5])
```

```
The first 25 words of the word index dictionary with their number indices in the
text

 [('<OOV>', 1), ('the', 2), ('i', 3), ('and', 4), ('it', 5), ('is', 6), ('a',
7), ('this', 8), ('to', 9), ('phone', 10), ('my', 11), ('of', 12), ('for', 13),
('not', 14), ('on', 15), ('great', 16), ('with', 17), ('very', 18), ('was', 19),
('that', 20), ('in', 21), ('good', 22), ('have', 23), ('you', 24), ('works',
25)]
```

```
The training set showing the first 5 sentences transformed into numerical values

 [[16, 10], [2, 205, 12, 5, 6, 18, 302, 4, 2, 179, 6, 88, 4, 130, 17, 16, 410],
[100, 14, 411, 412], [3, 79, 2, 251, 80, 116, 38, 36, 1], [47, 1, 413, 32]]

The testing set showing the first 5 sentences transformed into numerical values

 [[72, 1, 72, 1, 72, 1, 14, 73, 7, 1, 343, 3, 162, 58, 78, 1, 17, 11, 149, 40,
13, 2, 1], [161, 55, 19, 151], [3, 1, 8, 13, 2, 120, 89, 4, 5, 65, 14, 59], [3,
51, 46, 8, 279, 489], [14, 201]]
```

Sentences are now represented as a sequence of words and words are converted to numeric values

### 1.2.2 Padding

[13]:
```python
# Padding
training_padded = pad_sequences(training_sequences, maxlen=max_length, padding
 ↪= padding_type, truncating=trunc_type)  #
testing_padded = pad_sequences(testing_sequences, maxlen=max_length, padding =
 ↪padding_type, truncating=trunc_type)  # , maxlen=max_length

print("Show first 3 rows of training set :\n\n", X_train_sentences[:2], "\n")
print("Show first 3 rows of training sequences :\n\n", training_sequences[:2],
 ↪"\n")
print("Show first 3 rows of training padded :\n\n", training_padded[:2], "\n\n")


print("Show first 3 rows of test set :\n\n", X_test_sentences[:2])
print("\nShow first 3 rows of testing sequences :\n\n", testing_sequences[:2],
 ↪"\n")
print("\nShow first 3 rows of testing padded:\n\n", testing_padded[:2])
```

```
Show first 3 rows of training set :

 ['Great Phone.', 'The look of it is very sharp and the screen is nice and
clear, with great graphics.']

Show first 3 rows of training sequences :

 [[16, 10], [2, 205, 12, 5, 6, 18, 302, 4, 2, 179, 6, 88, 4, 130, 17, 16, 410]]

Show first 3 rows of training padded :

 [[ 16  10   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

```
   0   0   0   0   0   0   0   0   0   0]
 [  2 205  12   5   6  18 302   4   2 179   6  88   4 130  17  16 410   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]]


Show first 3 rows of test set :

 ["No shifting, no bubbling, no peeling, not even a scratch, NOTHING!I couldn't
be more happier with my new one for the Droid.", 'Customer service was
terrible.']

Show first 3 rows of testing sequences :

 [[72, 1, 72, 1, 72, 1, 14, 73, 7, 1, 343, 3, 162, 58, 78, 1, 17, 11, 149, 40,
13, 2, 1], [161, 55, 19, 151]]


Show first 3 rows of testing padded:

 [[ 72   1  72   1  72   1  14  73   7   1 343   3 162  58  78   1  17  11
  149  40  13   2   1   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [161  55  19 151   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]]
```

Once the text processing is finished, we will now build our model

## 2 Model

[14]:
```python
# Important librairies
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Embedding, GlobalMaxPool1D
```

**1. Embedding Layer**

An embedding is a dense vector of floating point values. Vector length is usually set to 8 dimensions for small datasets (up to 1024 dimensions). On the other hand, when working with large datasets, a large dimension will be able to capture fine relationships between words. Link

Three useful arguments for building an integration layer :

- **input_dim** : vocabulary size in text data (vocab_size)
- **output_dim** : means that each word will be represented by a 16-dimensional vector
- **input_length** : length of input sequences.

**2. Flatten Layer**

Convert multidimensional arrays to one-dimensional arrays.

**3. Dense Layer**

A dense layer is a layer deeply connected to its previous layer, which means that each neuron receives input from all neurons in the previous layer. This layer is the most commonly used layer in neural networks.

