Computational Intelligence Lab

Assignment 6

Karolina Kotlowska IO czw. 9:30

6.1 Load IMDB reviews dataset

In [3]:
```
import tensorflow_datasets as tfds
import tensorflow as tf

ds_train = tfds.load('imdb_reviews', split='train', as_supervised=True, s
ds_test = tfds.load('imdb_reviews', split='test', as_supervised=True, shu
```

```
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-p
ackages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/
user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

Examine the content - we prefer texts and label values to tensors

In [4]:
```
import pandas as pd

df_train = pd.DataFrame(ds_train.take(10))
df_train.head()
```

```
2023-04-15 12:24:22.676530: W tensorflow/tsl/platform/profile_utils/cpu_
utils.cc:128] Failed to get CPU frequency: 0 Hz
2023-04-15 12:24:22.693977: W tensorflow/core/kernels/data/cache_dataset
_ops.cc:856] The calling iterator did not fully read the dataset being c
ached. In order to avoid unexpected truncation of the dataset, the parti
ally cached contents of the dataset  will be discarded. This can happen
if you have an input pipeline similar to `dataset.cache().take(k).repeat
()`. You should use `dataset.take(k).cache().repeat()` instead.
```

Out[4]:

	0	1
0	tf.Tensor(b"This was an absolutely terrible mo...	tf.Tensor(0, shape=(), dtype=int64)
1	tf.Tensor(b'I have been known to fall asleep d...	tf.Tensor(0, shape=(), dtype=int64)
2	tf.Tensor(b'Mann photographs the Alberta Rocky...	tf.Tensor(0, shape=(), dtype=int64)
3	tf.Tensor(b'This is the kind of film for a sno...	tf.Tensor(1, shape=(), dtype=int64)
4	tf.Tensor(b'As others have mentioned, all the ...	tf.Tensor(1, shape=(), dtype=int64)

In [5]:
```
data = [(text.numpy().decode('UTF8'),label.numpy()) for text,label in ds_
df_train = pd.DataFrame(data,columns=['text','label'])
df_train.head(len(df_train))
```

Out[5]:

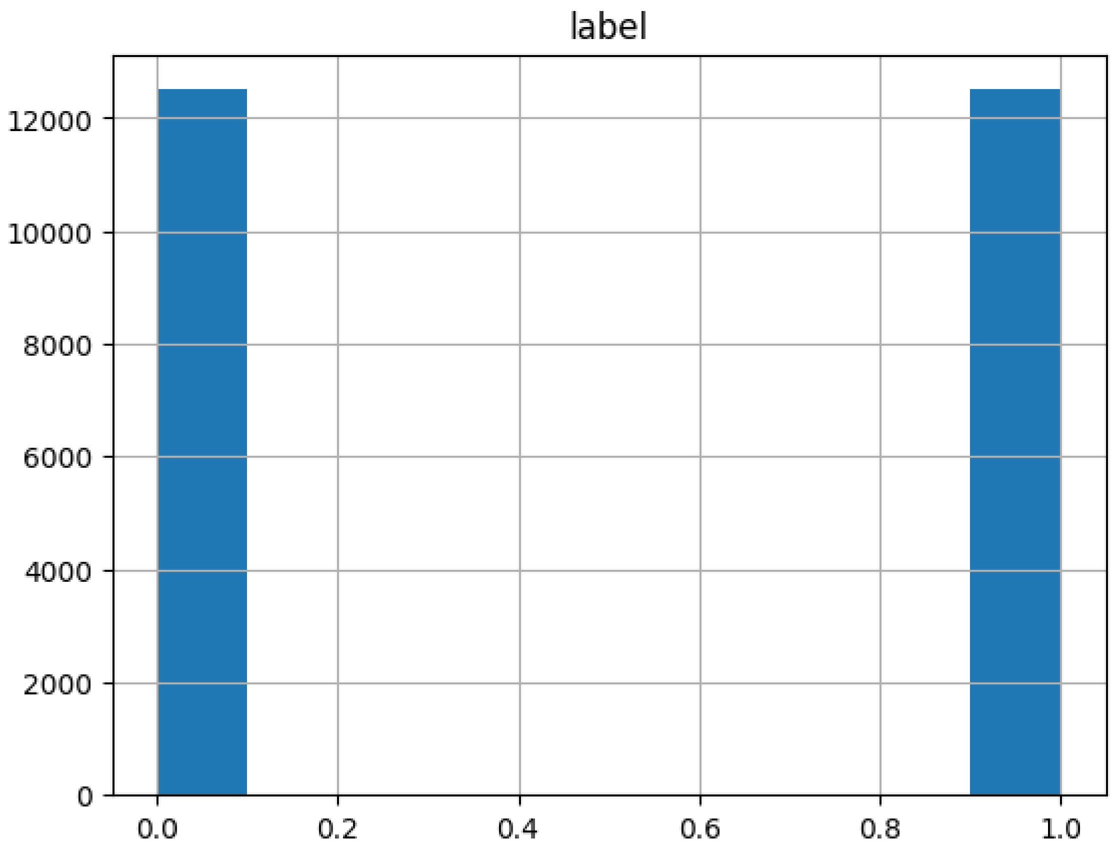
	text	label
0	This was an absolutely terrible movie. Don't b...	0
1	I have been known to fall asleep during films,...	0
2	Mann photographs the Alberta Rocky Mountains i...	0
3	This is the kind of film for a snowy Sunday af...	1
4	As others have mentioned, all the women that g...	1
...	...	...
24995	I have a severe problem with this show, severa...	0
24996	The year is 1964. Ernesto "Che" Guevara, havin...	1
24997	Okay. So I just got back. Before I start my re...	0
24998	When I saw this trailer on TV I was surprised....	0
24999	First of all, Riget is wonderful. Good comedy ...	1

25000 rows × 2 columns

How many labels and what is the class distribution?

In [6]:
```
df_train.hist()
```

Out[6]:
```
array([[<Axes: title={'center': 'label'}>]], dtype=object)
```



In [7]:
```
data = [(text.numpy().decode('UTF8'),label.numpy()) for text,label in ds_
df_test = pd.DataFrame(data,columns=['text','label'])
df_test.head(len(df_test))
```

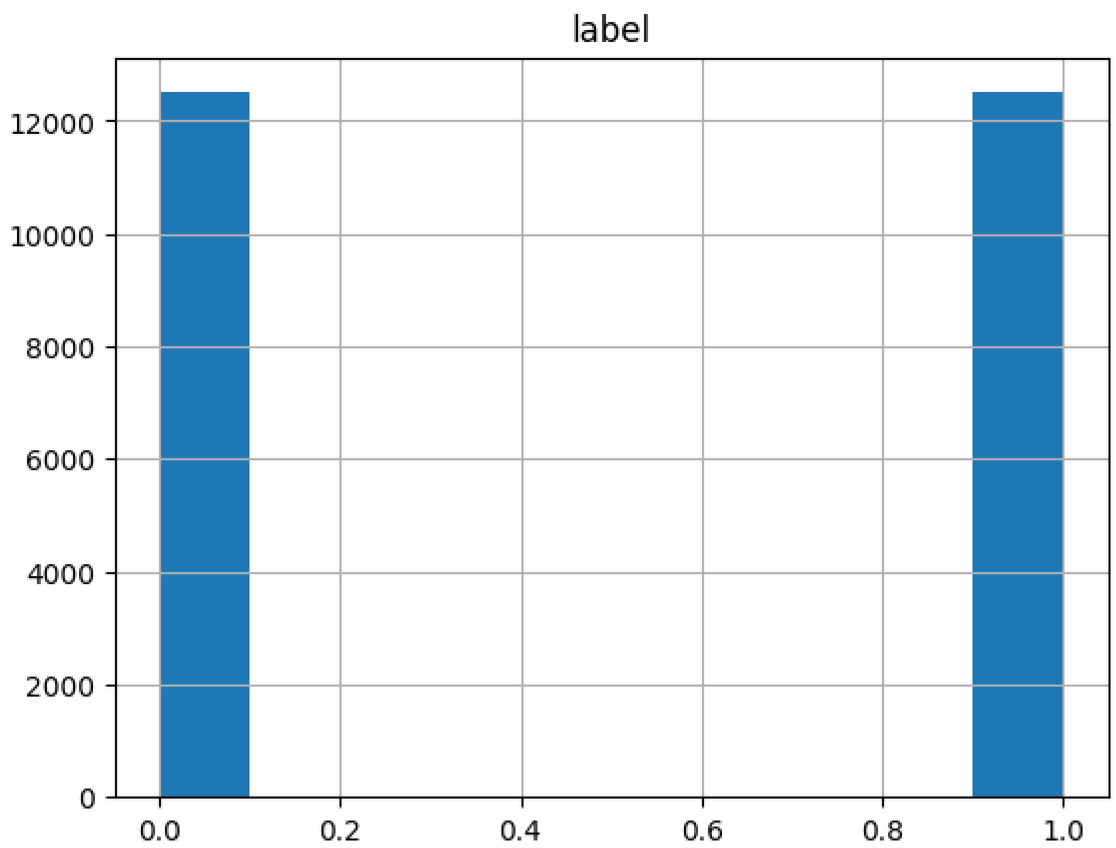
Out[7]:

	text	label
0	There are films that make careers. For George ...	1
1	A blackly comic tale of a down-trodden priest,...	1
2	Scary Movie 1-4, Epic Movie, Date Movie, Meet ...	0
3	Poor Shirley MacLaine tries hard to lend some ...	0
4	As a former Erasmus student I enjoyed this fil...	1
...	...	...
24995	Feeling Minnesota is not really a road movie, ...	0
24996	This is, without doubt, one of my favourite ho...	1
24997	Most predicable movie I've ever seen...extreme...	0
24998	It's exactly what I expected from it. Relaxing...	1
24999	They just don't make cartoons like they used t...	1

25000 rows × 2 columns

In [8]:
```
df_test.hist()
```

Out[8]:
```
array([[<Axes: title={'center': 'label'}>]], dtype=object)
```



6.2 Text preprocessing

Text preprocessing involves

- cleaning
- extracting tokens (words, bigrams, trigrams, sometimes character based tokens)
- counting tokens, converting documents to the bag-of-words representation
- for frequency based representations: scaling
- padding sequences in the case of neural networks

CountVectorizer

In [9]:
```
from sklearn.feature_extraction.text import CountVectorizer,TfidfVectoriz
from sklearn.feature_extraction.text import TfidfTransformer
import numpy as np

