Deep Learning Methods for Text Data - Word2Vec, GloVe and FastText

- based on the "A hands-on intuitive approach to Deep Learning Methods for Text Data - Word2Vec, Glove and FastText
- https://towardsdatascience.com/understanding-feature-engineering-part-4-deep-learning-methods-for-text-data-96c44370bbfa (https://towardsdatascience.com/understanding-feature-engineering-part-4-deep-learning-methods-for-text-data-96c44370bbfa)
- added new pytorch version of word2vec instead of Keras

The need for word embeddings

- With regard to speech or image recognition systems, all the information is already present in the form of rich dense feature vectors embedded in high-dimensional datasets like audio spectrograms and image pixel intensities.
- However when it comes to raw text data, especially count based models like Bag of Words, we are dealing with individual words which may have their own identifiers and do not capture the semantic relationship amongst words.
- **This leads to huge sparse word vectors for textual data and thus if we do not have enough data, we may end up getting poor models or even overfitting the data due to the curse of dimensionality**.

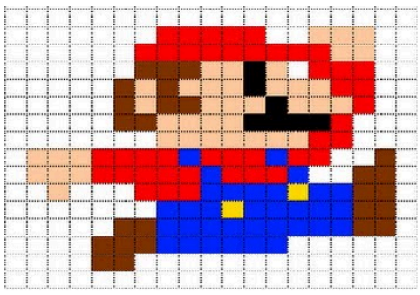
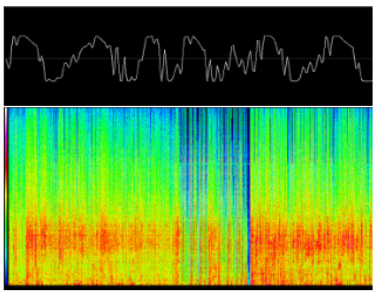


IMAGE PIXELS (DENSE)



AUDIO SPECTROGRAM (DENSE)

WORD VECTORS (SPARSE)

- To overcome the shortcomings of losing out semantics and feature sparsity in bag of words model based features, we need to make use of Vector Space Models (VSMs) in such a way that we can embed word vectors in this continuous vector space based on semantic and contextual similarity.
- In fact the distributional hypothesis in the field of distributional semantics tells us that words which occur and are used in the same context, are semantically similar to one another and have similar meanings.
- In simple terms, *'a word is characterized by the company it keeps'*.

