GRAND RAPIDS BIG DATA MEETUP

JULY 2016

Predictive Models With Text Input

Jeff Allard
Data Scientist 5/3 Bank

Focus

Practical introduction to using textual data as input for Classification and Regression Models….

- Numbers [1000.32,142, 47] : As is, no problem!
- Categories [‘A’, ‘B’, ‘C’] : Create 0/1 dummy codes, easy!
  - Example: if we have an instance with ‘A’ then two variables can be created cat_A = 1 and cat_b =0
- Text [“Your product looked really nice on the webpage but crap once I got it home”] : Hmmm?

Content

- Classic Technique (but still works well)
  - Term Frequency (tf)
  - Term Frequency Inverse Document Frequency (tfidf)
  - Classic Technique Models (Regularized) Linear models

Note: lots of other variations exist (e.g. BM25)

- Newer Classical Techniques (matrix factorization)
  - SVD / LSI / NNMF / etc.

- Modern Techniques
  - Word embeddings
    - Word2Vec
    - Doc2Vec
  - Deep learning Recurrent Neural Networks

Hot!!

- Python Demo (Sentiment Analysis)

The Classics

Corpus (3-documents):
'I like data science it is cool',
'Data science is like really cool, really cool',
'Python is cool for data'

Binary Word Vectors:

- One-hot-encode presence or absence of each term
- Number of predictors >= Size of vocabulary
  - Reduced by pruning stop words, stemming, very infrequent and very frequent words
  - Increased dramatically with n-grams (“I like”, “like data”, “data science”…..)
- Typically stored in a sparse matrix

Notice, order is lost

Document	cool	data	for	is	it	like	python	really	science
1	1	1	0	1	1	1	0	0	1
2	1	1	0	1	0	1	0	1	1
3	1	1	1	1	0	0	1	0	0

The Classics

Corpus (3 documents):
'I like data science it is cool',
'Data science is like really cool, really cool',
'Python is cool for data'

Term Frequency (TF) Word Vectors:

- Count frequency of each term
- Number of predictors >= Size of vocabulary
  - Reduced by pruning stop words, stemming, very infrequent and very frequent words
  - Increased dramatically with n-grams (“I like”, “like data”, “data science”…..)
- Typically stored in a sparse matrix

Document	cool	data	for	is	it	like	python	really	science
1	1	1	0	1	1	1	0	0	1
2	2	1	0	1	0	1	0	2	1
3	1	1	1	1	0	0	1	0	0

The Classics

Corpus (3 documents):
'I like data science it is cool',
'Data science is like really cool, really cool',
'Python is cool for data'

Term Frequency Inverse Document Frequency Vectors:

- Same as TF but count frequency of each term weighted by the relative popularity of the term in the corpus
- TFIDF = TF*Log(N/D+1) where N is the number of documents and D is the number of documents within which the term exists
- Down weights terms that occur in a lot of documents, because even if they are frequent in a given document, it doesn't likely mean much (e.g. "account" in a banking corpus). Same logic for up weighting.

- 'python' gets up-weighted for the 3rd document because it only occurs in 1 document, so there is a better chance it is important to the content / description of the document: *1*(np.log(3/1)+1) = 2.098612
- 'is' gets down-weighted to 1 since occurs in each document

Document	cool	data	for	is	it	like	python	really	science
1	1	1	0	1	2.099	1.41	0	0	1.41
2	2	1	0	1	0	1.41	0	4.2	1.41
3	1	1	2.099	1	0	0	2.099	0	0

Recap of Basic Method

- Text is pre-processed
  - Text is broken up into token (normally words, some applications use characters)
  - Generic stop words removed (e.g. ‘and’, ‘is’)
  - Domain specific stop words removed (e.g. ‘account’ in banking)
  - Stemming may take place (e.g. running ->run)
  - N-grams created (2-gram, 3-gram, n-gram, skip grams)
  - Very frequent and very infrequent tokens removed
- Large sparse matrix created with values as Boolean, count or otherwise weighted values