input = """Incy Wincy spider went up the water spout
Down came the rain and washed the spider out
Out came the sun and dried up all the rain
And the Incy Wincy spider went up the spout again
Incy Wincy spider went up the water spout
Down came the rain and washed the spider out
Out came the sun and dried up all the rain
And the Incy Wincy spider went up the spout again"""
corpus=input.split('\n')
print(corpus)

vectorizer = CountVectorizer()
vectorizer.fit(corpus)
X = vectorizer.transform(corpus)
```

```
['Incy Wincy spider went up the water spout', 'Down came the rain and wa
shed the spider out', 'Out came the sun and dried up all the rain', 'And
the Incy Wincy spider went up the spout again', 'Incy Wincy spider went
up the water spout', 'Down came the rain and washed the spider out', 'Ou
t came the sun and dried up all the rain', 'And the Incy Wincy spider we
nt up the spout again']
```

What is X? Looks like a sparse matrix...

In [10]:
```
print(X.shape)
print(X)
```

```
(8, 18)
  (0, 6)	1
  (0, 9)	1
  (0, 10)	1
  (0, 12)	1
  (0, 13)	1
  (0, 15)	1
  (0, 16)	1
  (0, 17)	1
  (1, 2)	1
  (1, 3)	1
  (1, 4)	1
  (1, 7)	1
  (1, 8)	1
  (1, 9)	1
  (1, 12)	2
  (1, 14)	1
  (2, 1)	1
  (2, 2)	1
  (2, 3)	1
  (2, 5)	1
  (2, 7)	1
  (2, 8)	1
  (2, 11)	1
  (2, 12)	2
  (2, 13)	1
  :	:
  (5, 3)	1
  (5, 4)	1
  (5, 7)	1
  (5, 8)	1
  (5, 9)	1
  (5, 12)	2
  (5, 14)	1
  (6, 1)	1
  (6, 2)	1
  (6, 3)	1
  (6, 5)	1
  (6, 7)	1
  (6, 8)	1
  (6, 11)	1
  (6, 12)	2
  (6, 13)	1
  (7, 0)	1
  (7, 2)	1
  (7, 6)	1
  (7, 9)	1
  (7, 10)	1
  (7, 12)	2
  (7, 13)	1
  (7, 16)	1
  (7, 17)	1
```

In [11]:

```
Y=X.toarray()
print(Y)
print(vectorizer.vocabulary_)
```

```
[[0 0 0 0 0 0 1 0 0 1 1 0 1 1 0 1 1 1]
 [0 0 1 1 1 0 0 1 1 1 0 0 2 0 1 0 0 0]
 [0 1 1 1 0 1 0 1 1 0 0 1 2 1 0 0 0 0]
 [1 0 1 0 0 0 1 0 0 1 1 0 2 1 0 0 1 1]
 [0 0 0 0 0 0 1 0 0 1 1 0 1 1 0 1 1 1]
 [0 0 1 1 1 0 0 1 1 1 0 0 2 0 1 0 0 0]
 [0 1 1 1 0 1 0 1 1 0 0 1 2 1 0 0 0 0]
 [1 0 1 0 0 0 1 0 0 1 1 0 2 1 0 0 1 1]]
{'incy': 6, 'wincy': 17, 'spider': 9, 'went': 16, 'up': 13, 'the': 12,
'water': 15, 'spout': 10, 'down': 4, 'came': 3, 'rain': 8, 'and': 2, 'wa
shed': 14, 'out': 7, 'sun': 11, 'dried': 5, 'all': 1, 'again': 0}
```

TF-IDF conversion

Tfidf - term frequency inverse document frequency assignes smaller weights to terms appering often in a set of documents. See Wikipedia

$$tf(t,d) = \frac{\#occurences\ of\ t\ in\ d}{sum\ of\ all\ term\ occurences}$$

$$idf(t,D) = log(\frac{number\ of\ documents}{number\ of\ documents\ containing\ term\ t})$$

$$tfidf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

In [12]:
```python
transformer = TfidfTransformer()
Z = transformer.fit_transform(X).toarray()
print(Z)
```

```
[[0.         0.         0.         0.         0.         0.
  0.36796129 0.         0.         0.2899856  0.36796129 0.
  0.23174479 0.2899856  0.         0.48634247 0.36796129 0.36796129]
 [0.         0.         0.25810931 0.32751362 0.43288191 0.
  0.         0.32751362 0.32751362 0.25810931 0.         0.
  0.41254109 0.         0.43288191 0.         0.         0.        ]
 [0.         0.39725862 0.23686864 0.30056145 0.         0.39725862
  0.         0.30056145 0.30056145 0.         0.         0.39725862
  0.37859173 0.23686864 0.         0.         0.         0.        ]
 [0.43583403 0.         0.25986954 0.         0.         0.
  0.32974717 0.         0.         0.25986954 0.32974717 0.
  0.41535451 0.25986954 0.         0.         0.32974717 0.32974717]
 [0.         0.         0.         0.         0.         0.
  0.36796129 0.         0.         0.2899856  0.36796129 0.
  0.23174479 0.2899856  0.         0.48634247 0.36796129 0.36796129]
 [0.         0.         0.25810931 0.32751362 0.43288191 0.
  0.         0.32751362 0.32751362 0.25810931 0.         0.
  0.41254109 0.         0.43288191 0.         0.         0.        ]
 [0.         0.39725862 0.23686864 0.30056145 0.         0.39725862
  0.         0.30056145 0.30056145 0.         0.         0.39725862
  0.37859173 0.23686864 0.         0.         0.         0.        ]
 [0.43583403 0.         0.25986954 0.         0.         0.
  0.32974717 0.         0.         0.25986954 0.32974717 0.
  0.41535451 0.25986954 0.         0.         0.32974717 0.32974717]]
```

In [13]:
```python
inv_vocab = {v: k for k, v in vectorizer.vocabulary_.items()}
vocabulary = [inv_vocab[i] for i in range(len(inv_vocab))]
```

TODO 6.2.1 Print the words with the smallest (but nonzero) tfidf values (in each row of Z)

In [14]:
```
# this is how you inverse the dictionary
inv_vocab = {v: k for k, v in vectorizer.vocabulary_.items()}

for i in range (Z.shape[0]):
  min_value = 1
  argmin=-1
  for j in range(Z.shape[1]):
    if(Z[i,j]!=0 and Z[i,j]<min_value):
      min_value=Z[i,j]
      argmin = j
  print(i, vocabulary[argmin])
```

```
0 the
1 and
2 and
3 and
4 the
5 and
6 and
7 and
```

6.3 Classification with MultinomialNB

MultinomialNB (Multinomial Naive Bayes) is a baseline classifier for all text related tasks. See Wikipedia

Convert texts to tokens (in one step using TfidfVectorizer) and check the shapes of resulting matrices

In [15]:
```
vectorizer = TfidfVectorizer(max_features=10_000)
vectorizer.fit(df_train.text)
X_train = vectorizer.transform(df_train.text)
X_test = vectorizer.transform(df_test.text)
```

Please note - the same vectorizer configuration, which was fit to the training data, is applied to transform test data.

In [16]:
```
print(X_train.shape)
print(X_test.shape)
```

```
(25000, 10000)
(25000, 10000)
```

Train and test classifier

In [17]:
```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import confusion_matrix, classification_report

cls = MultinomialNB()

cls.fit(X_train,df_train.label)
```

```
y_pred = cls.predict(X_test)
proba = cls.predict_proba(X_test)
print(classification_report(df_test.label,y_pred))

print(confusion_matrix(y_pred,df_test.label))
```

```
              precision    recall  f1-score   support

           0       0.81      0.87      0.84     12500
           1       0.86      0.80      0.83     12500

    accuracy                           0.84     25000
   macro avg       0.84      0.84      0.84     25000
weighted avg       0.84      0.84      0.84     25000

[[10936  2524]
 [ 1564  9976]]
```

Pipeline

Very often data processing can be considered a sequence of steps. In this case a pipeline can be built.

We improve slighly processing by including english stopwords. See Wikipedia

In [18]:
```
from nltk.corpus import stopwords
import nltk
nltk.download('stopwords')
stopwords = stopwords.words('english')
print(stopwords)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "yo
u're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourse
lves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'he
rself', 'it', "it's", 'its', 'itself', 'they', 'them', 'their', 'their
s', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "tha
t'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been',
'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing',
'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'w
hile', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between',
'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to',
'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'agai
n', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why',
'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'so
me', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 't
oo', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't", 'should',
"should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren',
"aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'h
adn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'm
a', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'sha
n', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "were
n't", 'won', "won't", 'wouldn', "wouldn't"]
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/spoton/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [19]:
```
from sklearn.pipeline import Pipeline
```

```
model = Pipeline([('vectorizer', TfidfVectorizer(max_features=10_000,ngra
                  ('classifier',MultinomialNB())])

model.fit(df_train.text,df_train.label)
```

Out[19]:

```
▸ Pipeline
▸ TfidfVectorizer
▸ MultinomialNB
```

TODO 6.3.1 Check if stopwords were removed. The vectorizer vocabulary can be accessed as `model['vectorizer'].vocabulary_`

In [20]:

```
for sw in stopwords:
    cnt = model['vectorizer'].vocabulary_
```

Now - predict labels and probabilities and compute scores

In [21]:

```
y_pred = model.predict(df_test.text)
proba = model.predict_proba(df_test.text)
print(classification_report(df_test.label,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.82      0.87      0.84     12500
           1       0.86      0.80      0.83     12500

    accuracy                           0.84     25000
   macro avg       0.84      0.84      0.84     25000
weighted avg       0.84      0.84      0.84     25000
```