Two parameters are important in this dense layer:

- **Units** : Units are one of the most basic and necessary parameters of Keras dense layer which defines the size of dense layer output. It must be a positive integer since it represents the dimensionality of the output vector.

- **activation function** : In neural networks, the activation function is a function used for the transformation of neuron input values. Basically, it introduces nonlinearity in neural network networks so that networks can learn the relationship between input and output values. If in this Keras layer no activation is defined, it will consider the linear activation function. there are many activation functions among which we have the function **Relu**, **Sigmoid** and **Softmax**. For more details on the activation functions used in Keras, click ici

[15]:
```
#Instantiate the model Sequential
model = Sequential()
#Add the layers to model
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

### 2.0.1 Compilation model

Compiling a model is necessary to finalize the model and make it completely ready for use. For compilation, an optimizer and a loss function are needed.

- **optimiseur**: We use the **Adam** optimizer in this notebook. However, there are different optimizers like RMSprop, SGD, Adam…

- **loss** : There are several loss functions, but in general for a binary classification problem one uses **binary_crossentropy**

- **metrics** : We add the **accuracy** metric on which the model will be scored

[16]:
```
# compile the model
```

```
model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

[17]:
```
#stop early to prevent overfitting
from tensorflow.keras.callbacks import EarlyStopping

early_stop =EarlyStopping(monitor='val_loss', patience = 2)
```

### 2.0.2 Training model

Before training the model, we just need to convert the labels to the array. If you notice, they are in list form:

[18]:
```
training_labels_final = np.array(y_train)
testing_labels_final = np.array(y_test)
```

[19]:
```
#fit the model
model.fit(training_padded, training_labels_final, epochs = 100,␣
 ↪validation_data=(testing_padded, testing_labels_final),␣
 ↪callbacks=[early_stop])
```

```
Epoch 1/100
22/22 [==============================] - 2s 24ms/step - loss: 0.6919 - accuracy:
0.5286 - val_loss: 0.7025 - val_accuracy: 0.4300
Epoch 2/100
22/22 [==============================] - 0s 7ms/step - loss: 0.6866 - accuracy:
0.5314 - val_loss: 0.7002 - val_accuracy: 0.4333
Epoch 3/100
22/22 [==============================] - 0s 7ms/step - loss: 0.6787 - accuracy:
0.6500 - val_loss: 0.6952 - val_accuracy: 0.4833
Epoch 4/100
22/22 [==============================] - 0s 7ms/step - loss: 0.6639 - accuracy:
0.6429 - val_loss: 0.6813 - val_accuracy: 0.5733
Epoch 5/100
22/22 [==============================] - 0s 7ms/step - loss: 0.6374 - accuracy:
0.7743 - val_loss: 0.6849 - val_accuracy: 0.5167
Epoch 6/100
22/22 [==============================] - 0s 7ms/step - loss: 0.5956 - accuracy:
0.7486 - val_loss: 0.6454 - val_accuracy: 0.6367
Epoch 7/100
22/22 [==============================] - 0s 7ms/step - loss: 0.5449 - accuracy:
0.8100 - val_loss: 0.6324 - val_accuracy: 0.6167
Epoch 8/100
22/22 [==============================] - 0s 7ms/step - loss: 0.4791 - accuracy:
0.8557 - val_loss: 0.6006 - val_accuracy: 0.6667
Epoch 9/100
22/22 [==============================] - 0s 7ms/step - loss: 0.4141 - accuracy:
0.8957 - val_loss: 0.5640 - val_accuracy: 0.7433
```

```
Epoch 10/100
22/22 [==============================] - 0s 8ms/step - loss: 0.3556 - accuracy:
0.9114 - val_loss: 0.5515 - val_accuracy: 0.7267
Epoch 11/100
22/22 [==============================] - 0s 8ms/step - loss: 0.3032 - accuracy:
0.9286 - val_loss: 0.5472 - val_accuracy: 0.6933
Epoch 12/100
22/22 [==============================] - 0s 7ms/step - loss: 0.2618 - accuracy:
0.9457 - val_loss: 0.5469 - val_accuracy: 0.6867
Epoch 13/100
22/22 [==============================] - 0s 6ms/step - loss: 0.2295 - accuracy:
0.9357 - val_loss: 0.5359 - val_accuracy: 0.6967
Epoch 14/100
22/22 [==============================] - 0s 7ms/step - loss: 0.1955 - accuracy:
0.9586 - val_loss: 0.5273 - val_accuracy: 0.7167
Epoch 15/100
22/22 [==============================] - 0s 8ms/step - loss: 0.1699 - accuracy:
0.9643 - val_loss: 0.5284 - val_accuracy: 0.7167
Epoch 16/100
22/22 [==============================] - 0s 7ms/step - loss: 0.1484 - accuracy:
0.9714 - val_loss: 0.5274 - val_accuracy: 0.7133
```

