Lecture 7
Applying Machine Learning to Sentiment Analysis

Wonjun Lee

Contents

- Sequence Processing – Word Embedding

K-means Clustering

Sequence Processing

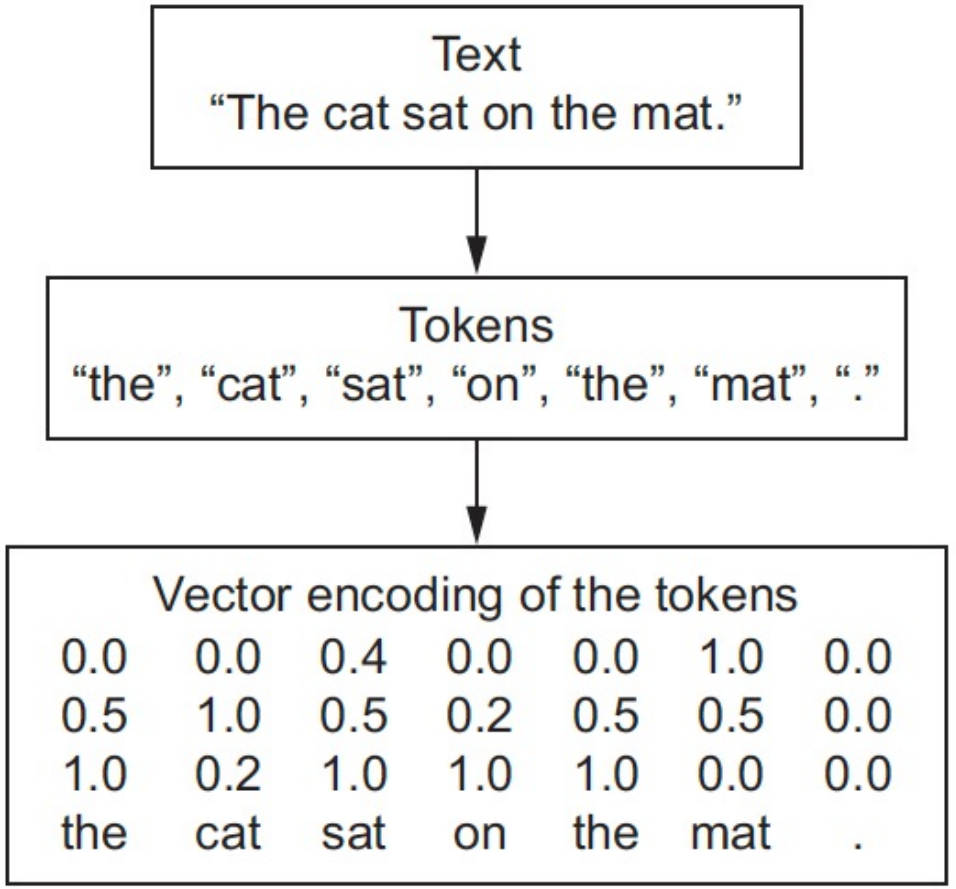
- Sequence processing by machine learning algorithms
  - ✓Text
  - ✓Timeseries
  - ✓Sequence data in general
- Applications of algorithms
  - ✓Document classification: identifying topic of an article, author of a book
  - ✓Timeseries comparisons: estimating how closely related two documents or two stock tickets are
  - ✓Sequence-to-sequence learning: decoding English into French
  - ✓Sentiment analysis: classifying sentiment of tweets or movie reviews as + or –
  - ✓Timeseries forecasting: predicting future weather at a certain location, given recent weather data

Working with text data

- *Vectorizing* text is the process of transforming text into numeric tensors
  - ✓Segment text into **words** → transform each word into a vector
  - ✓Segment text into **characters** → transform each character into a vector
  - ✓Extract **n-grams** of words or characters → transform each n-gram into a vector (n-grams are overlapping groups of multiple consecutive words or characters)

Working with text data

- Token: different unit into which you can break down text (words, characters, or n-grams)
- Tokenization: breaking text into tokens
- All text-vectorization processes consists of
  - ✓Tokenization
  - ✓Associating numeric vectors with generated tokens - multiple ways to associate a vector with a token: one-hot encoding, token embedding