Explain classifier decisions with lime

Let us test on an example from a training set

In [22]:

```
from lime import lime_text
i = 0
txt_instance = df_train.text[i]
print(txt_instance)
# ## check true value and predicted value
y_pred = model.predict([txt_instance])
proba = model.predict_proba([txt_instance])

print("True:", df_train.label[i], "--> Pred:", y_pred[i], "| Prob:", roun
# ## show explanation
explainer = lime_text.LimeTextExplainer(class_names=["Negative","Positive
explained = explainer.explain_instance(txt_instance,  model.predict_proba
explained.show_in_notebook(text=txt_instance, predict_proba=False)
```

```
This was an absolutely terrible movie. Don't be lured in by Christopher
Walken or Michael Ironside. Both are great actors, but this must simply
be their worst role in history. Even their great acting could not redeem
this movie's ridiculous storyline. This movie is an early nineties US pr
opaganda piece. The most pathetic scenes were those when the Columbian r
ebels were making their cases for revolutions. Maria Conchita Alonso app
eared phony, and her pseudo-love affair with Walken was nothing but a pa
thetic emotional plug in a movie that was devoid of any real meaning. I
am disappointed that there are movies like this, ruining actor's like Ch
ristopher Walken's good name. I could barely sit through it.
True: 0 --> Pred: 0 | Prob: 0.91
```

Text with highlighted words

This was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michael Ironside. Both are great actors, but this must simply be their worst role in history. Even their great acting could not redeem this movie's ridiculous storyline. This movie is an early nineties US propaganda piece. The most pathetic scenes were those when the Columbian rebels were making their cases for revolutions. Maria Conchita Alonso appeared phony, and her pseudo-love affair with Walken was nothing but a pathetic emotional plug in a movie that was devoid of any real meaning. I am disappointed that there are movies like this, ruining actor's like Christopher Walken's good name. I could barely sit through it.

We will define an utility function to reuse the code

In [23]:
```python
def explain(model,text_instance,true_label="Unknown"):
  y_pred = model.predict([text_instance])
  proba = model.predict_proba([text_instance])
  print(y_pred,proba)

  print("True:", true_label, "--> Pred:", y_pred[0], "| Prob:", round(np.
  ## show explanation
  explainer = lime_text.LimeTextExplainer(class_names=["Negative","Positi
  explanation = explainer.explain_instance(text_instance,  model.predict_
  explanation.show_in_notebook(text=text_instance, predict_proba=False)
```

TODO 6.3.2 Prepare 10 sentences and display their sentiment.

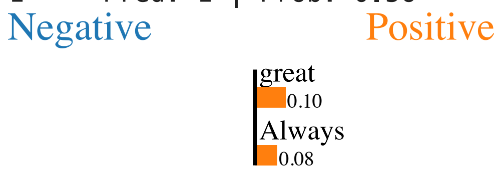
Note: the model will not recognize compound terms like *not bad* or *terribly sorry*, etc. To include bigrams into the dictionary you should probably set ngram_range to (1,2) and increase the number of features. `TfidfVectorizer(max_features=10_000,ngram_range=(1,2))`

In [24]:
```python
sentences = [
    "Always know you could be on the precipice of something great.",
    "Keep it fast, short and direct - whatever it is.",
    "You have to know when to call it quits and when to keep moving forwa
    "If you don't have problems, you're pretending or you don't run your 
    "Wishing a Happy Father's Day to all the Dad's out there - YOU are a 
    "Hear Donald Trump discuss big gov spending, banks, & taxes on Your W
    "The original Apprentice is coming back--do you have what it takes to
```

```
    "One of the other reviewers has mentioned that after watching just 1
    "I had the terrible misfortune of having to view this bad movice in i
    "Caddyshack Two is a good movie by itself but compared to the origina
]
```

In [25]:
```
for sentence in sentences:
    explain(model,sentence,1)
```

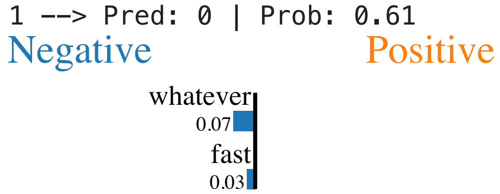
```
[1] [[0.43630531 0.56369469]]
True: 1 --> Pred: 1 | Prob: 0.56
```



Text with highlighted words

Always know you could be on the precipice of something great.

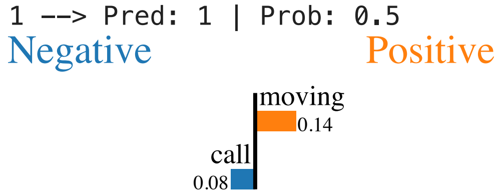
```
[0] [[0.60602402 0.39397598]]
True: 1 --> Pred: 0 | Prob: 0.61
```



Text with highlighted words

Keep it fast, short and direct - whatever it is.

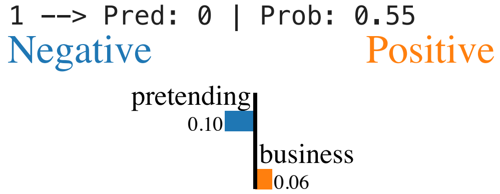
```
[1] [[0.49836339 0.50163661]]
True: 1 --> Pred: 1 | Prob: 0.5
```



Text with highlighted words

You have to know when to call it quits and when to keep moving forward.

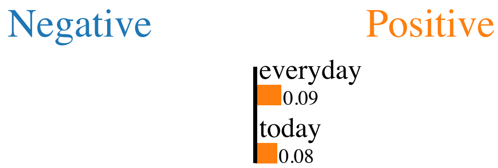
```
[0] [[0.55096397 0.44903603]]
True: 1 --> Pred: 0 | Prob: 0.55
```



Text with highlighted words

If you don't have problems, you're pretending or you don't run your own business.

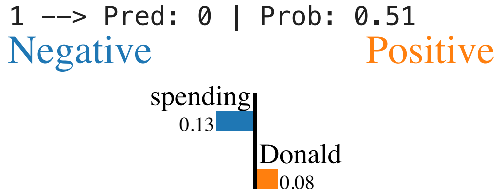
```
[1] [[0.26617492 0.73382508]]
True: 1 --> Pred: 1 | Prob: 0.73
```



Text with highlighted words

Wishing a Happy Father's Day to all the Dad's out there - YOU are a champion today and everyday!

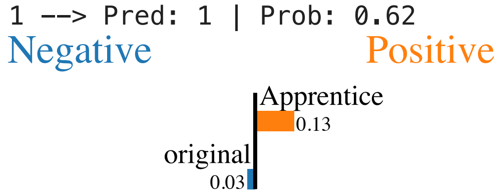
```
[0] [[0.50799509 0.49200491]]
True: 1 --> Pred: 0 | Prob: 0.51
```



Text with highlighted words

Hear Donald Trump discuss big gov spending, banks, | taxes on Your World

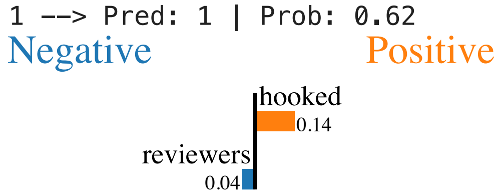
```
[1] [[0.37569647 0.62430353]]
True: 1 --> Pred: 1 | Prob: 0.62
```



Text with highlighted words

The original Apprentice is coming back--do you have what it takes to be the next Apprentice?

