Software

Word2Vec

Recall: Creating Numerical Features from Text

Code

```
import pandas as pd
from sklearn.feature_extraction.text
     import CountVectorizer

corpus = ['This is the first document.',
          'This is the second
    document.',
          'And the third one.']

cv = CountVectorizer()
X = cv.fit_transform(corpus)
pd.DataFrame(X.toarray(),
         columns=cv.get_feature_names())
```

Output

	and	document	first	is	one	second	the	third	this
0	0	1	1	1	0	0	1	0	1
1	0	1	0	1	0	1	1	0	1
2	1	0	0	0	1	0	1	1	0

Count Vectorizer

Word/Document Vectors with CountVectorizer

Document Vectors

- Doc 0: [0, 1, 1, 1, 0, 0, 1, 0, 1]
- Doc 1: [0, 1, 0, 1, 0, 1, 1, 0, 1]
- Doc 2: [1, 0, 0, 0, 1, 0, 1, 1, 0]

	and	document	first	is	one	second	the	third	this
0	0	1	1	1	0	0	1	0	1
1	0	1	0	1	0	1	1	0	1
2	1	0	0	0	1	0	1	1	0

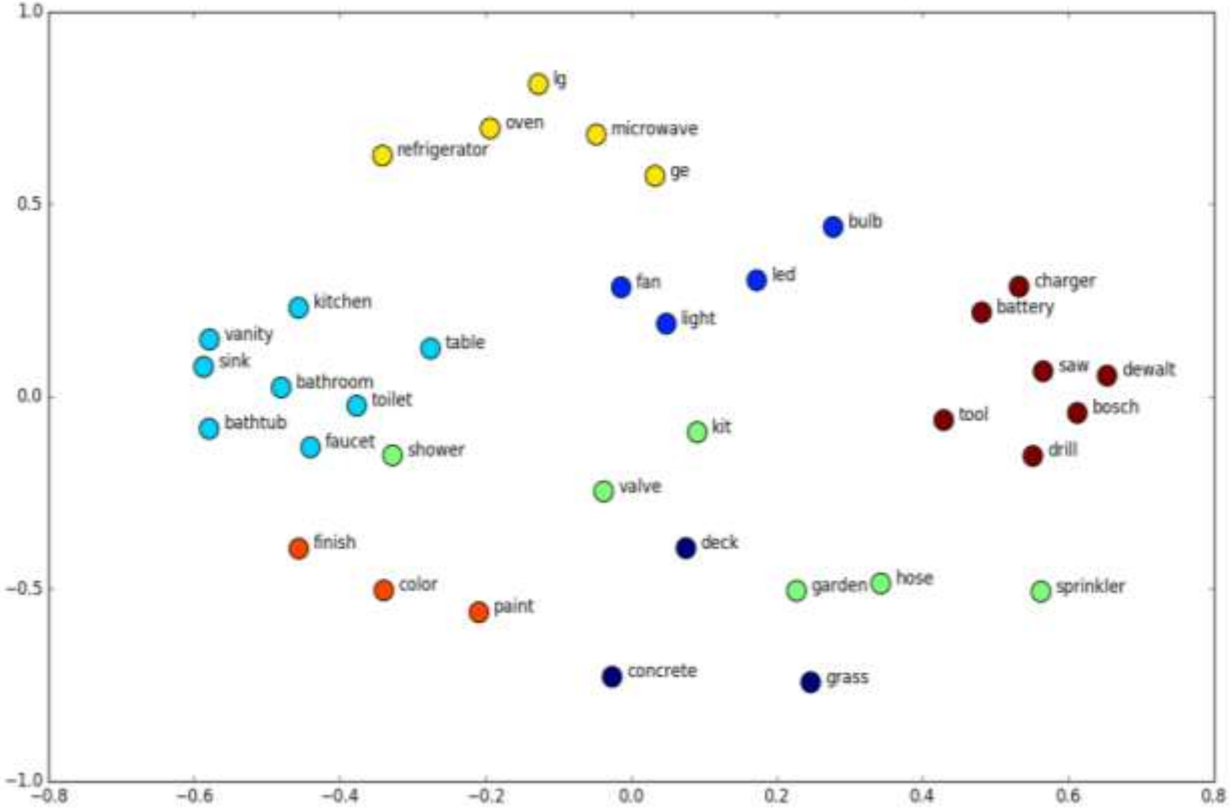
Flip it around → Word Vectors

- and: [0, 0, 1]
- document: [1, 1, 0]
- first: [1, 0, 0]
- is: [1, 1, 0]
- one: [0, 0, 1]
- second: [0, 1, 0]
- the: [1, 1, 1]
- third: [0, 0, 1]
- this: [1, 1, 0]

	0	1	2
and	0	0	1
document	1	1	0
first	1	0	0
is	1	1	0
one	0	0	1
second	0	1	0
the	1	1	1
third	0	0	1
this	1	1	0

Why Word Vectors?