One-hot encoding of words and characters

- It consists of associating a unique integer index with every word and then turning this integer index *i* into a binary vector of size *N* (the size of the vocabulary); the vector is all zeros except for the *i* th entry, which is 1

```
from keras.preprocessing.text import Tokenizer
samples = ['The cat sat on the mat.', 'The dog ate my homework.']
tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(samples)

sequences = tokenizer.texts_to_sequences(samples)

one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

Creates a tokenizer, configured to only take into account the 1,000 most common words

Builds the word index

Turns strings into lists of integer indices

How you can recover the word index that was computed

You could also directly get the one-hot binary representations. Vectorization modes other than one-hot encoding are supported by this tokenizer.

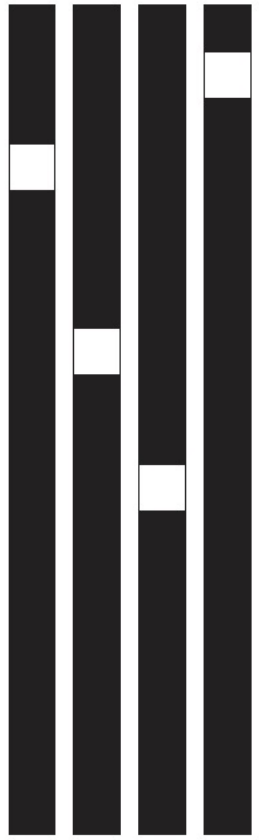
```
>>> sequences
[[1, 2, 3, 4, 1, 5], [1, 6, 7, 8, 9]]
>>> one_hot_results
array([[0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.]])
>>> word_index
{'on': 4, 'ate': 7, 'mat': 5, 'dog': 6, 'cat': 2, 'the': 1, 'my': 8, 'homework': 9, 'sat': 3}
```

One-hot hashing trick

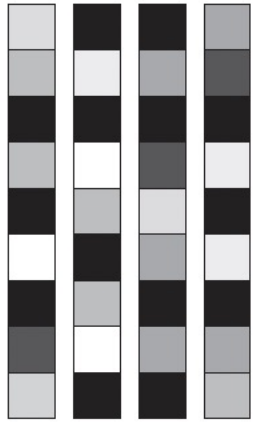
- when the number of unique tokens in your vocabulary is too large to handle
- One drawback: hash collisions
  - ✓Decreases when the dimensionality of the hashing space is much larger than the total number of unique tokens being hashed

Word embeddings

- Another associating way to vector with a word



One-hot word vectors:
- Sparse
- High-dimensional
- Hardcoded



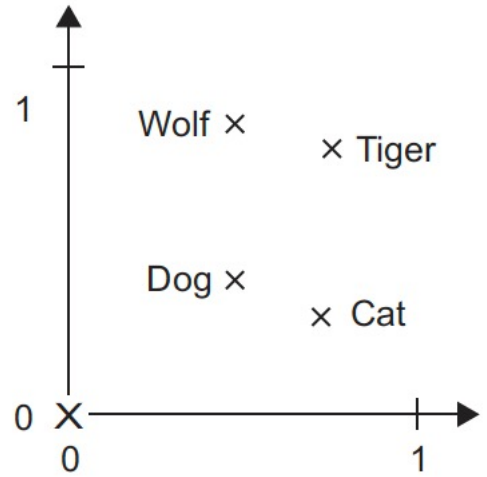
Word embeddings:
- Dense
- Lower-dimensional
- Learned from data

Learning word embeddings with embedding layer

- The simplest way to associate a dense vector with a word: choose the vector at random
  - ✓The problem of this approach: ex) *"accurate"* and *"exact"* end up with completely different embeddings, even though they are interchangeable

  ➔ geometric relationships between word vectors should reflect the semantic relationships between these words

  - EX) Four words are embedded on a 2D plane: *cat, dog, wolf,* and *tiger*
  - With the vector representations we chose here, some semantic relationships between these words can be encoded as geometric transformations
  - For instance, the same vector allows us to go from *cat* to *tiger* and from *dog* to *wolf*
    - this vector could be interpreted as the "from **pet** to **wild animal**" vector
  - Similarly, another vector lets us go from *dog* to *cat* and from *wolf* to *tiger*
    - interpreted as a "from **canine** to **feline**" vector



Learning word embeddings with embedding layer

- Common examples of meaningful geometric transformations – "**gender**" vectors and "**plural**" vectors
  - ✓For instance, by adding a "female" vector to the vector "king" we obtain the vector "queen"
  - ✓By adding a "plural" vector, we obtain "kings"
- A language is the reflection of a specific culture and a specific context
- But more pragmatically, what makes a good word-embedding space depends heavily on your task
  - ✓Because the importance of certain semantic relationships varies from task to task

Obtaining the Movie Review Dataset

- Download a compressed archive of the movie review dataset from http://ai.stanford.edu/~amaas/data/sentiment/

```
import os
import sys
import tarfile
import time
import urllib.request


source = 'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz'
target = 'aclImdb_v1.tar.gz'
```

Preprocessing the Movie Review Dataset into More Convenient Format

```
import pandas as pd
import os

# change the `basepath` to the directory of the
# unzipped movie dataset

basepath = 'aclImdb'

labels = {'pos': 1, 'neg': 0}

df = pd.DataFrame()
for s in ('test', 'train'):
    for l in ('pos', 'neg'):
        path = os.path.join(basepath, s, l)
        for file in sorted(os.listdir(path)):
            with open(os.path.join(path, file),
                      'r', encoding='utf-8') as infile:
                txt = infile.read()
            df = df.append([[txt, labels[l]]],
                           ignore_index=True)

df.columns = ['review', 'sentiment']
```

Transforming Documents into Feature Vectors

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
...     'This is the first document.',
...     'This document is the second document.',
...     'And this is the third one.',
...     'Is this the first document?',
... ]
>>> vectorizer = CountVectorizer()
>>> X = vectorizer.fit_transform(corpus)
>>> print(vectorizer.get_feature_names())
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
>>> print(X.toarray())
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
```

Transforming Documents into Feature Vectors

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

count = CountVectorizer()
docs = np.array([
        'The sun is shining',
        'The weather is sweet',
        'The sun is shining, the weather is sweet, and one and one is two'])
bag = count.fit_transform(docs)
```

Transforming Documents into Feature Vectors

- The vocabulary is stored in a Python dictionary, which maps the unique words that are mapped to integer indices

```
print(count.vocabulary_)
```

```
{'the': 6, 'sun': 4, 'is': 1, 'shining': 3, 'weather': 8, 'sweet': 5, 'and': 0, 'one': 2, 'two': 7}
```

Transforming Documents into Feature Vectors

- Each index position in the feature vectors shown here corresponds to the integer values that are stored as dictionary items in the `CountVectorizer` vocabulary

```
print(bag.toarray())
```

```
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

- Raw term frequencies: *tf (t,d)* — the number of times *a* term *t* occurs in a document *d*

Assessing Word Relevancy Via Term Frequency-Inverse Document Frequency

- Frequently occurring words typically don't contain useful information – Down weight such words in feature vectors

$$\text{tf-idf}(t, d) = \text{tf}\ (\text{t,d}) \times \text{idf}(t, d)$$

$$\text{idf}(t, d) = \log \frac{n_d}{1 + \text{df}(d, t)}$$

- Where n_d = total # of documents, df(d, t) = # of documents d that contains the term t
  - ✓Log is used to ensure that low document frequencies are not given too much weight