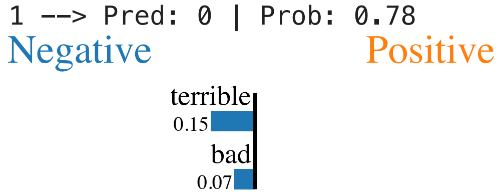
```
[1] [[0.37918233 0.62081767]]
True: 1 --> Pred: 1 | Prob: 0.62
```



Text with highlighted words

One of the other reviewers has mentioned that after watching just 1 Oz episode you'll be hooked. They are right, as this is exactly what happened with me.

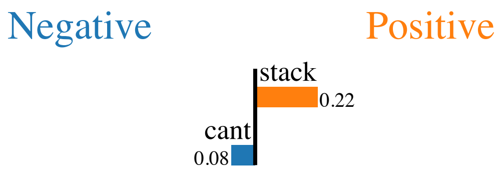
```
[0] [[0.78243706 0.21756294]]
True: 1 --> Pred: 0 | Prob: 0.78
```



Text with highlighted words

I had the terrible misfortune of having to view this bad movice in it's entirety.

```
[1] [[0.40796942 0.59203058]]
True: 1 --> Pred: 1 | Prob: 0.59
```



Text with highlighted words

Caddyshack Two is a good movie by itself but compared to the original it cant stack up.

6.4 Use embeddings

Embeddings are vector representations of words. Each word is represented by an n-dimensional vector, which captures its contexts. I reccommed this explanation.

Normally, embedding are learned in unsupervised manner. However, keras and TensorFlow offers an Embedding layer which is designed to be trained in supervised learning context.

Data preparation

Tokenization

The first step is tokenization. Each word occurence is replaced by an integer number. Tokenizer attribute `word_index` provides the mapping. Words are ordered by their frequency.

In [26]:
```
from keras.preprocessing.text import Tokenizer
tokenizer = Tokenizer(num_words=10_000)
tokenizer.fit_on_texts(df_train.text)
seq_train = tokenizer.texts_to_sequences(df_train.text)
seq_test = tokenizer.texts_to_sequences(df_test.text)
```

In [27]:
```
print(seq_train[0])
```

```
[11, 13, 32, 424, 391, 17, 89, 27, 8, 31, 1365, 3584, 39, 485, 196, 23,
84, 153, 18, 11, 212, 328, 27, 65, 246, 214, 8, 476, 57, 65, 84, 113, 9
7, 21, 5674, 11, 1321, 642, 766, 11, 17, 6, 32, 399, 8169, 175, 2454, 41
5, 1, 88, 1230, 136, 68, 145, 51, 1, 7576, 68, 228, 65, 2932, 15, 2903,
1478, 4939, 2, 38, 3899, 116, 1583, 16, 3584, 13, 161, 18, 3, 1230, 916,
7916, 8, 3, 17, 12, 13, 4138, 4, 98, 144, 1213, 10, 241, 682, 12, 47, 2
3, 99, 37, 11, 7180, 5514, 37, 1365, 49, 400, 10, 97, 1196, 866, 140, 9]
```

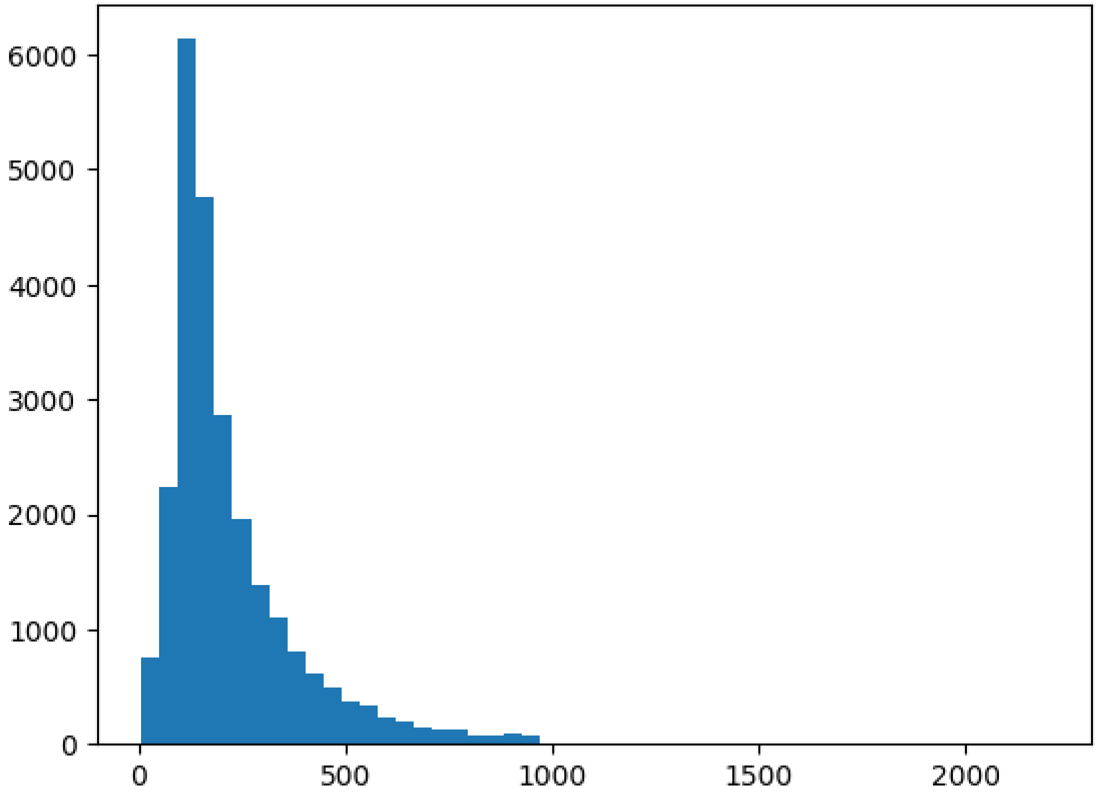
Padding

Neural networks expect data of equal size. Therefore sequences of word indexes should be constrained to have a selected length `maxlen`, and if required padded by 0s.

The optimal `maxlen` value should be established based on distribution of sequence lengths and overall classifier performance.