- Represent Conceptual "meaning" of words
- Word vectors close to each other have similar meaning

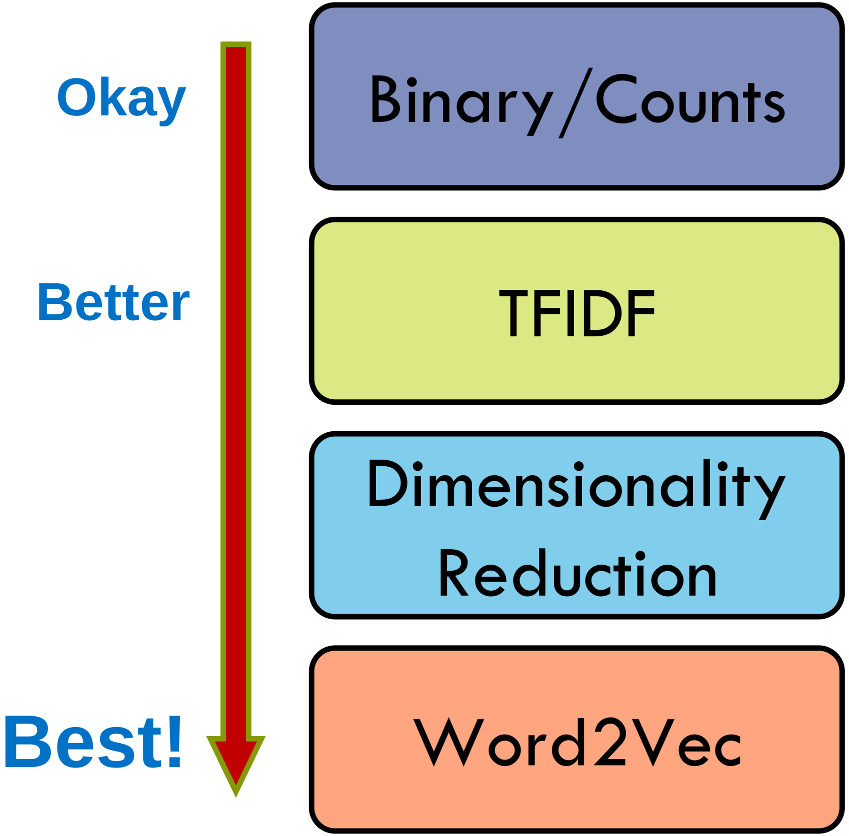


How to Use Word Vectors?

- Information Retrieval
  - e.g. conceptual search queries, concepts related to “painting”
- Document Vectors
  - A document vector is the average of its word vectors
- Machine Learning
  - Document Classification (from document vectors)
  - Document Clustering
- Recommendation
  - Recommend similar documents to search query or sample documents

Can we do better than counts?

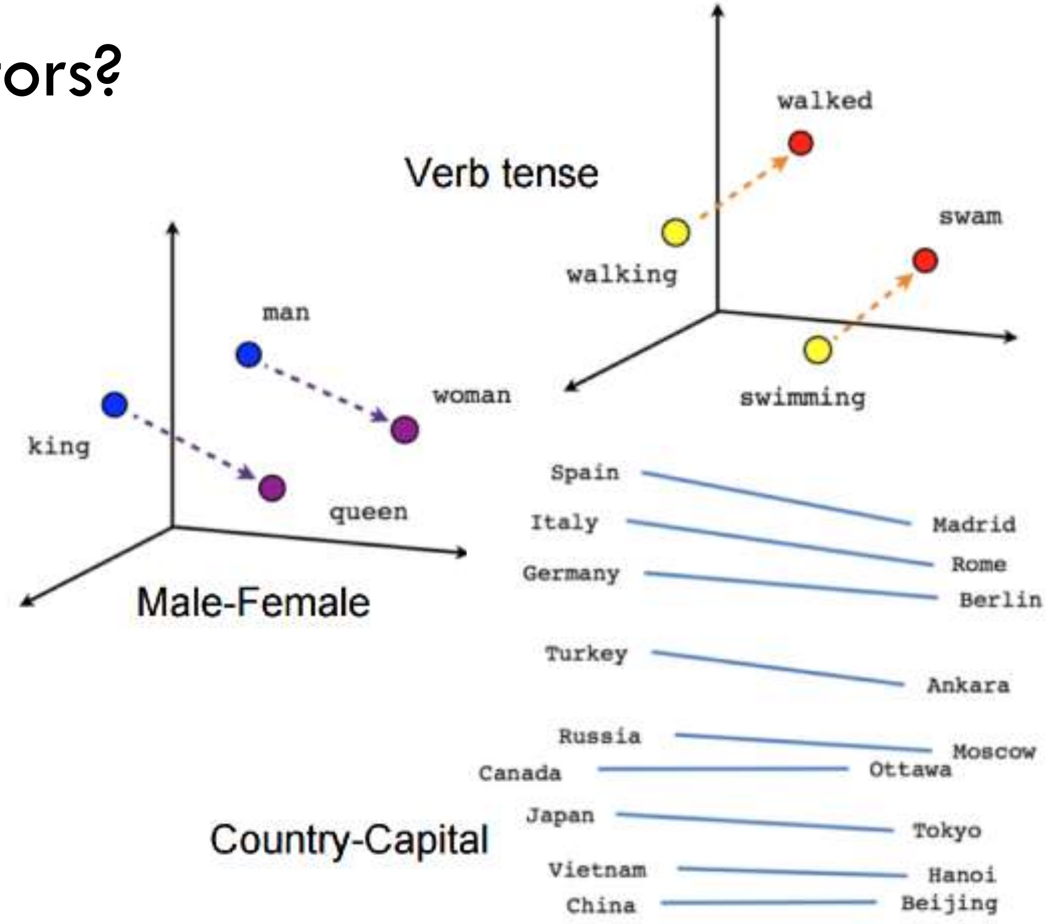
- Answer: YES!
- Problems with counts:
  - Limited information
    - Possible Resolution: TFIDF
  - Vectors HUGE for many documents
    - Possible Resolution: Matrix Factorization
      - Bag of Words → No Word Order
    - Possible Resolution:
      Neural Networks → Word2vec!



How to Use Word Vectors?