Assessing Word Relevancy Via Term Frequency-Inverse Document Frequency

- `TfidTransformer`
  - ✓Takes the raw term frequencies from the `CountVectorizer` class and transforms them into `tf-idfs`

```
from sklearn.feature_extraction.text import TfidfTransformer

tfidf = TfidfTransformer(use_idf=True,
                         norm='l2',
                         smooth_idf=True)
print(tfidf.fit_transform(count.fit_transform(docs))
      .toarray())
```

```
[[0.   0.43 0.   0.56 0.56 0.   0.43 0.   0.  ]
 [0.   0.43 0.   0.   0.   0.56 0.43 0.   0.56]
 [0.5  0.45 0.5  0.19 0.19 0.19 0.3  0.25 0.19]]
```

Assessing Word Relevancy Via Term Frequency-Inverse Document Frequency

- 'is' had the largest term frequency in the 3rd document

```
>>> print(bag.toarray())
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

- Now associated with a relatively small tf-idf in the 3rd document
  - ✓Since it is also present in the first and second document and thus unlikely to contain any useful discriminatory information

Assessing Word Relevancy Via Term Frequency-Inverse Document Frequency

- The equation for the inverse document frequency implemented in scikit-learn:

$$idf(t,d) = \log \frac{1 + \mathrm{n_d}}{1 + df(d,t)}$$

- Similarly, the tf-idf computed in scikit-learn deviates slightly from default equation:

$$tf\text{-}idf(t,d) = tf(t,d) \times (idf(t,d) + 1)$$

- +1 : due to setting smooth_idf = True → assigning zero-weight (*idf(t,d)* = log(1) = 0) to terms that occur in all documents

Assessing Word Relevancy Via Term Frequency-Inverse Document Frequency

- `TfidTransformer`
  - ✓Directly normalizes the raw term frequencies before calculating
  - ✓By default (norm = ‘l2’), applies the L2 normalization: divide an unnormalized feature vector, v, by its L2-norm:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{{v_1}^2 + {v_2}^2 + \cdots + {v_n}^2}} = \frac{v}{(\sum_{i=1}^{n} v_i^2)^{1/2}}$$

Assessing Word Relevancy Via Term Frequency-Inverse Document Frequency

- Ex) ‘is’ has a term frequency, 3 (*tf* = 3) in the 3rd document,
  has a document frequency of this term 3 since the term ‘is’
  occurs in all three documents (*df* = 3)

$$idf(\text{"is"}, d_3) = \log \frac{1+3}{1+3} = 0$$

$$tf\text{-}idf(\text{"is"}, d_3) = 3 \times (0 + 1) = 3$$

- tf-idf vectors: [3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]

$$tf\text{-}idf(d_3)_{norm} = \frac{[3.39,\ 3.0,\ 3.39,\ 1.29,\ 1.29,\ 1.29,\ 2.0,\ 1.69,\ 1.29]}{\sqrt{3.39^2 + 3.0^2 + 3.39^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.0^2 + 1.69^2 + 1.29^2}}$$

$$= [0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19]$$

$$tf\text{-}idf(\text{"is"}, d_3) = 0.45$$

Processing Documents into Tokens

- `PorterStemmer` – process of transforming a word into its root form

```
from nltk.stem.porter import PorterStemmer

porter = PorterStemmer()

def tokenizer(text):
    return text.split()


def tokenizer_porter(text):
    return [porter.stem(word) for word in text.split()]
```

```
tokenizer('runners like running and thus they run')
```

```
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

```
tokenizer_porter('runners like running and thus they run')
```

```
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

Processing Documents into Tokens

- Stop-word removal : Stop-words are simply those words that are extremely common in all sorts of texts and probably bear no (or only a little) useful information
  - ✓Ex) is, and, has, like
- Use set of 127 English stop-words : available from the NLTK library (`nltk.download`)

```
import nltk

nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /home/jovyan/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

```
True
```

Processing Documents into Tokens

```
from nltk.corpus import stopwords

stop = stopwords.words('english')
[w for w in tokenizer_porter('a runner likes running and runs a lot')[-10:]
if w not in stop]
```

```
['runner', 'like', 'run', 'run', 'lot']
```

Training a Logistic Regression Model for Document Classification

- Train logistic regression model to classify movie reviews into positive and negative: Divide DataFrame of cleaned text documents into 25,000 documents for training and 25,000 documents for testing

```
X_train = df.loc[:25000, 'review'].values
y_train = df.loc[:25000, 'sentiment'].values
X_test = df.loc[25000:, 'review'].values
y_test = df.loc[25000:, 'sentiment'].values
```

Training a Logistic Regression Model for Document Classification

- Use a GridSearchCV object to find the optimal set of parameters for logistic regression model using 5-fold stratified cross-validation

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import GridSearchCV

tfidf = TfidfVectorizer(strip_accents=None,
                        lowercase=False,
                        preprocessor=None)

param_grid = [{'vect__ngram_range': [(1, 1)],
               'vect__stop_words': [stop, None],
               'vect__tokenizer': [tokenizer, tokenizer_porter],
               'clf__penalty': ['l1', 'l2'],
               'clf__C': [1.0, 10.0, 100.0]},
              {'vect__ngram_range': [(1, 1)],
               'vect__stop_words': [stop, None],
               'vect__tokenizer': [tokenizer, tokenizer_porter],
               'vect__use_idf':[False],
               'vect__norm':[None],
               'clf__penalty': ['l1', 'l2'],
               'clf__C': [1.0, 10.0, 100.0]},
              ]

lr_tfidf = Pipeline([('vect', tfidf),
                     ('clf', LogisticRegression(random_state=0, solver='liblinear'))])

gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
                           scoring='accuracy',
                           cv=5,
                           verbose=2,
                           n_jobs=-1)
```

Training a Logistic Regression Model for Document Classification