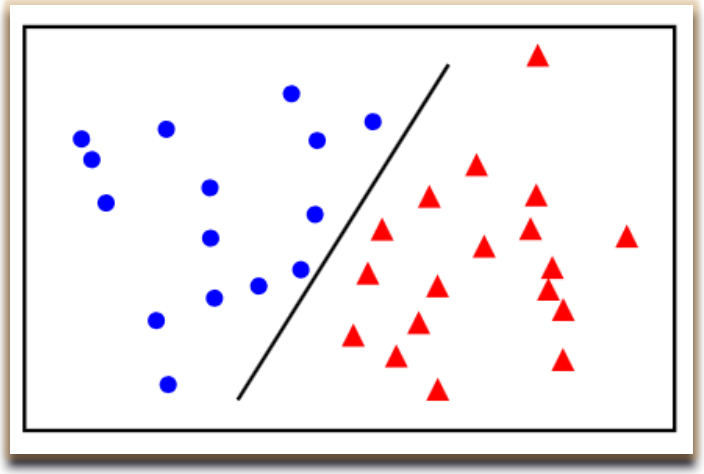
Python

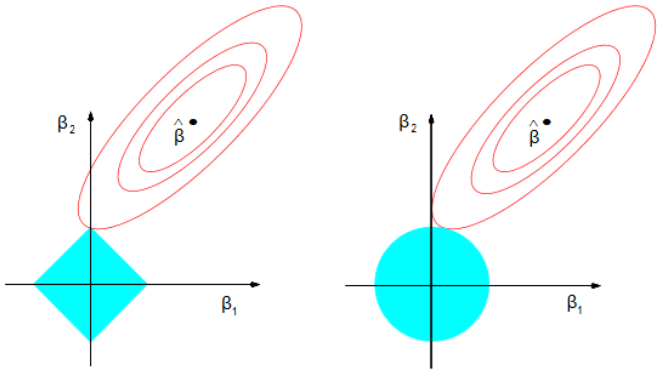
```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer

corpus=['I like data science it is cool',
        'Data science is like really cool, really cool',
        'Python is cool for data']

#default settings (split on whitespace, no stop words, 1 grams, etc)
cv=CountVectorizer()
tf_matrix=cv.fit_transform(corpus) #sparse matrix
print 'Term Freq..\n'
print 'type: ',type(tf_matrix)

print '\nDense term frequency matrix..'
print pd.DataFrame(tf_matrix.toarray(),columns= cv.get_feature_names()) #dense



print '\nTerm Freq Inverse Document Fequency...'

tfidf=TfidfTransformer(norm=None, use_idf=True, smooth_idf=False, sublinear_tf=False) #no smoothing

tfidf_matrix=tfidf.fit_transform(tf_matrix)

print '\nDense tfidf matrix..'
print pd.DataFrame(tfidf_matrix.toarray(),columns= cv.get_feature_names()) #dense
```

```
Term Freq..

type:  <class 'scipy.sparse.csr.csr_matrix'>

Dense term frequency matrix..
   cool  data  for  is  it  like  python  really  science
0     1     1    0   1   1     1       0       0        1
1     2     1    0   1   0     1       0       2        1
2     1     1    1   1   0     0       1       0        0

Term Freq Inverse Document Fequency...

Dense tfidf matrix..
   cool  data       for   is        it      like    python    really   science
0   1.0   1.0  0.000000  1.0  2.098612  1.405465  0.000000  0.000000  1.405465
1   2.0   1.0  0.000000  1.0  0.000000  1.405465  0.000000  4.197225  1.405465
2   1.0   1.0  2.098612  1.0  0.000000  0.000000  2.098612  0.000000  0.000000
```

Models for the classics

- Given the enormous number of variables, often the best model to use is a linear model
  - E.g. linear/logistic regression or linear support vector machine
- Needs to be fast to train and score new data
- Input is the document x term matrix
- Always want to use a linear model with regularization to control overfitting
  - L1 or L2 (or combination) penalty for the coefficients getting large
  - Lasso and ridge