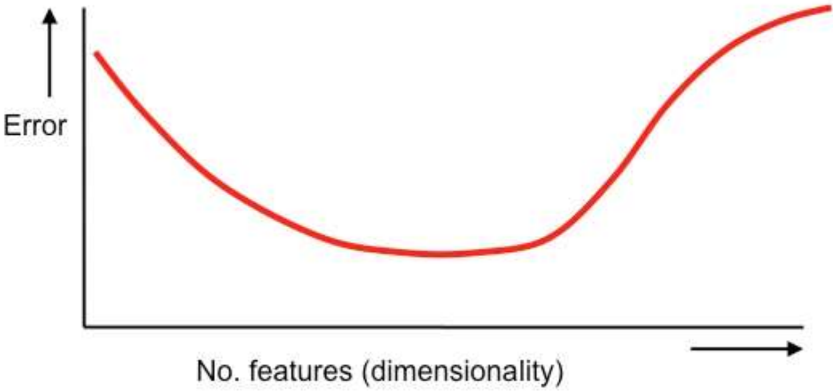
- Answer: Comparability with human intuition

- Standard Baseline Tests:
  - Analogies
  - Ratings of Word Similarity
  - Verb Tenses
  - Country-Capital Relationships

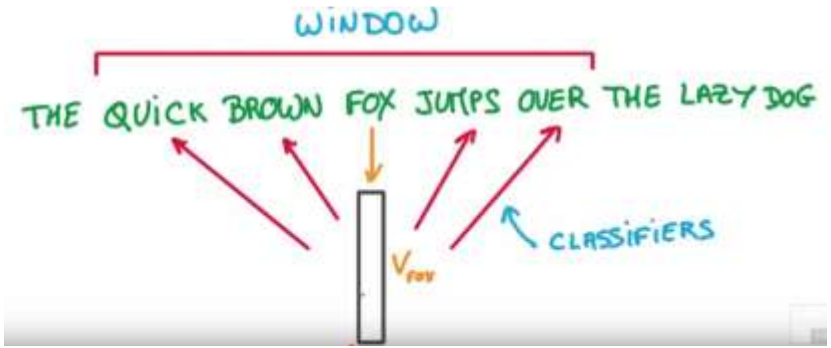


Finding Better Word Vectors: Word2Vec

- Problem: Count vectors far too large for many documents.
  - Solution: Word2Vec reduces number of dimensions (configurable e.g. 300)



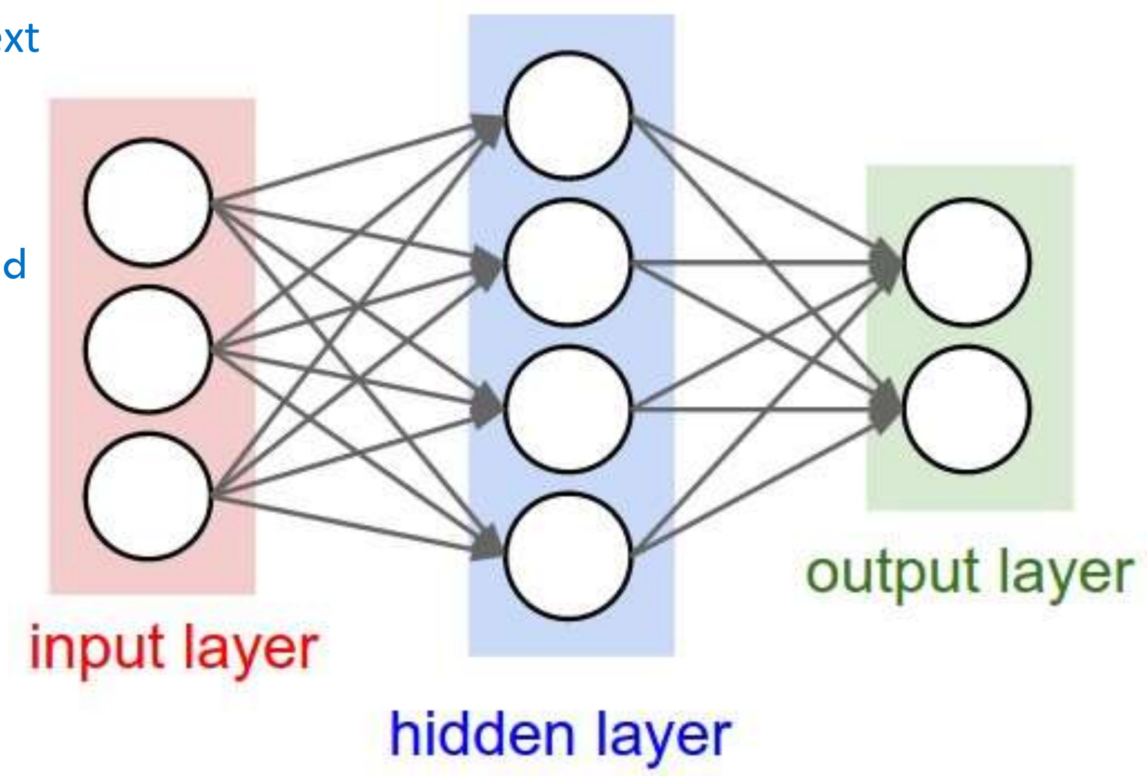
- Problem: Bag of Words neglects word order.
  - (Partial) Solution: Word2Vec trains on small sequences of text ("context windows")



Training Word2Vec

- Use a Neural Network on Context Windows
- 2 main approaches for inputs and labels:
  - Skip-Grams
  - Continuous Bag of Words (CBOW)