[19]: <keras.callbacks.History at 0x18674fe88e0>

[20]:
```
# summarize the model
print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 100, 8)            4000

 flatten (Flatten)           (None, 800)               0

 dense (Dense)               (None, 10)                8010

 dense_1 (Dense)             (None, 1)                 11

=================================================================
Total params: 12,021
Trainable params: 12,021
Non-trainable params: 0
_________________________________________________________________
None
```

[22]:
```
# model history
data_history = pd.DataFrame(model.history.history)
```

```python
print(data_history.head())
```

```
       loss  accuracy  val_loss  val_accuracy
0  0.691922  0.528571  0.702542      0.430000
1  0.686560  0.531429  0.700183      0.433333
2  0.678711  0.650000  0.695243      0.483333
3  0.663940  0.642857  0.681334      0.573333
4  0.637445  0.774286  0.684925      0.516667
```

[36]:
```python
round(data_history[["loss", "val_loss"]].min(), 2)
```

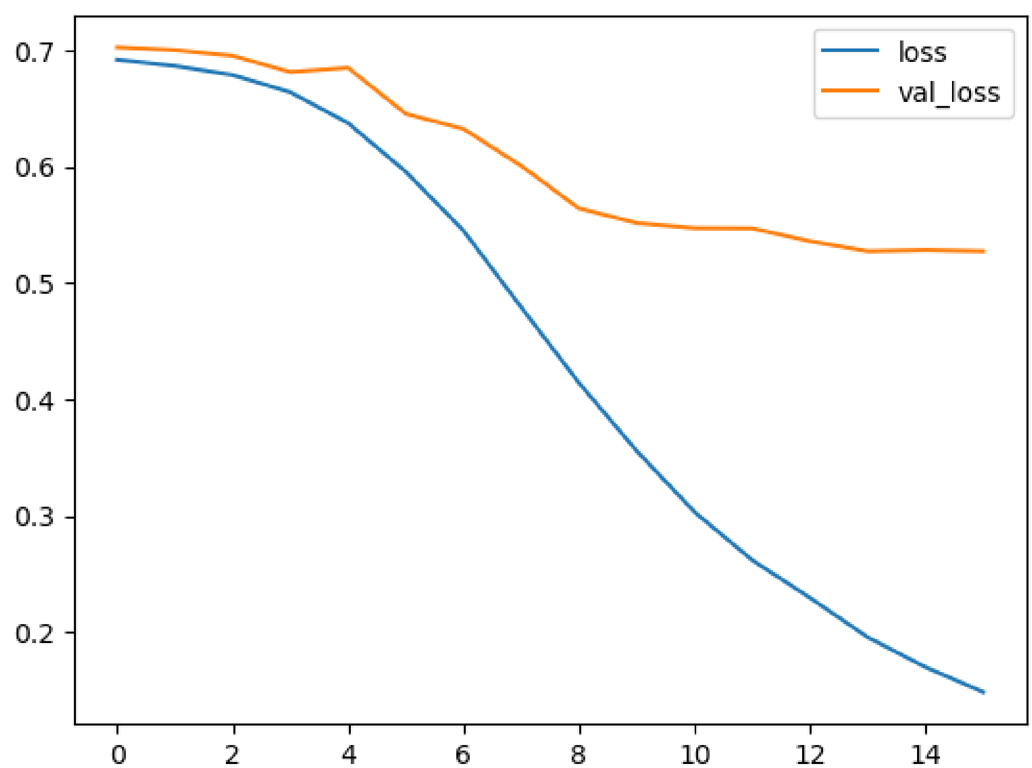
[36]:
```
loss        0.15
val_loss    0.53
dtype: float64
```