In [28]:
```
import matplotlib.pyplot as plt

lens = [len(s) for s in seq_train]
_ =plt.hist(lens,bins=50)
```



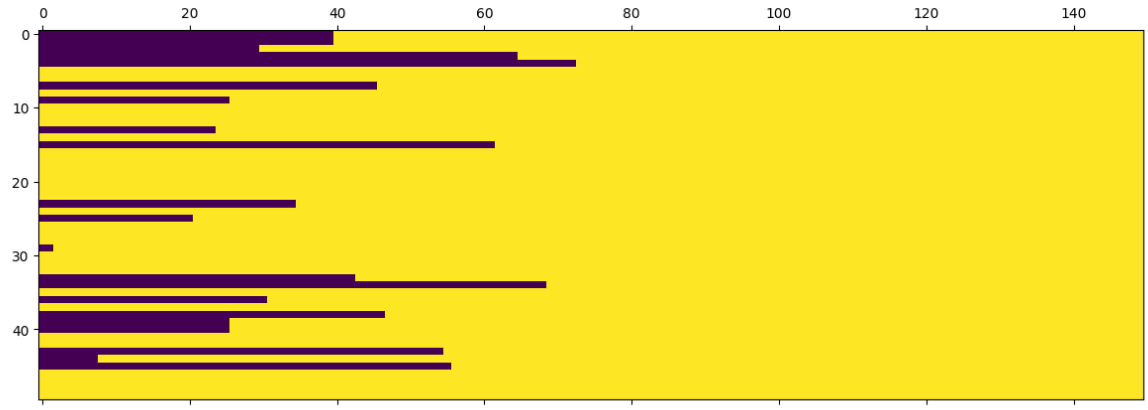
In [29]:
```
# FIXME

import keras
maxlen = 150
seq_train_padded = keras.utils.pad_sequences(seq_train, maxlen=maxlen)
seq_test_padded = keras.utils.pad_sequences(seq_test, maxlen=maxlen)
```

Examples of padded sequences

In [30]:
```
import matplotlib.pyplot as plt
print(seq_train_padded[3])
sub_seq = seq_train_padded[0:50,:]
sub_seq=np.where(sub_seq>0,1,0)
_=plt.matshow(sub_seq,cmap=None)
```

```
[    0     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0    11     6     1   240     4
    19    15     3  8775  2706  2652    51     1   357     4     1   179    67   137
  1400    16    91   202   967    14    22    80     3   191  3108  3036     2    15
     3   375     4   631   386   351    36  6354     2  5558  1927    14   207  8595
  3432     1   111   364    47    23    54     5  1656    54  1817  4311    40     3
  2262     2  1907   140   159   779   110    30    91   115     3   220    19     8
   172   278     2    28    12  1011     1  2813     9  1984]
```



Classifier

We define the network architecture. Please observe the number of parameters for the Embedding layer. It is equal to the numeber of words multiplied by the size of embedding vectors

In [31]:
```
from keras.models import Sequential
from keras.layers import Flatten, Dense
from keras.layers import Embedding

model = Sequential()
# We specify the maximum input length to our Embedding layer, so we can l
model.add(Embedding(input_dim=10_000, output_dim=8, input_length=maxlen))
# After the Embedding layer, our activations have shape `(samples, maxlen

# We flatten the 3D tensor of embeddings into a 2D tensor of shape `(samp
model.add(Flatten())

# Finally, we add the classifier on top
model.add(Dense(2, activation='softmax'))
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy'
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 embedding (Embedding)       (None, 150, 8)            80000     
                                                                 
 flatten (Flatten)           (None, 1200)              0         
                                                                 
 dense (Dense)               (None, 2)                 2402      
                                                                 
=================================================================
Total params: 82,402
Trainable params: 82,402
Non-trainable params: 0
_________________________________________________________________
```

Train the model

In [32]:
```
print(seq_train_padded.shape)
hist = model.fit(seq_train_padded, df_train.label,
                    epochs=20,
                    batch_size=32,
                    validation_split=0.2)
```

```
(25000, 150)
Epoch 1/20
625/625 [==============================] - 1s 977us/step - loss: 0.5534
- acc: 0.7278 - val_loss: 0.3696 - val_acc: 0.8518
Epoch 2/20
625/625 [==============================] - 1s 869us/step - loss: 0.2966
- acc: 0.8806 - val_loss: 0.2938 - val_acc: 0.8772
Epoch 3/20
625/625 [==============================] - 0s 792us/step - loss: 0.2296
- acc: 0.9094 - val_loss: 0.2849 - val_acc: 0.8816
Epoch 4/20
625/625 [==============================] - 1s 835us/step - loss: 0.1904
- acc: 0.9299 - val_loss: 0.2883 - val_acc: 0.8786
Epoch 5/20
625/625 [==============================] - 1s 919us/step - loss: 0.1583
- acc: 0.9439 - val_loss: 0.2997 - val_acc: 0.8718
Epoch 6/20
625/625 [==============================] - 1s 845us/step - loss: 0.1310
- acc: 0.9554 - val_loss: 0.3155 - val_acc: 0.8694
Epoch 7/20
625/625 [==============================] - 0s 716us/step - loss: 0.1068
- acc: 0.9645 - val_loss: 0.3282 - val_acc: 0.8680
Epoch 8/20
625/625 [==============================] - 0s 692us/step - loss: 0.0852
- acc: 0.9741 - val_loss: 0.3487 - val_acc: 0.8640
Epoch 9/20
625/625 [==============================] - 0s 687us/step - loss: 0.0674
- acc: 0.9816 - val_loss: 0.3702 - val_acc: 0.8576
Epoch 10/20
625/625 [==============================] - 0s 679us/step - loss: 0.0523
- acc: 0.9870 - val_loss: 0.3933 - val_acc: 0.8572
Epoch 11/20
625/625 [==============================] - 0s 682us/step - loss: 0.0398
- acc: 0.9903 - val_loss: 0.4248 - val_acc: 0.8522
Epoch 12/20
625/625 [==============================] - 0s 686us/step - loss: 0.0305
- acc: 0.9932 - val_loss: 0.4444 - val_acc: 0.8494
Epoch 13/20
625/625 [==============================] - 0s 684us/step - loss: 0.0227
- acc: 0.9951 - val_loss: 0.4753 - val_acc: 0.8498
Epoch 14/20
625/625 [==============================] - 0s 682us/step - loss: 0.0170
- acc: 0.9964 - val_loss: 0.4988 - val_acc: 0.8496
Epoch 15/20
625/625 [==============================] - 0s 681us/step - loss: 0.0124
- acc: 0.9973 - val_loss: 0.5242 - val_acc: 0.8466
Epoch 16/20
625/625 [==============================] - 0s 685us/step - loss: 0.0091
- acc: 0.9979 - val_loss: 0.5492 - val_acc: 0.8480
Epoch 17/20
625/625 [==============================] - 0s 686us/step - loss: 0.0068
- acc: 0.9987 - val_loss: 0.5756 - val_acc: 0.8450
Epoch 18/20
625/625 [==============================] - 0s 688us/step - loss: 0.0049
- acc: 0.9992 - val_loss: 0.6024 - val_acc: 0.8454
Epoch 19/20
625/625 [==============================] - 0s 679us/step - loss: 0.0038
- acc: 0.9994 - val_loss: 0.6293 - val_acc: 0.8444
Epoch 20/20
```

```
625/625 [==============================] - 0s 676us/step - loss: 0.0029
- acc: 0.9996 - val_loss: 0.6399 - val_acc: 0.8448
```