Vectors usually similar, subtle differences, also differences in computational time

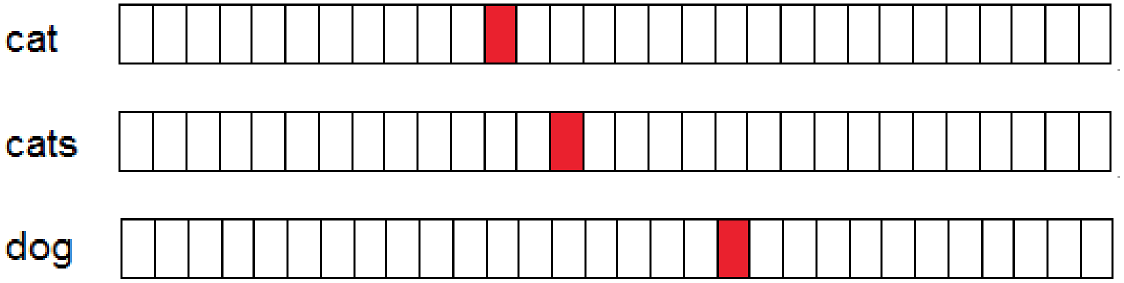


Training Word2Vec: Context Windows

- Input Layer: Context Windows
- Observations for word2vec: All context windows in a corpus
- Context window size determines size of relevant window around word:
  - e.g.: Document: "The quick brown fox jumped over the lazy dog."
  - Window size: 4, target word "fox".
    - Window 1: "The quick brown fox jumped over the lazy dog."
    - Window 2: "The quick brown fox jumped over the lazy dog."
    - Window 3: "The quick brown fox jumped over the lazy dog."
    - Window 4: "The quick brown fox jumped over the lazy dog."

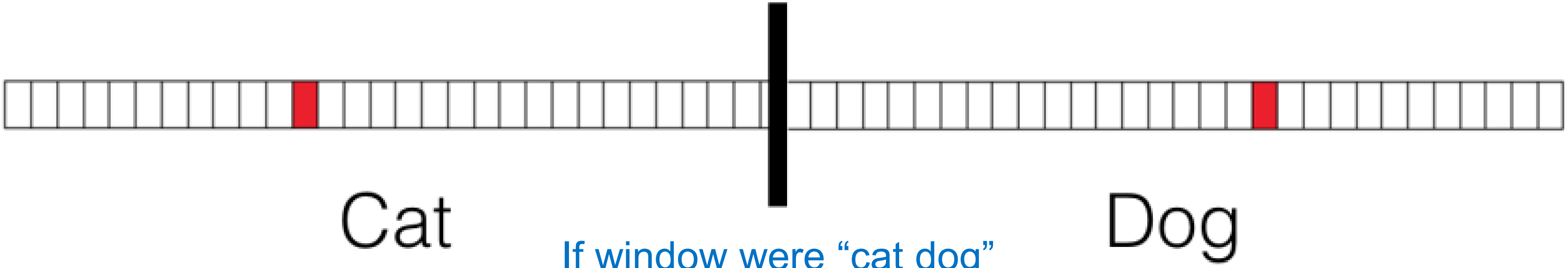
Training Word2Vec: One-Hot Word Vectors

- We need to be able to represent a sequence of words as a vector
- Assign each word an index from 0 to V
  - V is the size of the vocabulary aka # distinct words in the corpus
- A word vector is:
  - 1 for the index of that word
  - 0 for all other entries
- Called One-Hot Encoding



Training Word2Vec: One-Hot Context Windows

- Need vectors for context windows
- A window has vector that's the concatenation of its word vectors
- For window size d, the vector is of length (V x d)
  - Only d entries (one for each word) will be nonzero (1s)



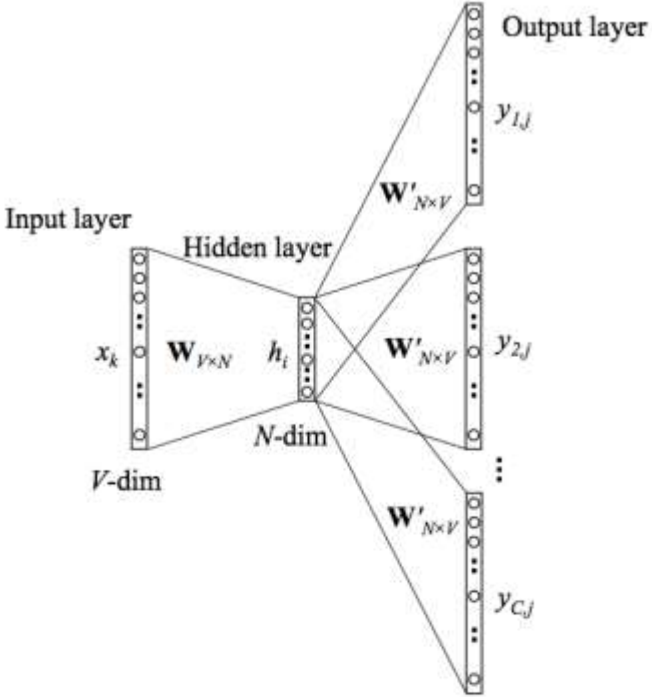
If window were “cat dog”

Training Word2Vec: SkipGrams

- SkipGrams is a neural network architecture that uses a word to predict the words in the surrounding context, defined by the window size.
- Inputs:
  - The middle word of the context window (one-hot encoded)
  - Dimensionality: V
- Outputs:
  - The other words of the context window (one-hot encoded)
  - Dimensionality: (V x (d-1))
  - Turn the crank!