```
print('Best parameter set: %s ' % gs_lr_tfidf.best_params_)
```

```
Best parameter set: {'clf__C': 10.0, 'vect__stop_words': None,
'clf__penalty': 'l2', 'vect__tokenizer': <function tokenizer at
0x7f6c704948c8>, 'vect__ngram_range': (1, 1)}
```

- Average 5-fold cross-validation accuracy scores on the training dataset

```
print('CV Accuracy: %.3f' % gs_lr_tfidf.best_score_)
```

- Classification accuracy on the test dataset

```
clf = gs_lr_tfidf.best_estimator_
print('Test Accuracy: %.3f' % clf.score(X_test, y_test))
```

Working with Bigger Data – Online Algorithms and Out-Of-Core Learning

- Computationally expensive to construct feature vectors for 50,000 movie review dataset during grid search
- Out-of-core learning – allows to work with large datasets by fitting the classifier incrementally on smaller batches of a dataset

Working with Bigger Data – Online Algorithms and Out-Of-Core Learning

```
import os
import gzip


if not os.path.isfile('movie_data.csv'):
    if not os.path.isfile('movie_data.csv.gz'):
        print('Please place a copy of the movie_data.csv.gz'
              'in this directory. You can obtain it by'
              'a) executing the code in the beginning of this'
              'notebook or b) by downloading it from GitHub:'
              'https://github.com/rasbt/python-machine-learning-'
              'book-2nd-edition/blob/master/code/ch08/movie_data.csv.gz')
    else:
        with gzip.open('movie_data.csv.gz', 'rb') as in_f, \
                open('movie_data.csv', 'wb') as out_f:
            out_f.write(in_f.read())
```

Working with Bigger Data – Online Algorithms and Out-Of-Core Learning

```
import numpy as np
import re
from nltk.corpus import stopwords

stop = stopwords.words('english')


def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) +\
        ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized
```

Working with Bigger Data – Online Algorithms and Out-Of-Core Learning

```
def stream_docs(path):
    with open(path, 'r', encoding='utf-8') as csv:
        next(csv)  # skip header
        for line in csv:
            text, label = line[:-3], int(line[-2])
            yield text, label
```

Working with Bigger Data – Online Algorithms and Out-Of-Core Learning

```
def get_minibatch(doc_stream, size):
    docs, y = [], []
    try:
        for _ in range(size):
            text, label = next(doc_stream)
            docs.append(text)
            y.append(label)
    except StopIteration:
        return None, None
    return docs, y
```

Working with Bigger Data – Online Algorithms and Out-Of-Core Learning

- Can't use `CountVectorizer` for out-of-core learning since it requires holding the complete vocabulary in memory
- Also, `TfidfVectorizer` needs to keep all the feature vectors of the training dataset in memory to calculate the inverse document frequencies
- `HashingVectorizer` : Converts a collection of text documents to a matrix of token occurrences

```
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.linear_model import SGDClassifier


vect = HashingVectorizer(decode_error='ignore',
                         n_features=2**21,
                         preprocessor=None,
                         tokenizer=tokenizer)
```

Working with Bigger Data – Online Algorithms and Out-Of-Core Learning

```
from distutils.version import LooseVersion as Version
from sklearn import __version__ as sklearn_version

clf = SGDClassifier(loss='log', random_state=1)


doc_stream = stream_docs(path='movie_data.csv')
```

Working with Bigger Data – Online Algorithms and Out-Of-Core Learning

```
import pyprind
pbar = pyprind.ProgBar(45)

classes = np.array([0, 1])
for _ in range(45):
    X_train, y_train = get_minibatch(doc_stream, size=1000)
    if not X_train:
        break
    X_train = vect.transform(X_train)
    clf.partial_fit(X_train, y_train, classes=classes)
    pbar.update()
```

- Iterated over 45 mini-batches of documents where each mini-batch consists of 1,000 documents

Working with Bigger Data – Online Algorithms and Out-Of-Core Learning

- Use the last 5,000 documents to evaluate the performance

```
X_test, y_test = get_minibatch(doc_stream, size=5000)
X_test = vect.transform(X_test)
print('Accuracy: %.3f' % clf.score(X_test, y_test))
```

- Out-of-core learning is very memory efficient and it took less than a minute to complete
- Finally, we can use the last 5,000 documents to update our model

```
clf = clf.partial_fit(X_test, y_test)
```

Decomposing Text Documents with Latent Dirichlet Allocation

- Set the maximum document frequency of words to be considered to 10 percent (max_df=.1)
- Limited the number of words to be considered to the most frequently occurring 5,000 words (max_features=5000)

```
from sklearn.feature_extraction.text import CountVectorizer

count = CountVectorizer(stop_words='english',
                        max_df=.1,
                        max_features=5000)
X = count.fit_transform(df['review'].values)
```

Decomposing Text Documents with Latent Dirichlet Allocation

- Infer the 10 different topics from the documents
- `learning_method='batch'`:lda estimator does its estimation based on all available training data in one iteration
  - ✓Slower than ‘online’ (mini batch)

```
from sklearn.decomposition import LatentDirichletAllocation

lda = LatentDirichletAllocation(n_components=10,
                                random_state=123,
                                learning_method='batch')
X_topics = lda.fit_transform(X)
```

Decomposing Text Documents with Latent Dirichlet Allocation

```
lda.components_.shape
```

```
n_top_words = 5
feature_names = count.get_feature_names()

for topic_idx, topic in enumerate(lda.components_):
    print("Topic %d:" % (topic_idx + 1))
    print(" ".join([feature_names[i]
                    for i in topic.argsort()\
                        [:-n_top_words - 1:-1]]))
```

Learning word embeddings with embedding layer

- Instantiating an Embedding layer