[37]:
```python
round(data_history[["accuracy", "val_accuracy"]].max(), 2)
```

[37]:
```
accuracy        0.97
val_accuracy    0.74
dtype: float64
```

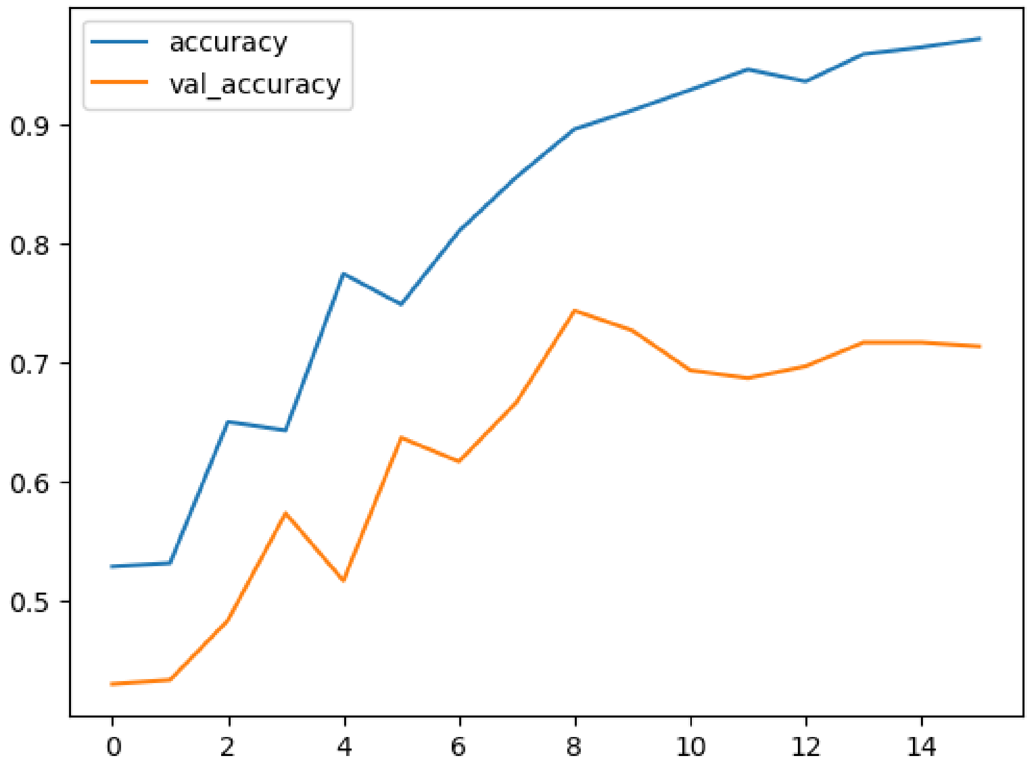
[25]:
```python
data_history[["loss", "val_loss"]].plot()
```

[25]: `<Axes: >`



[26]:
```
data_history[["accuracy", "val_accuracy"]].plot()
```

[26]: <Axes: >




Another way to work with embeddings is to use a **MaxPooling1D/AveragePooling1D** or a **GlobalMaxPooling1D/GlobalAveragePooling1D** layer after embedding.

[27]:
```
embedding_dim = 8

Model = Sequential()
Model.add(Embedding(input_dim=vocab_size,
                           output_dim=embedding_dim,
                           input_length=max_length))
Model.add(GlobalMaxPool1D())
Model.add(Dense(10, activation='relu'))
Model.add(Dense(1, activation='sigmoid'))
Model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 100, 8)            4000

 flatten (Flatten)           (None, 800)               0

 dense (Dense)               (None, 10)                8010

 dense_1 (Dense)             (None, 1)                 11

=================================================================
Total params: 12,021
Trainable params: 12,021
Non-trainable params: 0
_________________________________________________________________
```

