# Sequential data

## Preparing

- **Get the IMDB dataset as a csv**

## Real language data

- Most real language data is sequential
- So what?
- Consider what we might do in order to classify (as this is all we have been doing up til now) language data: for example (to keep things simple), mapping sentences to **grammatical** or **ungrammatical**
- **Possibility 1**: Sentences are sequences of words: find a way of encoding all the words in a sentence as a collection of variables (e.g., suppose we have a ten dimensional, i.e., ten variable, representation of each word: include all the variables for all of the words)
- *Problem*: Not all sentences have the same number of words; possibly (depending on how we encode the words) it could be a problem that the same relationship between words can occur in different positions in the sentence
- **Possibility 1.1**: combine, a.k.a., "pool" all the words together into one mega-word
- *Problem*: Losing information
- **Possibility 2**: Deal with only one (or $k$) words at a time
- *Problem*: The models we have been discussing assume that the distribution over observations is constant; the whole issue is that the words in a sentence are **not** independent of one another: whether a word is grammatical depends on what came before it
- **Possibility 1.1.1**: look at some window of $k > 1$ words at a time, then pool **windows** together; inherits all the problems but maybe mitigates them
- We need some more serious way of dealing with **time**, or, if you don't want to think about such physical concepts, **sequential data** - where the distribution (in our case, the classification problem) changes depending on what data came before

- Let's try and work through some real-life examples just by using one of the above possibilities (except for number 1)
- Here I'll walk through a text processing example
- Then, when we come to *actual* sequential models (i.e., models that have some way of genuinely treating the whole sequence as-is as a series of non-independent observations), you'll have a better feel for what is going on

## The IMDB Dataset

- The book uses a dataset of reviews on IMDB, which are marked as either positive or negative sentiment
- **Read in the CSV and look at the head**
- Most of data science is data preprocessing
- What might cause problems with this data?

```
def process(x):
    x = re.sub('[,\.!?:()"]', '', x)
    x = re.sub('<.*?>', ' ', x)
    x = re.sub('http\S+', ' ', x)
    x = re.sub('[^a-zA-Z0-9]', ' ', x)
    x = re.sub('\s+', ' ', x)
    return x.lower().strip()

reviews = [process(x) for x in imdb['review'].tolist()]
```

- Now: how am I going to represent words?
- We have a couple of obvious choices
- We could create big vectors with dimension equal to the size of the set of all (or most) distinct tokens in the data set, which have one dimension per word, and which are 1 if and only if the word is that specific word (i.e. the word corresponding to the given dimension)
- This is surprisingly non-hard to deal with
- This is called a **one-hot encoding**
- However, to make our lives more non-trivial, let's instead use **pre-trained word vectors**
- Word vector training typically happens roughly as follows: (1) take a one-hot representation like the one I just said; (2) train some lower-dimensional representation to predict adjacent words
- I tracked down some pre-trained Glove vectors, and I've made them available as a resource in the project folder

```
glove = {}
with open("glove_imdb.6B.50d.txt") as hf:
    for line in hf:
        word = line.split()[0]
        vector = np.array(line.split()[1:],
                          dtype="float16")
        glove[word] = vector

reviews_glove = []
for review in reviews:
    review_words = review.split()
    review_glove_words = []
    for word in review_words:
        try:
            review_glove_words.append(glove[word])
        except KeyError:
            continue
    review_glove = np.concatenate(review_glove_words).reshape((-1,50))
    reviews_glove.append(review_glove)
```

## Ignoring time by pooling

```
reviews_glove_pooled_l = []
for review in reviews_glove:
    reviews_glove_pooled_l.append(np.average(review, axis=0))
reviews_glove_pooled = np.concatenate(reviews_glove_pooled_l).reshape(-1, 50)
```

- As before we can train a model (logistic regression or otherwise) using Torch
- On a logistic regression I get about 75% accuracy, which is not bad - the state of the art is around 90%, but I have done something extremely simple

## Ignoring time by treating words as independent observations

- You can try it, but you will regret it
- It sucks
- It also will probably eat up a lot of memory, because that means loading a lot of words in memory at once
- Let's skip this for now

## Not completely ignoring time by windowing

- Let's look at a dumb middle ground between these two extremes of dumbness (averaging **all** the words together and averaging **none** of the words together)
- What if we average words together within a **window** of a few words?

- That is to say, we want to make sure we have a chance of finding enough words that, together or separately, they tell us something about the sentiment
- But we hope that, if we don't smush **all** the words together, we might be able to do better
- So, for each document, we will have several 50-dimensional average word vectors
- We can then combine them into one single mega-vector per document
- Now, we can't say, average every five words together - the documents vary in length, and we need a fixed number of inputs
- But we can split into an equal number of parts per document - say five
- This is often what's done for measuring vowels: we know that vowels have dynamics, so, rather than measure formants at the midpoint, we look at several measurements together - perhaps at the first, second and third quartile
- Another alternative (more like our second approach) would be to cut the reviews into sub-documents, not of one word, but of four or five words each - I'll leave that up to you to try out

```
reviews_glove_windowed_l = []
n_windows_per_doc = 10
for review in reviews_glove:
    if len(review) < n_windows_per_doc:
        review = np.pad(review,
            ((0, n_windows_per_doc - len(review)),
            (0,0)))
    words_per_window = len(review)//n_windows_per_doc
    windows = []
    for i in range(n_windows_per_doc):
        start = i*words_per_window
        end = start + words_per_window
        windows.append(
            np.average(review[start:end,:],
            axis=0))
    reviews_glove_windowed_l.append(np.concatenate(windows))
reviews_glove_windowed = np.concatenate(
    reviews_glove_windowed_l).reshape(
        -1, n_windows_per_doc*50)
```