```
from keras.layers import Embedding

embedding_layer = Embedding(1000, 64)
```

The Embedding layer takes at least two arguments: the number of possible tokens (here, 1,000: 1 + maximum word index) and the dimensionality of the embeddings (here, 64).

- The Embedding layer is best understood as a dictionary that maps integer indices (which stand for specific words) to dense vectors
  - ✓ It takes integers as input, it looks up these integers in an internal dictionary, and it returns the associated vectors

  - ✓ Embedding layer takes as input a 2D tensor of integers of shape (samples, sequence_length)
  - ✓ All sequences in a batch must have the same length, so sequences that are shorter than others should be padded with zeros, and sequences that are longer should be truncated
  - ✓ This layer returns a 3D floating-point tensor of shape (sample, sequence_length, **embedding_dimensionality**)
  - ✓ Such a 3D tensor can then be processed by a ML

Learning word embeddings with embedding layer

- Loading the IMDB data for use with an Embedding layer

```
from keras.datasets import imdb
from keras import preprocessing

max_features = 10000
maxlen = 20

(x_train, y_train), (x_test, y_test) = imdb.load_data(
    num_words=max_features)

x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
```

Number of words to consider as features

Cuts off the text after this number of words (among the max_features most common words)

Loads the data as lists of integers

Turns the lists of integers into a 2D integer tensor of shape (samples, maxlen)

Learning word embeddings with embedding layer

- Using an Embedding layer and classifier on the IMDB data

Specifies the maximum input length to the Embedding layer so you can later flatten the embedded inputs. After the Embedding layer, the activations have shape (samples, maxlen, 8).

Flattens the 3D tensor of embeddings into a 2D tensor of shape (samples, maxlen * 8)

Adds the classifier on top

```
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential()
model.add(Embedding(10000, 8, input_length=maxlen))

model.add(Flatten())

model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
model.summary()

history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_split=0.2)
```

Learning word embeddings with embedding layer

```
_________________________________________________________________
>>> history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 1s 53us/step - loss: 0.6759 - acc: 0.6046 - val_loss: 0.6398 - val_acc: 0.6810
Epoch 2/10
20000/20000 [==============================] - 1s 37us/step - loss: 0.5658 - acc: 0.7423 - val_loss: 0.5468 - val_acc: 0.7204
Epoch 3/10
20000/20000 [==============================] - 1s 38us/step - loss: 0.4752 - acc: 0.7805 - val_loss: 0.5114 - val_acc: 0.7386
Epoch 4/10
20000/20000 [==============================] - 1s 37us/step - loss: 0.4263 - acc: 0.8077 - val_loss: 0.5008 - val_acc: 0.7452
Epoch 5/10
20000/20000 [==============================] - 1s 37us/step - loss: 0.3930 - acc: 0.8258 - val_loss: 0.4981 - val_acc: 0.7538
Epoch 6/10
20000/20000 [==============================] - 1s 37us/step - loss: 0.3667 - acc: 0.8397 - val_loss: 0.5014 - val_acc: 0.7532
Epoch 7/10
20000/20000 [==============================] - 1s 37us/step - loss: 0.3434 - acc: 0.8537 - val_loss: 0.5052 - val_acc: 0.7522
Epoch 8/10
20000/20000 [==============================] - 1s 38us/step - loss: 0.3222 - acc: 0.8655 - val_loss: 0.5132 - val_acc: 0.7486
Epoch 9/10
20000/20000 [==============================] - 1s 37us/step - loss: 0.3021 - acc: 0.8765 - val_loss: 0.5214 - val_acc: 0.7492
Epoch 10/10
20000/20000 [==============================] - 1s 38us/step - loss: 0.2838 - acc: 0.8863 - val_loss: 0.5303 - val_acc: 0.7472
>>> x_train.shape
(25000, 20)
```

- Validation accuracy of ~75%, which is pretty good considering that you're only looking at the first 20 words in every review

Using pretrained word embeddings

- Instead of learning word embeddings jointly with the problem you want to solve, you can load embedding vectors from a precomputed embedding space
- ***Word2vec*** is precomputed databases of word embeddings: by Google in 2013
- ***GloVe*** (Global Vectors for Word Representation): by Stanford in 2014
  - ✓Based on factorizing a matrix of word co-occurrence statistics
  - ✓precomputed embeddings for millions of English tokens, obtained from Wikipedia data and Common Crawl data

Processing the labels of the raw IMDB data

- Download the raw IMDB dataset as raw text at http://mng.bz/0tIo
- Collect individual training reviews into a list of strings (*texts*), one string per review and review labels into a *labels* list

```
import os
imdb_dir = '/var/containerDir/aclImdb'
train_dir = os.path.join(imdb_dir, 'train')
labels = []
texts = []
for label_type in ['neg', 'pos']:
   dir_name = os.path.join(train_dir, label_type)
   for fname in os.listdir(dir_name):
       if fname[-4:] == '.txt':
          f = open(os.path.join(dir_name, fname))
          texts.append(f.read())
          f.close()
          if label_type == 'neg':
              labels.append(0)
          else:
              labels.append(1)
```

Tokenizing the data

- Because pretrained word embeddings are meant to be particularly useful on problems where little training data is available, we restrict the training data to the first 200 samples

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 100                          /Cuts off reviews after 100 words
training_samples = 200
validation_samples = 10000
max_words = 10000

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]
```

Splits the data into a training set and a validation set, but first shuffles the data, because you're starting with data in which samples are ordered (all negative first, then all positive)

```
>>> texts[0]
"Working with one of the best Shakespeare sources, this film manages to be creditable to it's source, whilst still appealing to a
wider audience.<br /><br />Branagh steals the film from under Fishburne's nose, and there's a talented cast on good form."
>>> sequences[0]
[777, 16, 28, 4, 1, 115, 2278, 6887, 11, 19, 1025, 5, 27, 5, 42, 2425, 1861, 128, 2270, 5, 3, 6985, 308, 7, 7, 3383, 2373, 1, 19,
36, 463, 3169, 2, 222, 3, 1016, 174, 20, 49, 808]
>>> data[0]
array([   4,    1,  346,   34,  806,    8,  116,   16,   38,    7,    7,
        249,    1,  362, 2148,    9,   53,   16,    3, 1653,  651,  274,
          7,    7,   18,    9,  215,  251,    5,   64,    8,   60,    1,
        455,    1,  924,  470,    5,    7,    7,  137,   35,  355,  800,
         47,    2,  358,  126, 1542,   15,    1,  357, 3332,    6, 1303,
         14,    6,  739, 7462,    2,   14,    3,    7,    7, 5710,   21,
          3,   84,   19,   18,  431,   32,  734,   28,   45,    7,    7,
        871,  112,  107,  893,  390, 1274,    9,    1,  372,   55,   42,
        616,   20, 1874,    7,    7,   39,  849,    1,   42,  287,    1,
        778], dtype=int32)
>>> data.shape
(25000, 100)
```