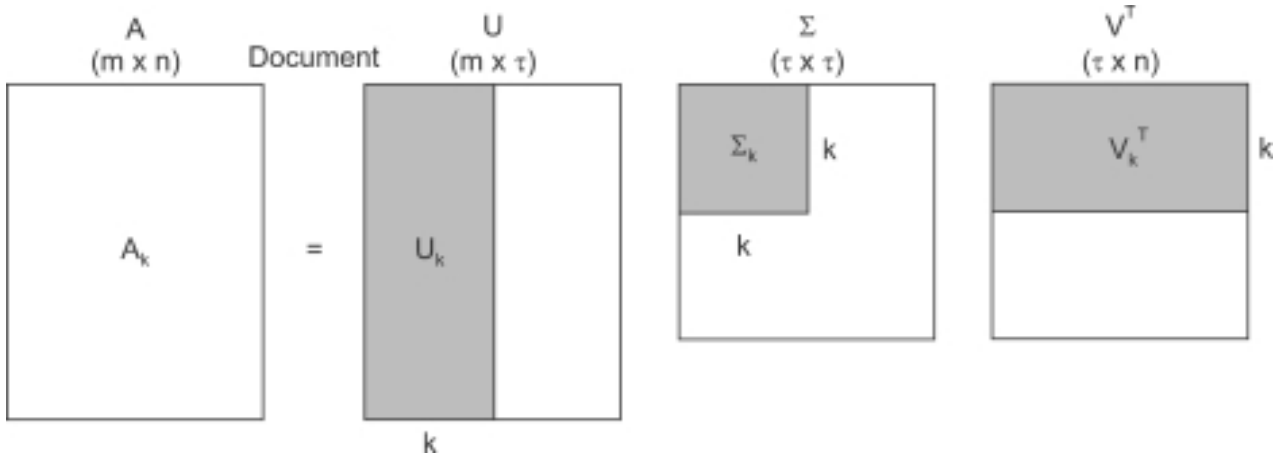


Newer Classics

- Start with one of the classics from prior slides (often TFIDF)
- Use some sort of matrix factorization to…
  - Reduce dimensionality
  - De-noise
  - Discover latent variables / topics / themes…

Latent Semantic Indexing

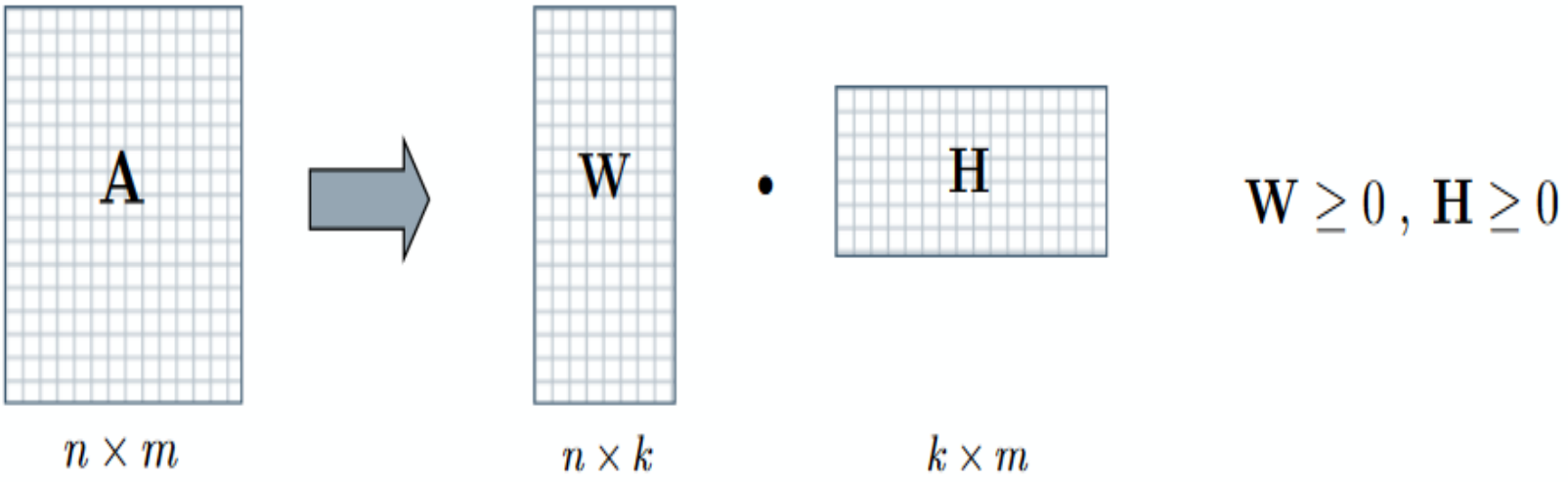
- Classical information retrieval technique (query / document similarity)
- LSI uses a singular value decomposition (SVD) to factor a matrix $A = U\Sigma V^T$
- Some number of dimensions (k) is retained:



- M documents and n variables from the prior slides
- Factor matrix A using SVD
- Retain first k columns of U as the new input or $U_k\Sigma_k$

Non Negative Matrix Factorization

- Very similar but constrained to have non-negative entries
- One of popular techniques for “Topic Modeling” or discovering common topics in large text documents
- If A is documents (n) by terms (m) matrix, use the W matrix as input to the model (columns are latent features)



Source: http://derekgreene.com/slides/nmf_insight_workshop.pdf

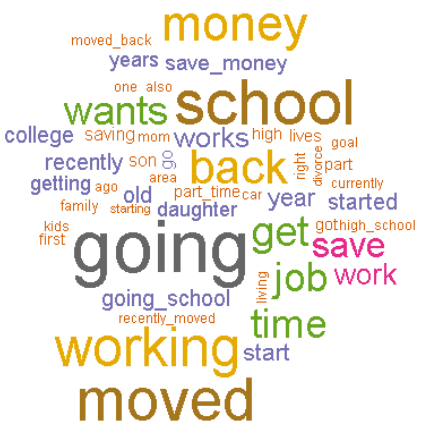
Non Negative Matrix Factorization (topic models diversion)

In [12]:
```
model = decomposition.NMF(init="nndsvd", n_components=50, max_iter=1000)
W = model.fit_transform(A)
H = model.components_    #components are the generic name here for topics

W = W / np.sum(W, axis=1, keepdims=True)  #normalize the document x topics matrix so summs to 1
```

In [13]:
```
print A.shape #document x terms
print W.shape #document x topics (topic distribution for each document)
print H.shape #topic x  terms (term distribution for each topic)
print "Generated factor W of size %s which represents (documents x topics)  \n
       and factor H of size %s which represents (topics x terms)" % ( str(W.shape), str(H.shape) )
```

```
(207723, 218964)
(207723L, 50L)
(50L, 218964L)
Generated factor W of size (207723L, 50L) which represents (documents x topics) and factor H of size
(50L, 218964L) which represents (topics x terms)
```

Example topics from FNA



Models Newer Classics

- Given a (much) smaller dimensionality of the input matrix, any machine learning algorithm can now be used
  - Linear models
  - Decision trees
  - Nearest neighbor
  - Neural networks
  - etc.

Modern Techniques – word embeddings

- Changed the world of natural language processing a couple of years ago
- The idea is that words (or generically 'symbols' e.g. anything encoded in sequences like genomic data) are represented not as one-hot vectors but dense vectors
- How to represent the word 'data' to a computer?