The Word2Vec Model

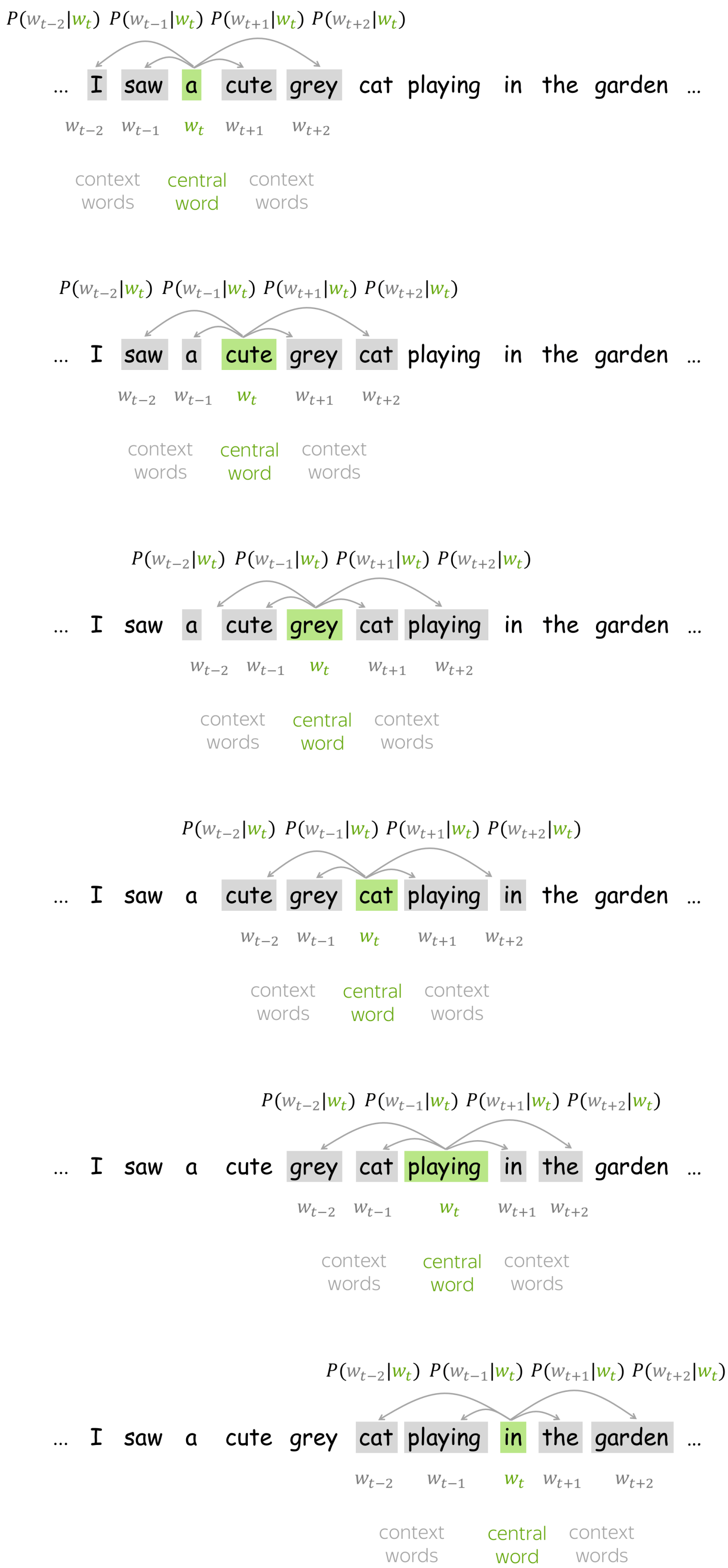
- This model was created by Google in 2013 and is a predictive deep learning based model to compute and generate high quality, distributed and continuous dense vector representations of words, which capture **contextual and semantic similarity**.
- Essentially these are unsupervised models which can take in massive textual corpora, create a vocabulary of possible words and generate **dense word embeddings** for each word in the vector space representing that vocabulary. Usually you can specify the size of the word embedding vectors and the total number of vectors are essentially the size of the vocabulary.
- This makes the dimensionality of this dense vector space much lower than the high-dimensional sparse vector space built using traditional Bag of Words models.
- Two model Architectures

- The Continuous Bag of Words (CBOW) Model
- The Skip-gram Model

- There were originally introduced by Mikolov et al. papers, *'Distributed Representations of Words and Phrases and their Compositionality' 'Efficient Estimation of Word Representations in Vector Space'*

Word2Vec is an iterative method. Its main idea is as follows (according to Skip-gram Model):

- take a huge text corpus;
- go over the text with a sliding window, moving one word at a time. At each step, there is a central word and context - words (other words in this window);
- for the central word, compute probabilities of context words;
- adjust the vectors to increase these probabilities.



Objective Function: Negative Log-Likelihood

For each position $t = 1, \dots, T$ in a text corpus, Word2Vec predicts context words within a m-sized window given the central word w_t:

$$\text{Likelihood} = L(\theta) = \prod_{t=1}^{T} \prod_{-m \le j \le m, j \ne 0} P(w_{t+j} | w_t, \theta),$$

where θ are all variables to be optimized. The objective function (aka loss function or cost function) $J(\theta)$ is the average negative log-likelihood:

$$\text{Loss} = J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m, \\ j \ne 0}} \log P(w_{t+j} | w_t, \theta)$$

agrees with our plan above ⟼ go over text | with a sliding window | compute probability of the context word given the central

Note how well the loss agrees with our plan main above: go over text with a sliding window and compute probabilities. Now let's find out how to compute these probabilities.

Equvalence to minimizing cross-entropy

Note that maximizing data log-likelihood is equivalent to minimizing cross entropy between the target probability distribution $p^* = (0, \dots, 0, 1, 0, \dots)$ (1 for the target label, 0 for the rest) and the predicted by the model distribution $p = (p_1, \dots, p_K), p_i = p(i|x)$:

$$Loss(p^*, p) = -p^* \log(p) = -\sum_{i=1}^{K} p_i^* \log(p_i).$$

Since only one of p_i^* is non-zero (1 for the target label k, 0 for the rest), we will get $Loss(p^*, p) = -\log(p_k) = -\log(p(k|x))$.