GloVe word embeddings

- Go to https://nlp.stanford.edu/projects/glove, and download the precomputed embeddings from 2014 English Wikipedia
  - ✓It's an 822 MB zip file called ***glove.6B.zip***, containing 100-dimensional embedding vectors for 400,000 words

Preprocessing the embeddings

- Parse .txt file to build an index that maps words (strings) to their vector representation (vectors)

```
glove_dir = '/var/containerDir/glove.6B’

embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs
f.close()


print('Found %s word vectors.' % len(embeddings_index))
```

Preprocessing the embeddings

- Build an embedding matrix (to load into Embedding layer)
  - ✓*(max_words, embedding_dim)*
  - ✓Each entry *i* contains *embedding_dim* dimensional vector

```
embedding_dim = 100
embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    if i < max_words:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
```

Words not found in the embedding index will be all zeros.

Preprocessing the embeddings

```
>>> embedding_matrix[0]
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
>>> embedding_matrix.shape
(10000, 100)
>>> embedding_matrix[1]
array([-0.038194  , -0.24487001,  0.72812003, -0.39961001,  0.083172  ,
        0.043953  , -0.39140999,  0.3344    , -0.57545   ,  0.087459  ,
        0.28786999, -0.06731   ,  0.30906001, -0.26383999, -0.13231   ,
       -0.20757   ,  0.33395001, -0.33848   , -0.31742999, -0.48335999,
        0.1464    , -0.37303999,  0.34577   ,  0.052041  ,  0.44946   ,
       -0.46970999,  0.02628   , -0.54154998, -0.15518001, -0.14106999,
       -0.039722  ,  0.28277001,  0.14393   ,  0.23464   , -0.31020999,
        0.086173  ,  0.20397   ,  0.52623999,  0.17163999, -0.082378  ,
       -0.71787   , -0.41531   ,  0.20334999, -0.12763   ,  0.41367   ,
        0.55186999,  0.57907999, -0.33476999, -0.36559001, -0.54856998,
       -0.062892  ,  0.26583999,  0.30204999,  0.99774998, -0.80480999,
       -3.0243001 ,  0.01254   , -0.36941999,  2.21670008,  0.72201002,
       -0.24978   ,  0.92136002,  0.034514  ,  0.46744999,  1.10790002,
       -0.19358   , -0.074575  ,  0.23353   , -0.052062  , -0.22044   ,
        0.057162  , -0.15806   , -0.30798   , -0.41624999,  0.37972   ,
        0.15006   , -0.53211999, -0.20550001, -1.25259995,  0.071624  ,
        0.70564997,  0.49744001, -0.42063001,  0.26148   , -1.53799999,
       -0.30223   , -0.073438  , -0.28312001,  0.37103999, -0.25217   ,
        0.016215  , -0.017099  , -0.38984001,  0.87423998, -0.72569001,
       -0.51058   , -0.52028   , -0.1459    ,  0.82779998,  0.27061999])
```

Defining a model

- Same model architecture as before

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

Loading the GloVe embeddings in the model

- Load the GloVe matrix into the Embedding layer, the first layer in the model

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

- Additionally, you'll freeze the Embedding layer (set its *trainable* attribute to False): the pretrained parts shouldn't be updated during training, to avoid forgetting what they already know
  - ✓The large gradient updates triggered by the randomly initialized layers would be disruptive to the already-learned features