[28]:

```
#fit the model
Model.fit(training_padded, training_labels_final, epochs = 100,␣
 ↪validation_data=(testing_padded, testing_labels_final),␣
 ↪callbacks=[early_stop])
```

```
Epoch 1/100
22/22 [==============================] - 2s 22ms/step - loss: 0.6919 - accuracy:
0.5286 - val_loss: 0.6961 - val_accuracy: 0.4300
Epoch 2/100
22/22 [==============================] - 0s 6ms/step - loss: 0.6888 - accuracy:
0.5300 - val_loss: 0.6957 - val_accuracy: 0.4300
Epoch 3/100
22/22 [==============================] - 0s 7ms/step - loss: 0.6852 - accuracy:
0.5300 - val_loss: 0.6950 - val_accuracy: 0.4300
Epoch 4/100
22/22 [==============================] - 0s 7ms/step - loss: 0.6793 - accuracy:
0.5486 - val_loss: 0.6902 - val_accuracy: 0.4400
Epoch 5/100
22/22 [==============================] - 0s 7ms/step - loss: 0.6698 - accuracy:
0.5943 - val_loss: 0.6833 - val_accuracy: 0.4600
Epoch 6/100
22/22 [==============================] - 0s 7ms/step - loss: 0.6561 - accuracy:
0.6586 - val_loss: 0.6743 - val_accuracy: 0.5600
Epoch 7/100
22/22 [==============================] - 0s 8ms/step - loss: 0.6391 - accuracy:
0.7314 - val_loss: 0.6637 - val_accuracy: 0.6233
Epoch 8/100
22/22 [==============================] - 0s 7ms/step - loss: 0.6156 - accuracy:
0.7900 - val_loss: 0.6436 - val_accuracy: 0.7100
Epoch 9/100
```

```
22/22 [==============================] - 0s 7ms/step - loss: 0.5838 - accuracy:
0.8571 - val_loss: 0.6181 - val_accuracy: 0.7333
Epoch 10/100
22/22 [==============================] - 0s 6ms/step - loss: 0.5467 - accuracy:
0.8629 - val_loss: 0.5899 - val_accuracy: 0.7567
Epoch 11/100
22/22 [==============================] - 0s 6ms/step - loss: 0.5043 - accuracy:
0.8671 - val_loss: 0.5599 - val_accuracy: 0.7767
Epoch 12/100
22/22 [==============================] - 0s 7ms/step - loss: 0.4611 - accuracy:
0.8743 - val_loss: 0.5315 - val_accuracy: 0.7700
Epoch 13/100
22/22 [==============================] - 0s 6ms/step - loss: 0.4176 - accuracy:
0.8857 - val_loss: 0.5039 - val_accuracy: 0.7767
Epoch 14/100
22/22 [==============================] - 0s 6ms/step - loss: 0.3777 - accuracy:
0.8943 - val_loss: 0.4806 - val_accuracy: 0.7800
Epoch 15/100
22/22 [==============================] - 0s 7ms/step - loss: 0.3411 - accuracy:
0.9014 - val_loss: 0.4573 - val_accuracy: 0.7867
Epoch 16/100
22/22 [==============================] - 0s 6ms/step - loss: 0.3067 - accuracy:
0.9143 - val_loss: 0.4426 - val_accuracy: 0.7800
Epoch 17/100
22/22 [==============================] - 0s 6ms/step - loss: 0.2771 - accuracy:
0.9186 - val_loss: 0.4279 - val_accuracy: 0.7900
Epoch 18/100
22/22 [==============================] - 0s 7ms/step - loss: 0.2513 - accuracy:
0.9257 - val_loss: 0.4200 - val_accuracy: 0.7967
Epoch 19/100
22/22 [==============================] - 0s 6ms/step - loss: 0.2292 - accuracy:
0.9343 - val_loss: 0.4115 - val_accuracy: 0.7967
Epoch 20/100
22/22 [==============================] - 0s 7ms/step - loss: 0.2103 - accuracy:
0.9414 - val_loss: 0.4066 - val_accuracy: 0.8000
Epoch 21/100
22/22 [==============================] - 0s 7ms/step - loss: 0.1939 - accuracy:
0.9414 - val_loss: 0.4036 - val_accuracy: 0.8000
Epoch 22/100
22/22 [==============================] - 0s 6ms/step - loss: 0.1792 - accuracy:
0.9471 - val_loss: 0.4023 - val_accuracy: 0.8000
Epoch 23/100
22/22 [==============================] - 0s 6ms/step - loss: 0.1659 - accuracy:
0.9514 - val_loss: 0.4030 - val_accuracy: 0.8000
Epoch 24/100
22/22 [==============================] - 0s 7ms/step - loss: 0.1540 - accuracy:
0.9557 - val_loss: 0.4045 - val_accuracy: 0.8133
```