Training example: I liked the cat on the mat <eos> Label: k (target)

$\log P(y = k|x) \to max$ — Maximizing log-likelihood of the correct class

$\Updownarrow$

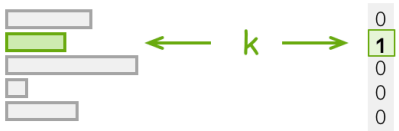
$-\log P(y = k|x) \to min$ — Minimizing negative log-likelihood of the correct class

$\Updownarrow$

$-\sum_{i=1}^{K} p_i^* \cdot \log P(y = i|x) \to min$ — Minimizing cross-entropy loss

$(p_k^* = 1, p_i^* = 0, i \ne k)$

Model prediction: P(class = i | I liked...<eos>) Target: p^*



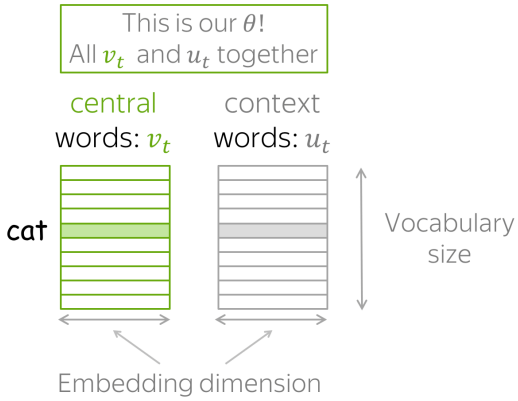
This equivalence is very important for you to understand: when talking about neural approaches, people usually say that they minimize the cross-entropy loss. Do not forget that this is the same as maximizing the data log-likelihood.

How to calculate $P(w_{t+j}|w_t, \theta)$

For each word we will have two vectors:

v_w when it is a central word;

u_w when it is a context word. (Once the vectors are trained, usually we throw away context vectors and use only word vectors.)



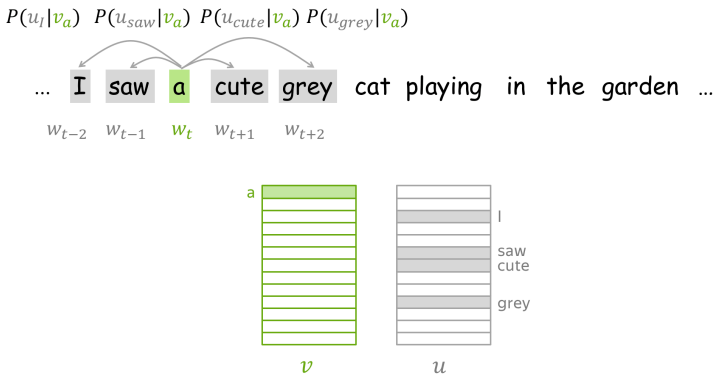
Then for the central word c (c - central) and the context word o (o - outside word) probability of the context word is

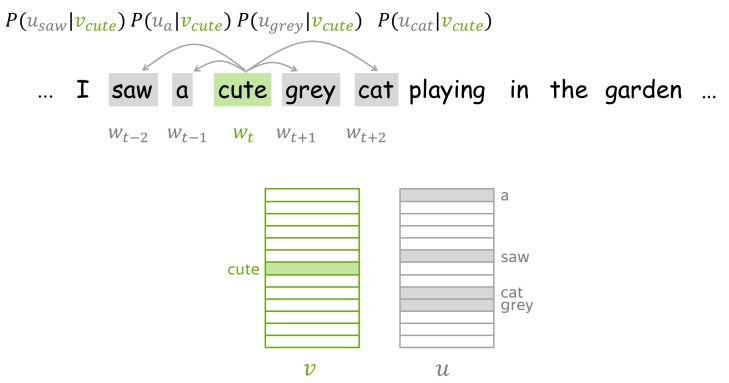
$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

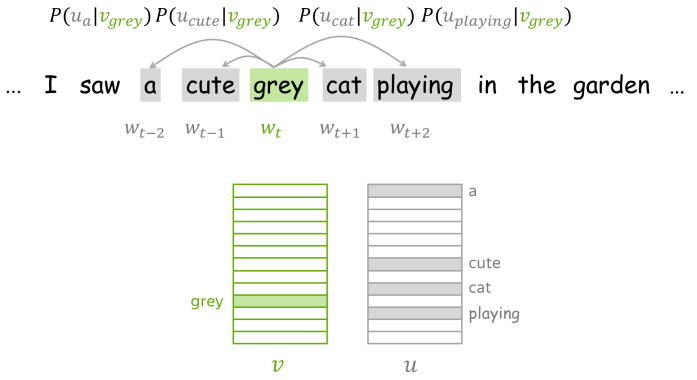
Dot product: measures similarity of o and c
Larger dot product = larger probability