Training and evaluation

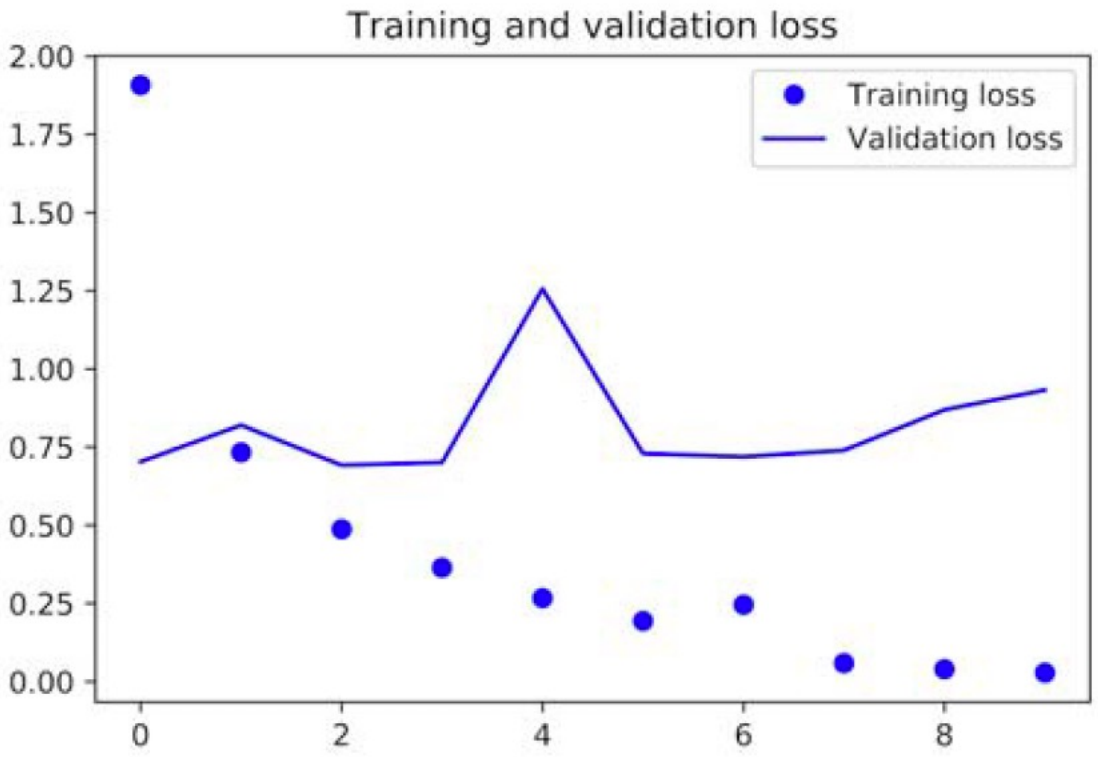
- Compile and train the model

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_val, y_val))
model.save_weights('pre_trained_glove_model.h5')
```

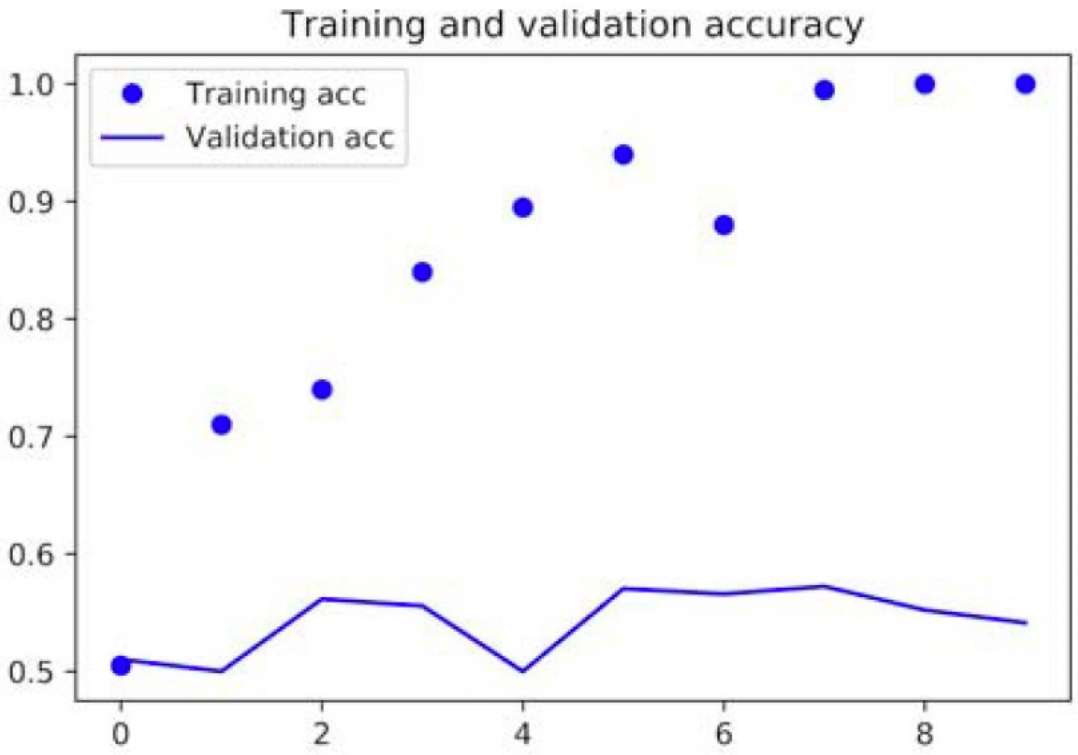
```
Train on 200 samples, validate on 10000 samples
Epoch 1/10
200/200 [==============================] - 1s 4ms/step - loss: 1.5178 - acc: 0.4700 - val_loss: 0.7167 - val_acc: 0.5035
Epoch 2/10
200/200 [==============================] - 0s 2ms/step - loss: 0.5915 - acc: 0.6600 - val_loss: 0.9005 - val_acc: 0.5012
Epoch 3/10
200/200 [==============================] - 0s 2ms/step - loss: 0.4659 - acc: 0.7900 - val_loss: 0.6983 - val_acc: 0.5463
Epoch 4/10
200/200 [==============================] - 0s 2ms/step - loss: 0.4021 - acc: 0.8350 - val_loss: 0.7451 - val_acc: 0.5215
Epoch 5/10
200/200 [==============================] - 0s 2ms/step - loss: 0.4039 - acc: 0.8150 - val_loss: 0.7970 - val_acc: 0.5303
Epoch 6/10
200/200 [==============================] - 0s 2ms/step - loss: 0.1785 - acc: 0.9800 - val_loss: 1.1030 - val_acc: 0.5006
Epoch 7/10
200/200 [==============================] - 0s 2ms/step - loss: 0.1476 - acc: 0.9650 - val_loss: 0.8004 - val_acc: 0.5574
Epoch 8/10
200/200 [==============================] - 0s 2ms/step - loss: 0.0857 - acc: 0.9850 - val_loss: 2.2700 - val_acc: 0.5021
Epoch 9/10
200/200 [==============================] - 0s 2ms/step - loss: 0.2508 - acc: 0.8500 - val_loss: 0.7580 - val_acc: 0.5713