[28]: <keras.callbacks.History at 0x1867677ea90>

[29]:
```
data_histories = pd.DataFrame(Model.history.history)
print(data_histories.head())
```

```
       loss  accuracy  val_loss  val_accuracy
0  0.691881  0.528571  0.696115          0.43
1  0.688812  0.530000  0.695677          0.43
2  0.685214  0.530000  0.694951          0.43
3  0.679298  0.548571  0.690177          0.44
4  0.669802  0.594286  0.683302          0.46
```

[35]:
```
round(data_histories[["loss", "val_loss"]].min(), 2)
```

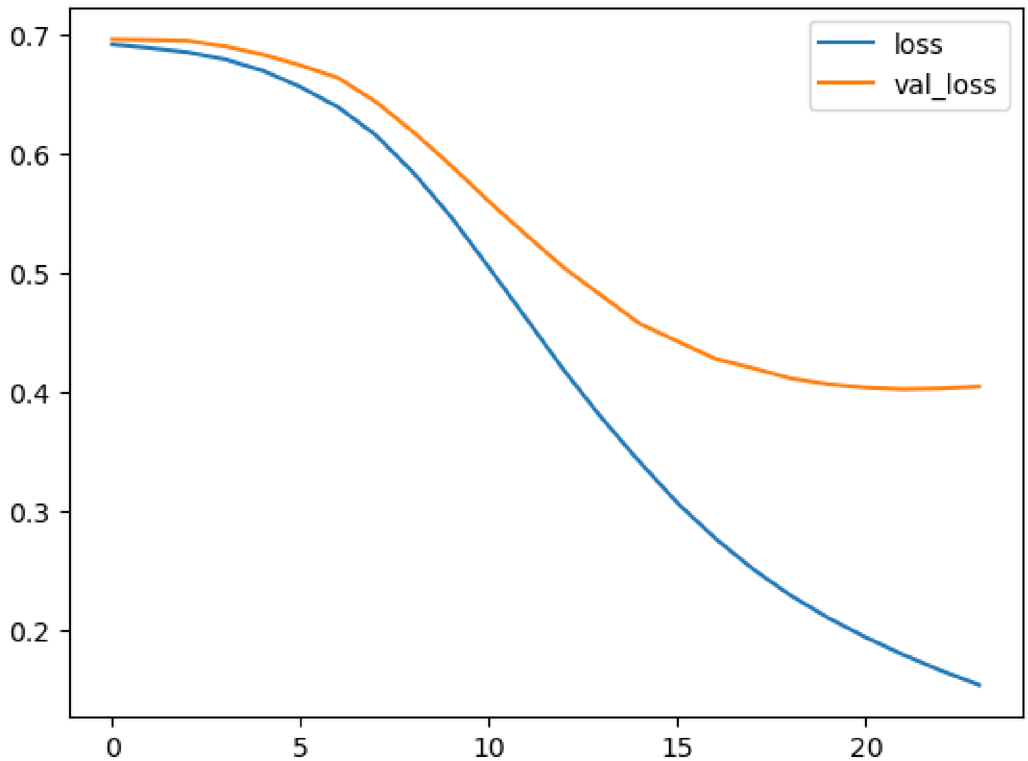
[35]:
```
loss        0.15
val_loss    0.40
dtype: float64
```