TODO 6.4.1 make predictions and print classification scores

In [33]:
```
y_test=[label.numpy() for img,label in ds_test]

probs = model.predict(seq_test_padded)
y_pred = np.argmax(probs,axis=1)

print(classification_report(y_test, y_pred))
```

```
782/782 [==============================] - 0s 335us/step
              precision    recall  f1-score   support

           0       0.85      0.84      0.84     12500
           1       0.84      0.85      0.84     12500

    accuracy                           0.84     25000
   macro avg       0.84      0.84      0.84     25000
weighted avg       0.84      0.84      0.84     25000
```

TODO 6.4.2 Repeat experiments for various values of lengths of input sequence `maxlen in [100,250,500]` and dimensions of embeddings `output_dim in [8, 32, 64]`. Present results in a table (markdown or data frame)

In [34]:
```
max_lenghts = [100, 250, 500]
output_dims = [8, 32, 64]


for max_length, output_dim in zip(max_lenghts, output_dims):

    seq_train_padded = keras.utils.pad_sequences(seq_train, maxlen=max_le
    seq_test_padded = keras.utils.pad_sequences(seq_test, maxlen=max_leng

    model = Sequential()
    # We specify the maximum input length to our Embedding layer, so we c
    model.add(Embedding(input_dim=10_000, output_dim=output_dim, input_le
    # After the Embedding layer, our activations have shape `(samples, ma

    # We flatten the 3D tensor of embeddings into a 2D tensor of shape `(
    model.add(Flatten())

    # Finally, we add the classifier on top
    model.add(Dense(2, activation='softmax'))
    model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentr
    model.summary()

    y_test=[label.numpy() for img,label in ds_test]

    probs = model.predict(seq_test_padded)
    y_pred = np.argmax(probs,axis=1)

    print(classification_report(y_test, y_pred))
    print("----------------------------")
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 100, 8)            80000

 flatten_1 (Flatten)         (None, 800)               0

 dense_1 (Dense)             (None, 2)                 1602

=================================================================
Total params: 81,602
Trainable params: 81,602
Non-trainable params: 0
_________________________________________________________________
782/782 [==============================] - 0s 355us/step
              precision    recall  f1-score   support

           0       0.50      0.48      0.49     12500
           1       0.50      0.52      0.51     12500

    accuracy                           0.50     25000
   macro avg       0.50      0.50      0.50     25000
weighted avg       0.50      0.50      0.50     25000

----------------------------
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, 250, 32)           320000

 flatten_2 (Flatten)         (None, 8000)              0

 dense_2 (Dense)             (None, 2)                 16002

=================================================================
Total params: 336,002
Trainable params: 336,002
Non-trainable params: 0
_________________________________________________________________
782/782 [==============================] - 0s 583us/step
              precision    recall  f1-score   support

           0       0.50      0.76      0.60     12500
           1       0.49      0.24      0.32     12500

    accuracy                           0.50     25000
   macro avg       0.50      0.50      0.46     25000
weighted avg       0.50      0.50      0.46     25000

----------------------------
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, 500, 64)           640000

 flatten_3 (Flatten)         (None, 32000)             0
```

```
 dense_3 (Dense)             (None, 2)                 64002

=================================================================
Total params: 704,002
Trainable params: 704,002
Non-trainable params: 0
_________________________________________________________________
782/782 [==============================] - 1s 852us/step
              precision    recall  f1-score   support

           0       0.50      0.36      0.42     12500
           1       0.50      0.64      0.56     12500

    accuracy                           0.50     25000
   macro avg       0.50      0.50      0.49     25000
weighted avg       0.50      0.50      0.49     25000

----------------------------
```

6.5 Using Glove embeddings

We will download embeddings obtained by training on large text corpora and try to use them as weights of Embdedding layer.

We will use gensim library providing a number of pretrained embeddings models.

In [35]:
```
import gensim
import gensim.downloader as api

nlp = gensim.downloader.load('glove-wiki-gigaword-100')
```

Check the dimension of an embedding vector

In [36]:
```
embedding = nlp['film']
print(type(embedding))
print(embedding.shape)
```

```
<class 'numpy.ndarray'>
(100,)
```

Using gensim you may easily compute distance between terms by computing distance between embeddings.

In [37]:
```
print(nlp.distance("film","movie"))
```

```
0.09448790550231934
```

TODO 6.5.1 Prepare 10 pairs of words, for which you expect certain similartity and display their distance based on embeddings.

In [38]:
```
words=[
       ["film","movie"],
       ["actor","cast"],
       ["house","building"],
       ["sky","cloud"],
       ["money", "cash"],
```

```
        ["nice", "pleasant"],
        ["old", "ancient"],
        ["weak", "fragile"],
        ["true", "real"],
        ["rich", "wealthy"],
        ["positive", "optimistic"],
        ["key", "critical"],
        ["difficult", "hard"],
        ["honest", "fair"]
]

for pair in words:
  print(f'{pair[0]}\t{pair[1]}: {nlp.distance(pair[0],pair[1])}')
```

```
film    movie: 0.09448790550231934
actor   cast: 0.32412874698638916
house   building: 0.3152713179588318
sky     cloud: 0.38010847568511963
money   cash: 0.1515163779258728
nice    pleasant: 0.4089140295982361
old     ancient: 0.5115182399749756
weak    fragile: 0.41438615322113037
true    real: 0.3087722063064575
rich    wealthy: 0.3595759868621826
positive        optimistic: 0.4089689254760742
key     critical: 0.2988662123680115
difficult       hard: 0.2147454023361206
honest  fair: 0.42741161584854126
```

Weights matrix for Embedding layer

Building the weights matrix we must consider two elements:

- vocabulary used by tokenizer (limited to a certain number of words)
- mapping from words to embeddings within gensim

Small test performed on a list of three strings

In [39]:
```
tok = Tokenizer(num_words=4)
texts = np.array(["apple big cat","cat dog apple","apple eat dog dog dog"
tok.fit_on_texts(texts)
print(tok.word_counts)
print(tok.word_index)
seq = tok.texts_to_sequences(texts)
print(seq)
```

```
OrderedDict([('apple', 3), ('big', 1), ('cat', 2), ('dog', 4), ('eat',
1)])
{'dog': 1, 'apple': 2, 'cat': 3, 'big': 4, 'eat': 5}
[[2, 3], [3, 1, 2], [2, 1, 1, 1]]
```

As num_wors=4, only three words were used (index 0 is reserved for padding). They were selected based on frequency (the most frequent 3 words are dog, apple and cat).