One-hot:

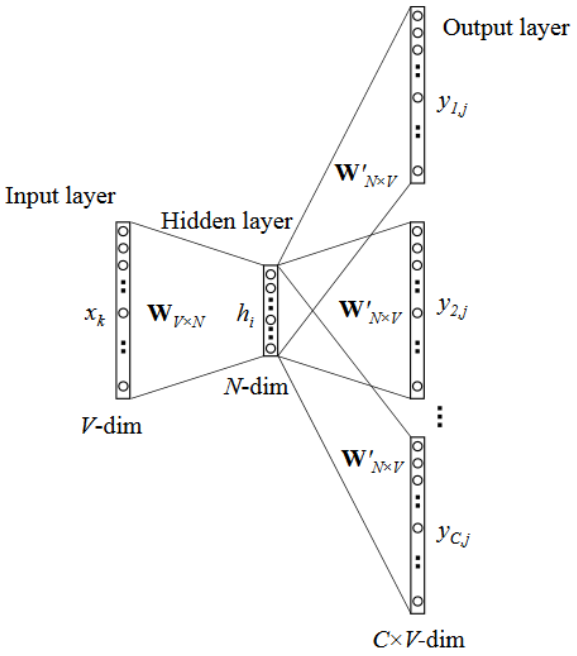
[0,0,0,0,0,…..1…0,00,0] (length of vocabulary n-grams, e.g. 100,000)

Word-vector:

[0.245, −0.126, 0.875 ….0.743] (length of embedding, e.g. 300)

Word2Vec

- A family of models, developed at Google (2013)
- Lots of details, lots of great papers and tutorials out there
- One way ('skipgram') to think about it is as a shallow neural network where weights are learned to predict which words exists in a context around a given word



- Take a window across text of a given size (e.g. 1 for illustration) on both sides of each term

[I really like doing data science]

  - Input: [really]
  - Output: [I,like]
  - Input: [like]
  - Output: [really, doing]
  - ……

- What results are vectors (W and W' to the left) for each word in our vocabulary (e.g. 300 length)
- Words used in similar context should have similar vectors

Source: www-personal.umich.edu/~ronxin/pdf/w2vexp.pdf

Word2Vec

- Now each word in the vocabulary is represented as a (say) 300 dimensional vector – perhaps each dimension is a latent dimension that describes something in your corpus?

Recall: Fixed vector with all zeros and a single “1”

- If we one-hot-encode ‘dog’ and ‘puppy’ they have no more similarity than ‘man’ and ‘air-freshener’. Same for ‘dog’ and ‘pet’ although these are often used in the same context
- But with Word2Vec, they will be very similar vectors*

*in the cosine similarity sense

Word2Vec

- What was amazing and powerful about this is that people found that not only were words used in similar context similar in their vector space but that you could do addition and subtraction on the vectors and the results made sense for analogies
- Example:

Vector1 =vec_King -vec_Man + vec_Woman

Which word vector will vector1 be closest to, if you check each one from a very large corpus?

vec_Queen

Word2Vec

- We still have the problem of how to represent text (e.g. sentences, reviews, strings of symbols).

- One option is just to average or otherwise take a weighted sum of the word vectors –
  - E.g. for the review [Your product was really great], we would average the 5 word vectors and use the 300 dimensional vector with a classification model
  - Works well for shorter text (e.g. tweets)
  - Tends to fail for larger text as the averaging loses information, context etc.
- Two other options we will discuss:
  - Doc2Vec (aka paragraph2vec)
  - Recurrent Neural Networks (RNN)

Doc2Vec

- Similar to word2vec but at the document level
- Whereas word2vec learned correlation between words, doc2vec adds IDs to each document (unique for each document or perhaps multiple IDs) and learns correlation between document labels and words as well
- With Doc2Vec each document gets a single document vector associated with it after training which can be used as input to a predictive model
- Often works better than averaging the word2vec vectors