[34]:
```
round(data_histories[["accuracy", "val_accuracy"]].max(), 2)
```

[34]:
```
accuracy        0.96
val_accuracy    0.81
dtype: float64
```

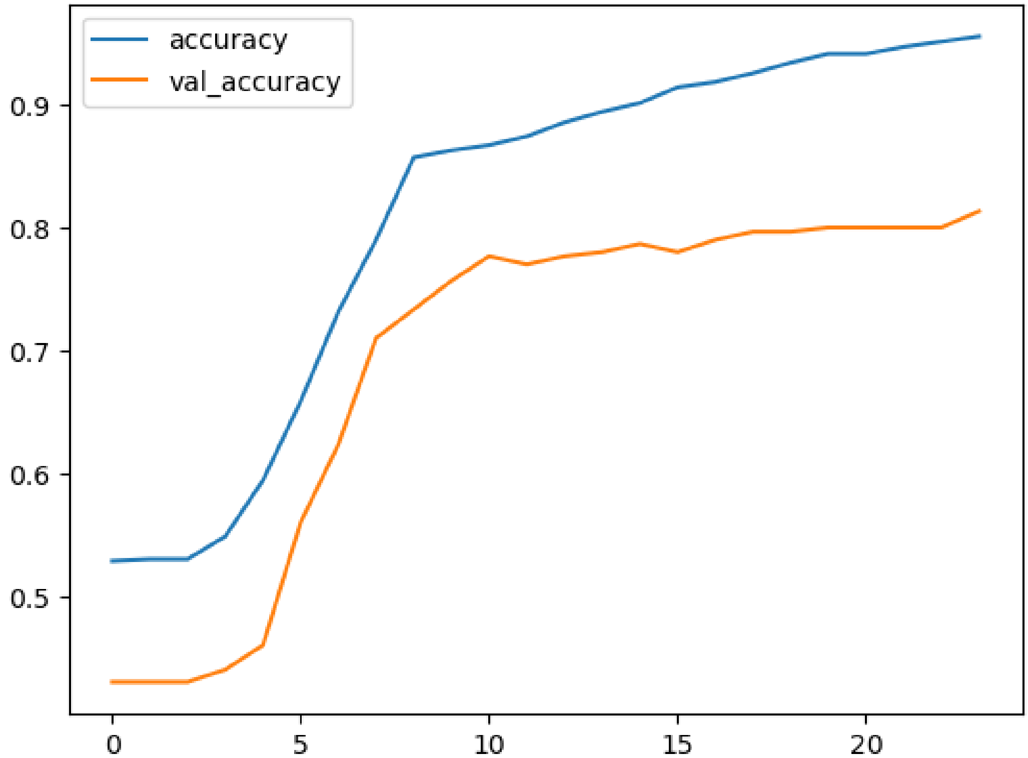
[32]:
```
data_histories[["loss", "val_loss"]].plot()
```

[32]: <Axes: >




[33]:
```
data_histories[["accuracy", "val_accuracy"]].plot()
```

[33]: <Axes: >




### 2.0.3 Model evaluation with GlobalMaxPool1D flayer

[82]:
```
predictions = (Model.predict(testing_padded) > 0.5).astype("int32")
```

```
10/10 [==============================] - 0s 2ms/step
```

[85]:
```
# https://en.wikipedia.org/wiki/Precision_and_recall
print(classification_report(testing_labels_final,predictions))
```

```
              precision    recall  f1-score   support

           0       0.84      0.83      0.84       171
           1       0.78      0.79      0.78       129

    accuracy                           0.81       300
   macro avg       0.81      0.81      0.81       300
weighted avg       0.81      0.81      0.81       300
```

[86]:
```
#Confusion Matrix
print(confusion_matrix(testing_labels_final,predictions))
```

```
[[142  29]
```

```
[ 27 102]]
```

#### 2.0.4 Results of two models

[40]:
```python
dict = {"accuracy " :  [0.96, 0.97], "val_accuracy" : [0.81, 0.74], "loss":[0.
↪15, 0.15], "val_loss": [0.40, 0.53]}
```

[43]:
```python
pd.DataFrame(dict, index = ["Model with GlobalMaxPool1D flayer", "Model with␣
↪Flatten flayer"])
```

[43]:

| | accuracy | val_accuracy | loss | val_loss |
|---|---|---|---|---|
| Model with GlobalMaxPool1D flayer | 0.96 | 0.81 | 0.15 | 0.40 |
| Model with Flatten flayer | 0.97 | 0.74 | 0.15 | 0.53 |

# 3 Conclusion

As you have seen, it is very easy to perform sentiment analysis with the Tensorflow and Keras libraries. The part that takes a lot of time is word processing. And we also find that our models have an overfitting problem, but the model with a GlobalMaxPool1D layer decreases the overfitting problem a bit.We can also use a dropout layer to see if we can decrease the overfitting. In short, whether in Machine Learning or Deep Learning, you have to use several models and play with hyperparameters to see if the models do better. But sometimes you have to make sure that the data is confirmed and increase the amount of data so that the models can capture the most information.

# 4 references

https://towardsdatascience.com/a-complete-step-by-step-tutorial-on-sentiment-analysis-in-keras-and-tensorflow-ea420cc8913f

https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/

https://machinelearningmastery.com/what-are-word-embeddings/

https://python.plainenglish.io/python-word2vec-for-text-classification-with-lstm-d9e63e84f6ee

https://www.tensorflow.org/text/guide/word_embeddings?hl=fr