Training Word2Vec: SkipGrams

- SkipGrams architecture:

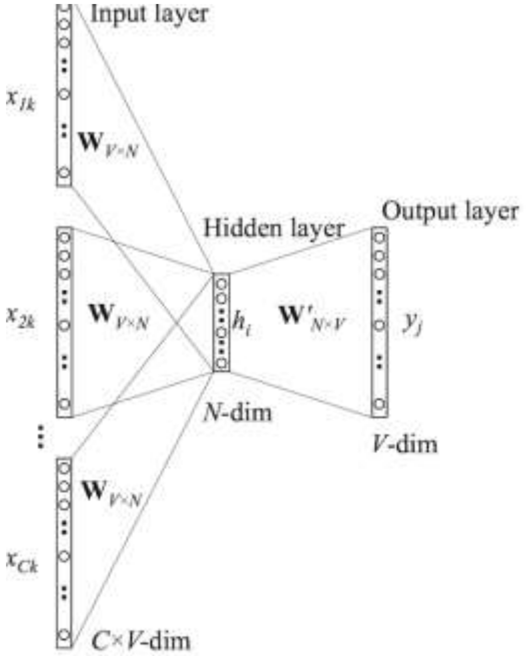


Training Word2Vec: CBOW

- CBOW (continuous bag of words) uses the surrounding context (defined by the window size) to predict the word.
- Inputs:
  - The other words of the context window (one-hot encoded)
  - Dimensionality: (V x (d-1))
- Outputs:
  - The middle word of the context window (one-hot encoded)
  - Dimensionality: V

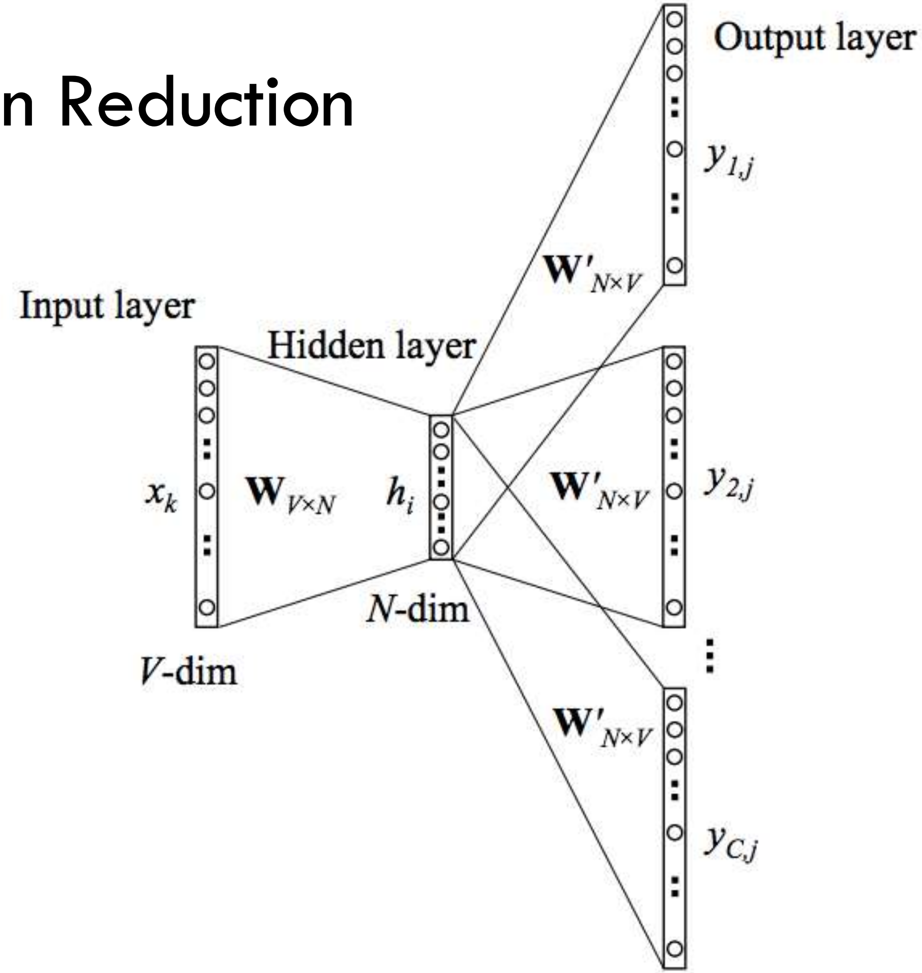
Training Word2Vec: CBOW

- CBOW architecture:



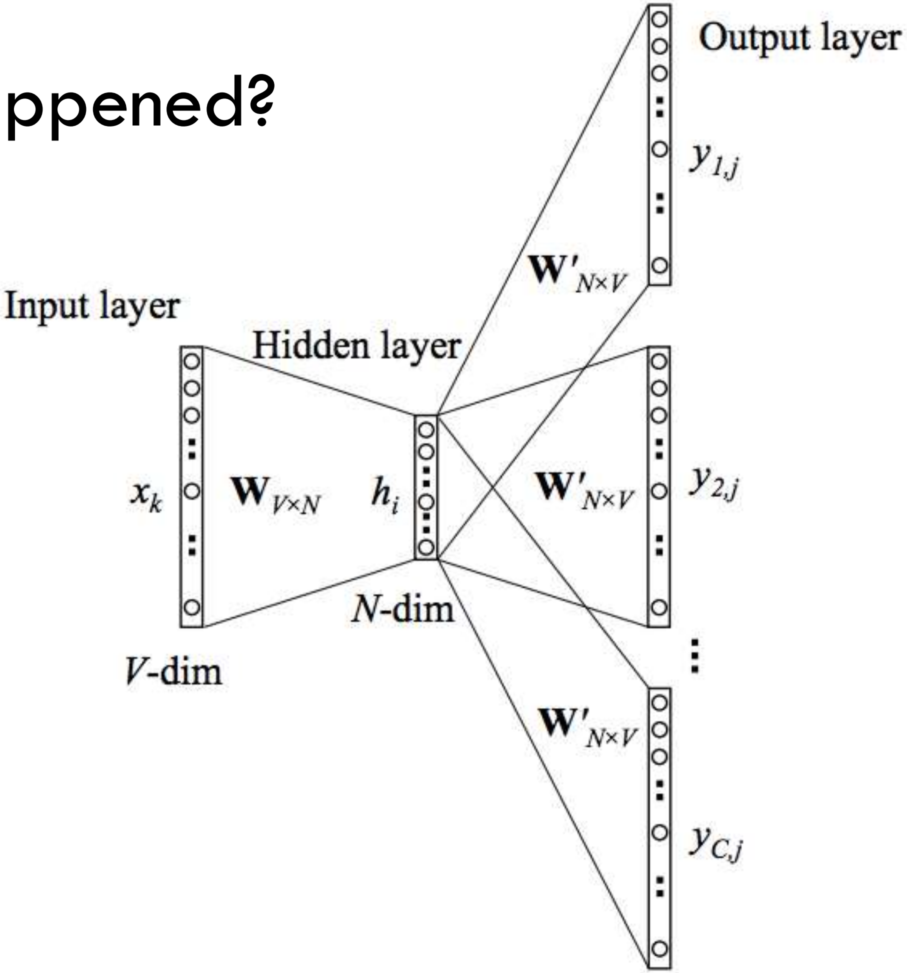
Training Word2Vec: Dimension Reduction

- Number of nodes in hidden layer, N, is a parameter
  - It is the (reduced) dimensionality of our resulting word vector space!
  - Fit neural net → find weights matrix W
  - Word Vectors: $x_N = W^T x$
  - Checking dimensions:
    - x: V x 1
    - W^T: N x V
    - x_N: N x 1



Training Word2Vec: What Happened?

- Learn words likely to appear near each word
- This context information ultimately leads to vectors for related words falling near one another!
- Which gives us really good word vectors! Aka “Word Embeddings”

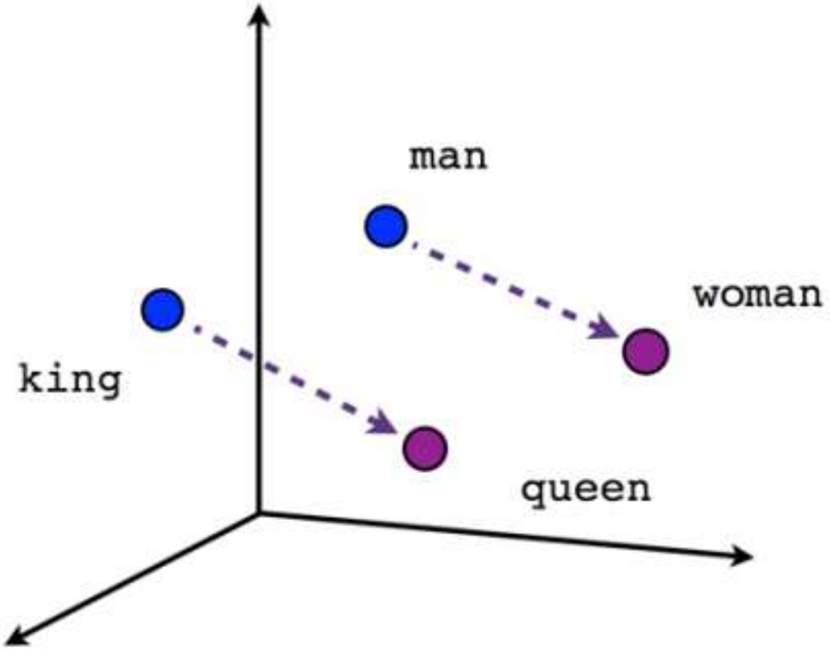


Do I need to Train Word2Vec?

- Answer: NO!
- You can download pre-trained Word2Vec models trained on massive corpora of data.
- Common example: Google News Vectors, 300 dimensional vectors for 3 million words, trained on Google News articles.
- File containing vectors (1.5 GB) can be downloaded for free and easily loaded into gensim.

Nice Properties of Word2Vec Embeddings

- word2vec (somewhat magically!) captures nice geometric relations between words
  - e.g.: Analogies
    - King is to Queen as Man is to Woman
    - The vector between King and Queen is the same as that between man and woman!
    - Works for all sorts of things: capitals, cities, etc



Word2Vec with Gensim

Input:

```
from gensim.models.KeyedVectors import load_word2vec_format

google_model = load_word2vec_format(google_vec_file, binary=True)

# woman - man + king
print(google_model.most_similar(positive=['woman', 'king'], negative=['man'],
topn=3))
```