Doc2Vec

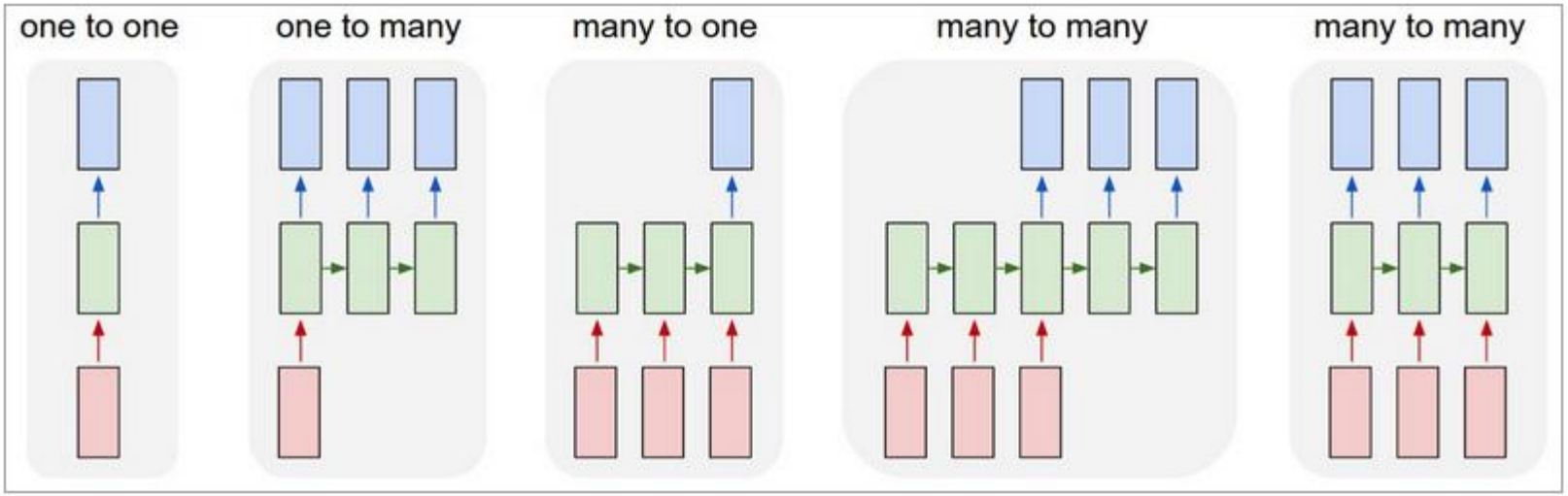
```
In [7]: for model in models:
            print(str(model))
            pprint(model.docvecs.most_similar(positive=["Machine learning"], topn=20))
```

```
Doc2Vec(dbow+w,d200,hs,w5,mc5,t8)
[('Artificial neural network', 0.7255396246910095),
 ('Theoretical computer science', 0.7125768661499023),
 ('Data mining', 0.6895816326141357),
 ('Pattern recognition', 0.678891658782959),
 ('List of important publications in computer science', 0.6695302724838257),
 ('Outline of computer science', 0.667578935623169),
 ('Information visualization', 0.6667760014533997),
 ('Unsupervised learning', 0.6627277135848999),
 ('Bayesian network', 0.6622973680496216),
 ('Support vector machine', 0.6594343781471252),
 ('Algorithmic composition', 0.6593101024627686),
 ("Solomonoff's theory of inductive inference", 0.6554585695266724),
 ('Kriging', 0.6505937576293945),
 ('Model checking', 0.6501827239990234),
 ('Information theory', 0.6447420120239258),
 ('Computational learning theory', 0.6422973871231079),
 ('Generalization error', 0.6414266228675842),
 ('Complexity', 0.6391021609306335),
 ('Glossary of artificial intelligence', 0.6353012323379517),
 ('Theory of computation', 0.6329255104064941)]
Doc2Vec(dm/m,d200,hs,w5,mc5,t8)
[('Theoretical computer science', 0.6481858491897583),
 ('Unsupervised learning', 0.6421648859977722),
 ('Artificial neural network', 0.637227475643158),
 ('Data stream mining', 0.6325934529304504),
 ('Pattern recognition', 0.6199989318847656),
 ('Outline of computer science', 0.6175510287284851),
 ('Deep learning', 0.6125383377075195),
 ('Algorithmic learning theory', 0.608863353729248),
 ('Statistical learning theory', 0.5993553400039673),
 ('Feature learning', 0.5990031957626343),
 ('Generalization error', 0.5920165777206421),
 ('Cognitive architecture', 0.5919679403305054),
 ('Reinforcement learning', 0.5869054794311523),
 ('Supervised learning', 0.5860797166824341),
 ('Cluster analysis', 0.5855995416641235),
 ('Data mining', 0.5830211639404297),
 ('Complexity', 0.5810916423797607),
 ('Decision tree', 0.5741485953330994),
 ('Kriging', 0.5734224915504456),
 ('Statistical inference', 0.5728234052658081)]
```