Epoch 10/10
200/200 [==============================] - 0s 2ms/step - loss: 0.0318 - acc: 1.0000 - val_loss: 0.7734 - val_acc: 0.5694
```

Plotting the results

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```



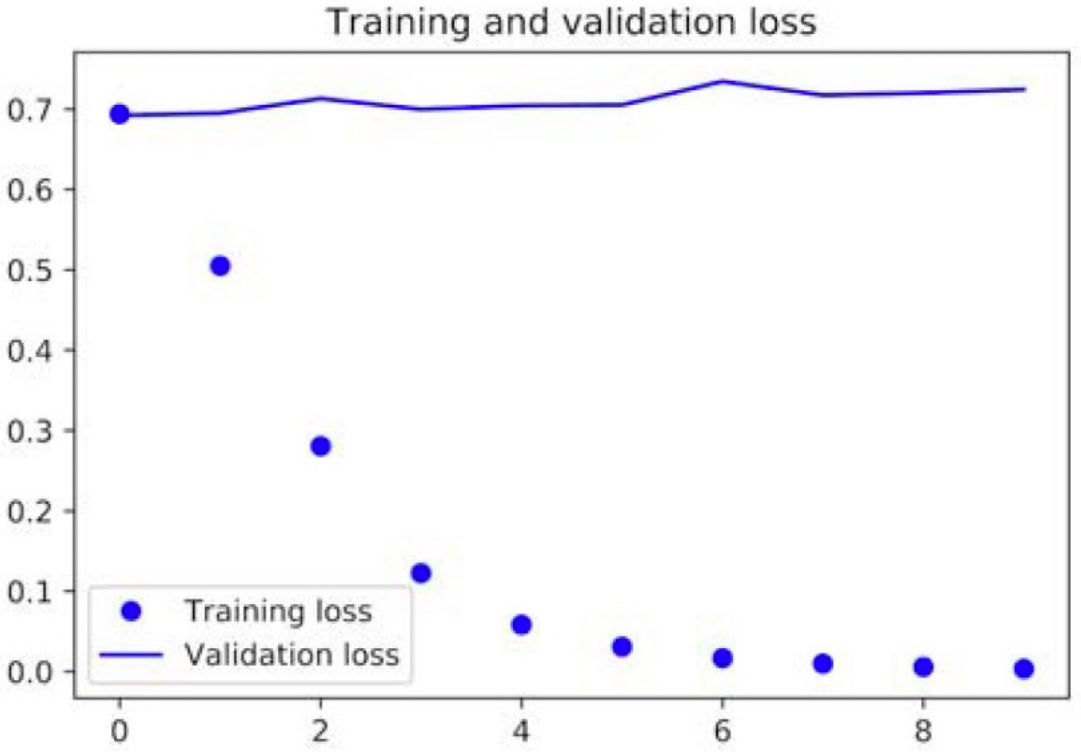
Plotting the results



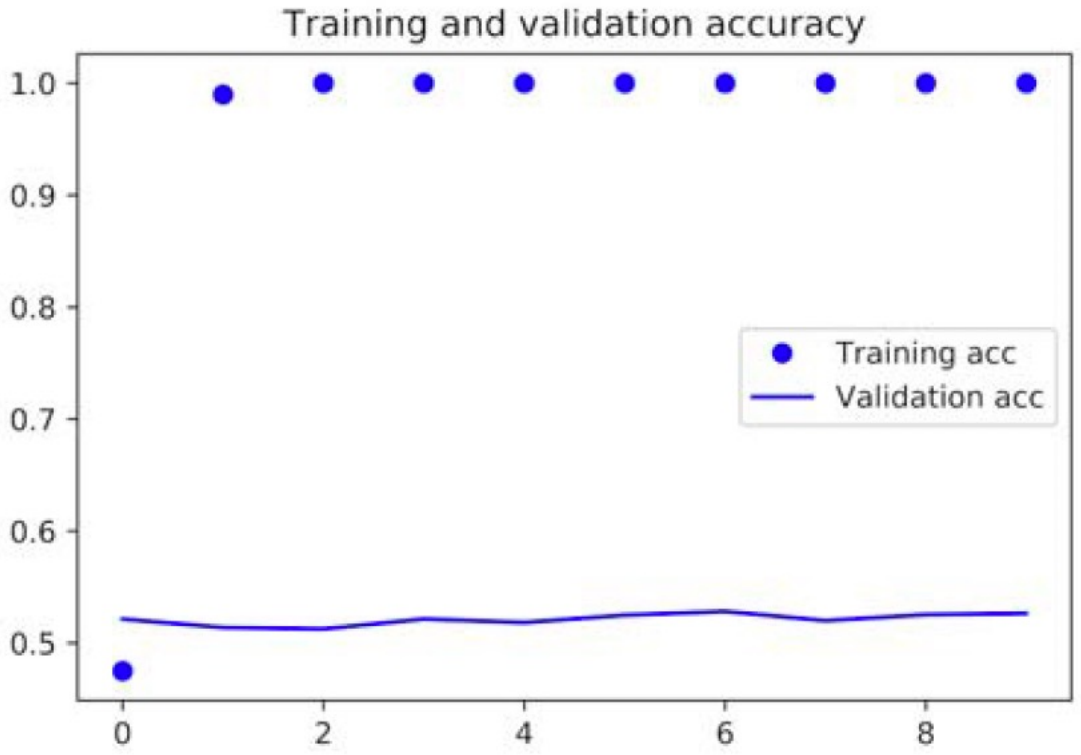
- The model quickly starts overfitting, which is unsurprising given the small number of training samples
- Validation accuracy has high variance, but it seems to reach the high 50s

Training the same model without pretrained word embeddings

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_data=(x_val, y_val))
```



Training the same model without pretrained word embeddings



- Validation accuracy stalls in the low 50s
- So in this case, pretrained word embeddings outperform jointly learned embeddings
- If you increase the number of training samples, this will quickly stop being the case

Tokenizing test data set

```
test_dir = os.path.join(imdb_dir, 'test')
labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(test_dir, label_type)
    for fname in sorted(os.listdir(dir_name)):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)

sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen=maxlen)
y_test = np.asarray(labels)
```

Evaluating model on test set

```
model.load_weights('pre_trained_glove_model.h5')
model.evaluate(x_test, y_test)
```

- Test accuracy = 56%
- Working with just a handful of training samples is difficult!