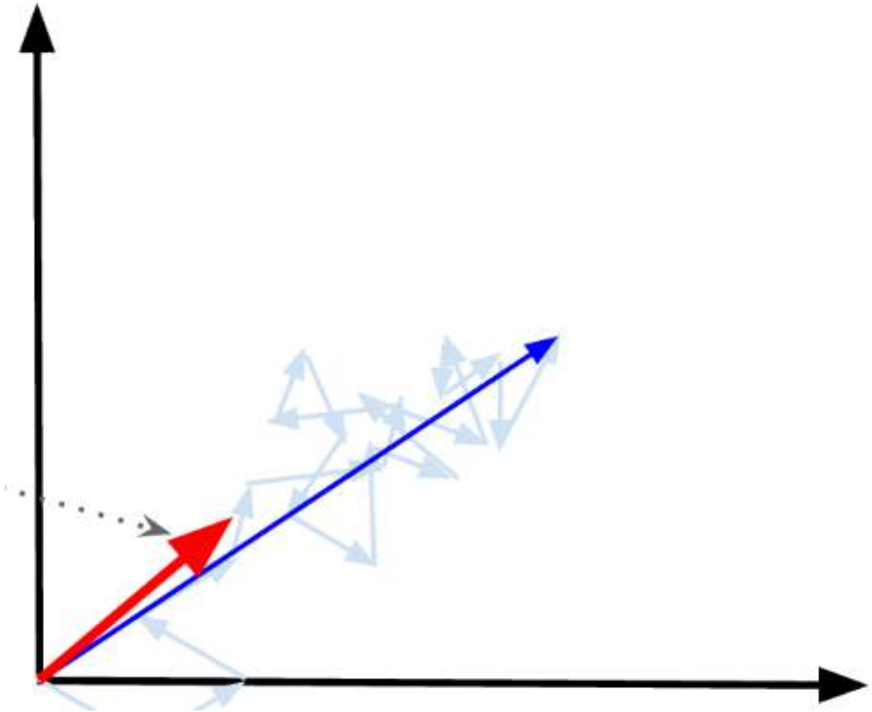
Output:

```
[('queen', 0.7118192911148071),
 ('monarch', 0.6189674139022827),
 ('princess', 0.5902431607246399)]
```

How can we Use Word2Vec?

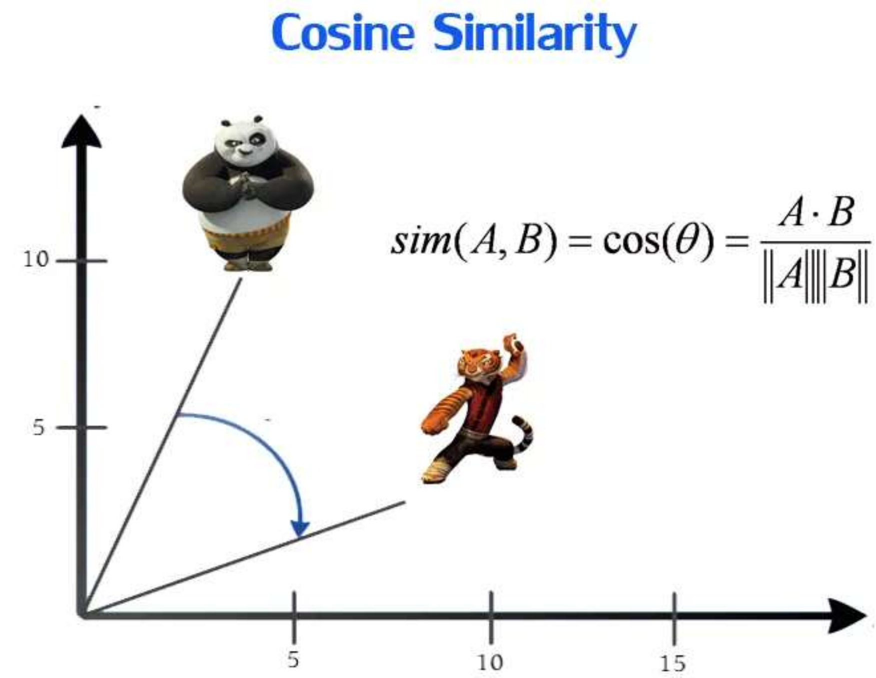
- Vectors can be combined to create features for documents
  - e.g. Document Vector is average (or sum) of its word vectors

- Use Document Vectors for ML on Documents:
  - Classification, Regression
  - Clustering
  - Recommendation



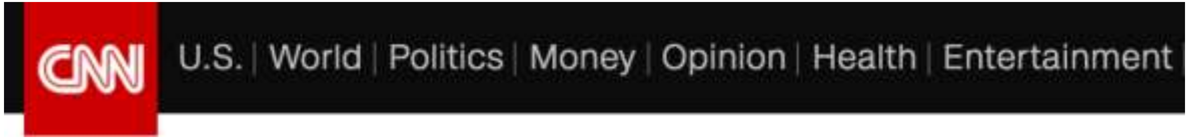
Comparing Word2Vec Embeddings

- How to compare 2 word vectors?
- Cosine Similarity
  - Scaled angle between the vectors
  - Vector length doesn't matter
  - Makes most sense for word vectors
  - Why?
    - e.g. [2, 2, 2] and [4, 4, 4] should be the same vector
    - It's the ratios of frequencies that define meaning



Word Vector Application: Text Classification

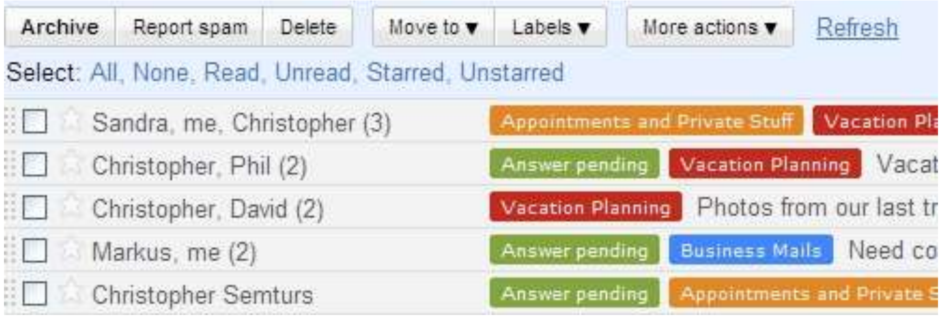
- Problem: Categorizing News Articles
- Is document about Politics? Sports? Science/Tech? etc