Useful for search and recommendation engine applications such as finding similar web pages, products based on description text etc.

RNN

- Deep learning for sequences
- Instead of a flat matrix input as for most all other ML algorithms, RNNs use sequence input to predict other sequences
- Many flavors for all kinds of NLP tasks

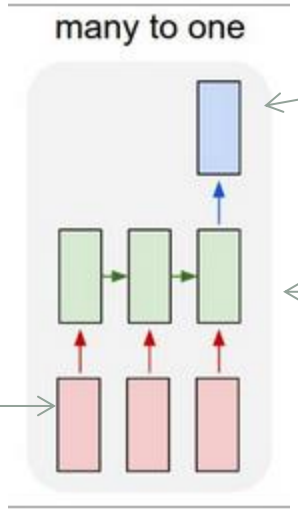


Source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

RNN

- For sentiment analysis, the neural network reads in a sequence of elements (e.g. word vectors from word2vec) and predicts a single outcome (the sentiment label)

Each time-step is a word (vector) in the review, sequentially such as the vector for “loved”, the vector for “product”, the vector for “great”



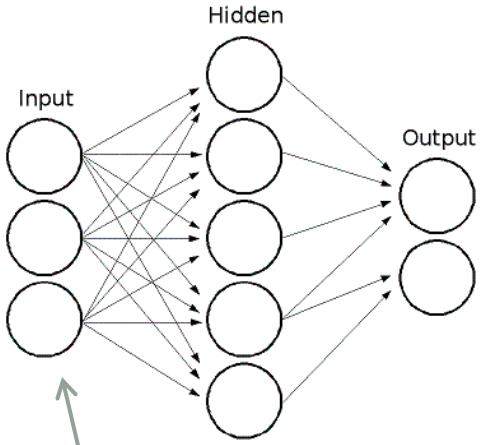
Represents a single training example

Outputs a vector to be used by another layer of a net

“Hidden vector” weights of the net that are learned

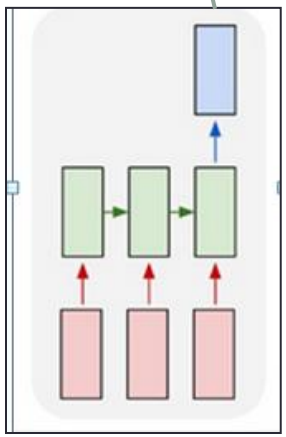
Appendix - LSTM

Idea of the model created in the demo



Next, the 400 dimensional vector from the LSTM layer is connected to a “normal” neural network

- “Input” is 400 dimensional
- “Hidden” is 50 dimensional
- “Output” is the probability that the review is positive or negative



At the end, the 400 dimensional hidden vector is output

Each green box is an LSTM cell. It updates the “hidden” vector of size 400 (we could pick any). The hidden vector and the word2vec vectors are being optimized as the network learns

Up to 50 steps, where each step corresponds to a word from a review in order. Each word is represented by its Word2Vec vector of size 400

Python

DEMO

- Regularized logistic regression on TFIDF matrix
- Regularized logistic regression on averaged word2vec word vectors
- LSTM (Long Short Term Memory) RNN