Below the function to build weights matrix. Embeddings are arranged as rows.

In [40]:
```
# dict maps words to indexes
# nlp is gensim wrapper mapping word to embedding vector

def prepare_embedding_weights(dict,nlp,input_dim=10_000, output_dim=100):
  weights = np.zeros((input_dim,output_dim) )

  for word in dict:
    idx = dict[word]
    if idx==input_dim:
      break
    if word in nlp.key_to_index:
      embedding = nlp[word]
    else:
      continue
    # print(embedding)
    length = min(len(embedding),output_dim)
    weights[idx,:length]=embedding[:length]

  return weights

weights = prepare_embedding_weights(tok.word_index,nlp,input_dim=4, outpu
print(weights.shape)
```

```
(4, 50)
```

Model preparation

We set the embedding layer as trainable. This can be switched off with:

`embedding_layer = Embedding(input_dim=10_000, output_dim=100, input_length=maxlen,weights=[weights],trainable=False)`

In [41]:
```
import tensorflow as tf
maxlen = 300

model = Sequential()
# We specify the maximum input length to our Embedding layer, so we can l
weights = prepare_embedding_weights(tokenizer.word_index,nlp,input_dim=10
embedding_layer = Embedding(input_dim=10_000, output_dim=100, input_lengt
                            trainable=True)
model.add(embedding_layer)
# After the Embedding layer, our activations have shape `(samples, maxlen

# We flatten the 3D tensor of embeddings into a 2D tensor of shape `(samp
model.add(Flatten())
# model.add(Dense(512, activation='relu'))

# Finally, we add the classifier on top
model.add(Dense(2, activation='softmax'))
model.compile(optimizer=tf.keras.optimizers.RMSprop(0.5), loss='sparse_ca
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_4 (Embedding)     (None, 300, 100)          1000000

 flatten_4 (Flatten)         (None, 30000)             0

 dense_4 (Dense)             (None, 2)                 60002

=================================================================
Total params: 1,060,002
Trainable params: 1,060,002
Non-trainable params: 0
_________________________________________________________________
```

Training and testing

Actually, we fine-tune embeddings. Results for frozen embeddings are much worse.

In [42]:

```
seq_train_padded = keras.utils.pad_sequences(seq_train, maxlen=maxlen)
seq_test_padded = keras.utils.pad_sequences(seq_test, maxlen=maxlen)

hist = model.fit(seq_train_padded, df_train.label,
                    epochs=10,
                    batch_size=32,
                    validation_split=0.2)
```

```
Epoch 1/10
625/625 [==============================] - 3s 4ms/step - loss: 3223.2078
- acc: 0.7171 - val_loss: 5457.8467 - val_acc: 0.7558
Epoch 2/10
625/625 [==============================] - 3s 4ms/step - loss: 2912.2390
- acc: 0.8794 - val_loss: 7486.2241 - val_acc: 0.7918
Epoch 3/10
625/625 [==============================] - 3s 5ms/step - loss: 2178.7944
- acc: 0.9294 - val_loss: 9702.6221 - val_acc: 0.8156
Epoch 4/10
625/625 [==============================] - 3s 5ms/step - loss: 1601.3752
- acc: 0.9535 - val_loss: 12402.9443 - val_acc: 0.8172
Epoch 5/10
625/625 [==============================] - 3s 5ms/step - loss: 1466.3060
- acc: 0.9636 - val_loss: 14952.6133 - val_acc: 0.8172
Epoch 6/10
625/625 [==============================] - 3s 6ms/step - loss: 1158.5391
- acc: 0.9722 - val_loss: 17235.5273 - val_acc: 0.8276
Epoch 7/10
625/625 [==============================] - 4s 6ms/step - loss: 923.6838
- acc: 0.9789 - val_loss: 20480.3945 - val_acc: 0.8210
Epoch 8/10
625/625 [==============================] - 3s 6ms/step - loss: 925.8815
- acc: 0.9799 - val_loss: 22970.8125 - val_acc: 0.8186
Epoch 9/10
625/625 [==============================] - 3s 5ms/step - loss: 840.7166
- acc: 0.9837 - val_loss: 24783.4160 - val_acc: 0.8200
Epoch 10/10
625/625 [==============================] - 3s 5ms/step - loss: 742.1108
- acc: 0.9850 - val_loss: 26368.4648 - val_acc: 0.8294
```

TODO 6.5.2 Make predictions, compute labels and print classification report

In [43]:
```
y_test=[label.numpy() for img,label in ds_test]

probs = model.predict(seq_test_padded)
y_pred = np.argmax(probs,axis=1)

print(classification_report(y_test, y_pred))
```

```
782/782 [==============================] - 1s 1ms/step
              precision    recall  f1-score   support

           0       0.82      0.82      0.82     12500
           1       0.82      0.82      0.82     12500

    accuracy                           0.82     25000
   macro avg       0.82      0.82      0.82     25000
weighted avg       0.82      0.82      0.82     25000
```

The code below displays sequence, padded sequence, performs classification and determines the predicted label.

TODO 6.5.3 Try it yourself. Perpare 10 sentences and display predicted labels. Gather them in a table. Write a short comment to the results.

In [44]:
```
#FIXME

text_instance=[]
text_instance.append('great film nice scenario')
text_instance.append('surprising action great play')
text_instance.append('amazing document fantastic film')
text_instance.append('creative screenplay amazing movie')
text_instance.append('horrible show boring plot')
text_instance.append('riveting plot incredible')
text_instance.append('dull movie flat plot')
text_instance.append('mundane screenplay lifeless plot')
text_instance.append('tedious film dry scenario')
text_instance.append('unentertaiing show monotonous plot')

labels = ['Negative', 'Positive']

for sentence in text_instance:
    seq = tokenizer.texts_to_sequences([sentence])
    seq_padded = keras.utils.pad_sequences(seq, maxlen=maxlen)
    probs = model.predict(seq_padded)
    y_pred = np.argmax(probs,axis=1)
    print(sentence)
    print(labels[int(y_pred)], "\n\n")
```

```
1/1 [==============================] - 0s 11ms/step
great film nice scenario
Positive


1/1 [==============================] - 0s 11ms/step
surprising action great play
Positive


1/1 [==============================] - 0s 10ms/step
amazing document fantastic film
Positive


1/1 [==============================] - 0s 11ms/step
creative screenplay amazing movie
Positive


1/1 [==============================] - 0s 10ms/step
horrible show boring plot
Negative


1/1 [==============================] - 0s 10ms/step
riveting plot incredible
Positive


1/1 [==============================] - 0s 11ms/step
dull movie flat plot
Negative


1/1 [==============================] - 0s 11ms/step
mundane screenplay lifeless plot
Negative


1/1 [==============================] - 0s 10ms/step
tedious film dry scenario
Negative


1/1 [==============================] - 0s 10ms/step
unentertaiing show monotonous plot
Negative
```