- Approach:
  - Word Vectors → Document Vectors
  - Classification on Document Vectors
    - Often KNN with Cosine Similarity

Word Vector Application: Text Clustering

- Problem: Grouping Similar Emails
- Work Emails, Bills, Ads, News, etc

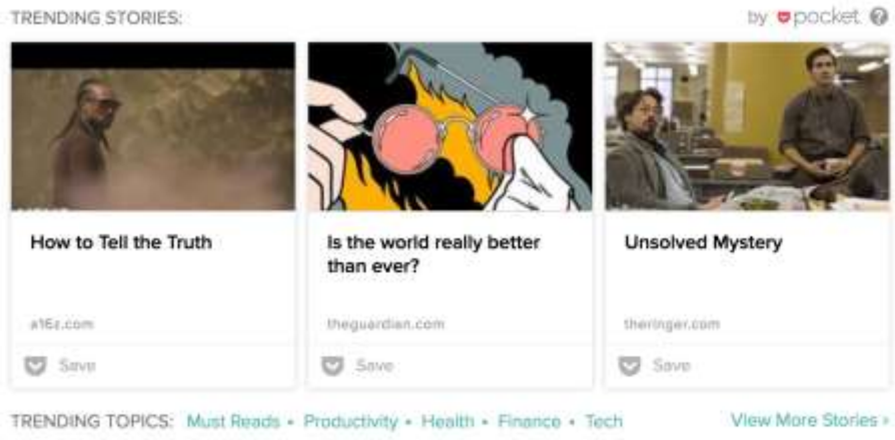


- Approach:
  - Word Vectors → Document Vectors
  - Clustering on Document Vectors
    - Use Cosine Similarity

Word Vector Application: Recommendation

- Problem: Find me news stories I care about!

- Approach:
  - Word Vectors → Document Vectors
  - Suggest documents similar to:
    a) User search query
    b) Example articles that user favorited



Summary

- With word vectors, so many possibilities!
- Can conceptually compare any bunch of words to any other bunch of words.
- Word2Vec finds really good, compact vectors.
  - Trains a Neural Network
  - On Context Windows
  - SkipGram predicts the context words from the middle word in the window.
  - CBOW predicts the middle word from the context words in the window.
- Word Vectors can be used for all sorts of ML

Software

Word2Vec in Python

Word2Vec in Python - Loading Model

Input:

```
from gensim.models.KeyedVectors import load_word2vec_format

google_model = load_word2vec_format(google_vec_file, binary=True)

print(type(google_model.vocab)) # dictionary
print("{:,}".format(len(google_model.vocab.keys()))) # number of words
print(google_model.vector_size) # vector size
```

Output:

```
dict
3,000,000
300
```

Word2Vec in Python - Examining Vectors

Input:

```
bat_vector = google_model.word_vec('bat')

print(type(bat_vector))
print(len(bat_vector))
print(bat_vector.shape)
print(bat_vector[:5])
```

Output:

```
<class 'numpy.ndarray'>
300
(300,)
[-0.34570312  0.32421875  0.15722656 -0.04223633 -0.28710938]
```

Word2Vec in Python - Vector Similarity

Input:

```
print(google_model.similarity(‘Bill_Clinton’, ‘Barack_Obama'))

print(google_model.similarity(‘Bill_Clinton', ‘Taylor_Swift’))
```

Output:

```
0.62116989722645277

0.25381746688228518
```

As expected, Bill Clinton is much more similar to Barack Obama than to Taylor Swift.

Word2Vec in Python - Most Similar Words

Input:

```
print(google_model.similar_by_word('Barack_Obama'))
```

Output:

```
[('Obama', 0.8036513328552246),
 ('Barrack_Obama', 0.7766816020011902),
 ('Illinois_senator', 0.757197916507721),
 ('McCain', 0.7530534863471985),
 ('Barack', 0.7448185086250305),
 ('Barack_Obama_D-Ill.', 0.7196038961410522),
 ('Hillary_Clinton', 0.6864978075027466),
 ('Sen._Hillary_Clinton', 0.6827855110168457),
 ('elect_Barack_Obama', 0.6812860369682312),
 ('Clinton', 0.6713168025016785)]
```

Word2Vec in Python - Analogies

Input:

```
print(google_model.most_similar(positive=[‘Paris', 'Spain'],
                                negative=['France'], topn=2))


print(google_model.most_similar(positive=[‘Yankees', ‘Boston'],
                                negative=['New_York'], topn=2)
```

Output:

```
[('Madrid', 0.7571904063224792), ('Barcelona', 0.6230698823928833)]

[('Red_Sox', 0.8348262906074524), ('Boston_Red_Sox', 0.7118345499038696)]
```

Word2Vec in Python - Odd Word Out

Input:

```
print(google_model.doesnt_match(['breakfast', 'lunch', 'dinner', ‘table']])

print(google_model.doesnt_match(['baseball', 'basketball', 'football',
                                 ‘mattress']))
```

Output:

```
table

mattress
```

As expected, “table” and “mattress” are the odd words out.

Software