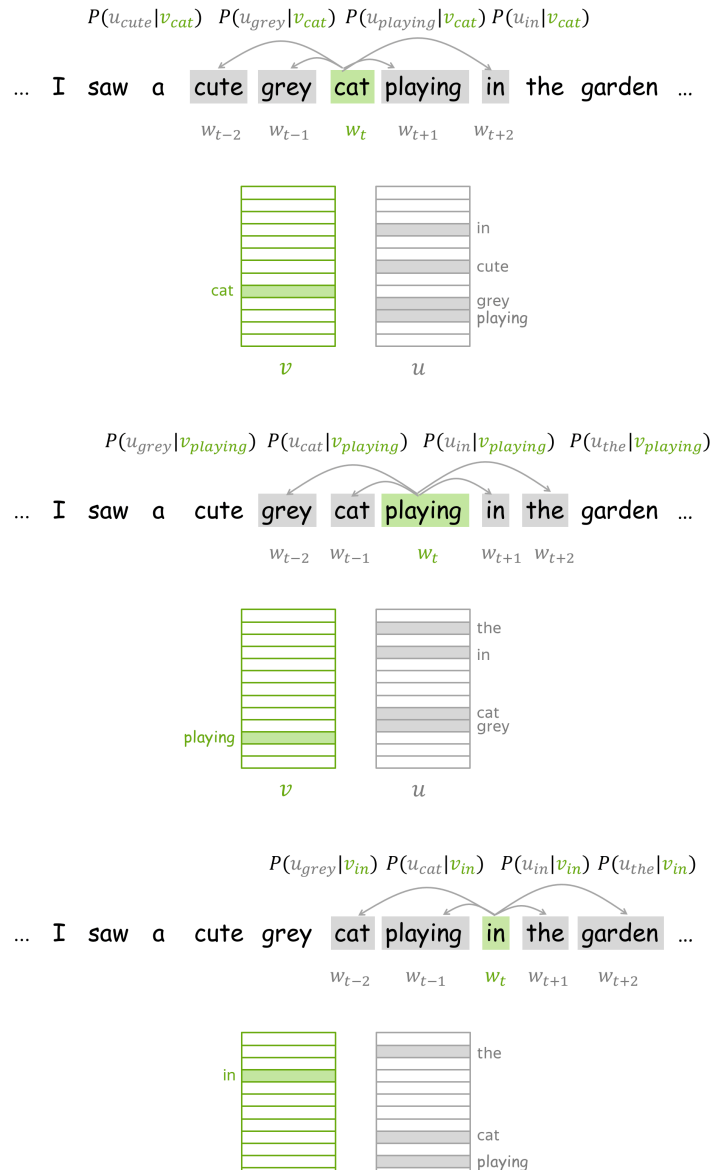
Normalize over entire vocabulary to get probability distribution

How to: go over the illustration. Note that for central words and context words, different vectors are used. For example, first the word a is central and we use v_a, but when it becomes context, we use u_a instead.









How to train: by Gradient Descent, One Word at a Time

Let us recall that our parameters θ are vectors v_w and u_w for all words in the vocabulary. These vectors are learned by optimizing the training objective via gradient descent (with some learning rate α):

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta).$$

One word at a time We make these updates one at a time: each update is for a single pair of a center word and one of its context words. Look again at the loss function:

$$\text{Loss} = J(\theta) = -\frac{1}{T}\log L(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-m\le j\le m, j\neq 0}\log P(w_{t+j}|w_t,\theta) = \frac{1}{T}\sum_{t=1}^{T}\sum_{-m\le j\le m, j\neq 0} J_{t,j}(\theta).$$

For the center word w_t , the loss contains a distinct term $J_{t,j}(\theta) = -\log P(w_{t+j}|w_t,\theta)$ for each of its context words w_{t+j}. Let us look in more detail at just this one term and try to understand how to make an update for this step. For example, let's imagine we have a sentence

... I saw a cute grey cat playing in the garden ...

with the central word cat, and four context words. Since we are going to look at just one step, we will pick only one of the context words; for example, let's take *cute*. Then the loss term for the central word *cat* and the context word *cute* is:

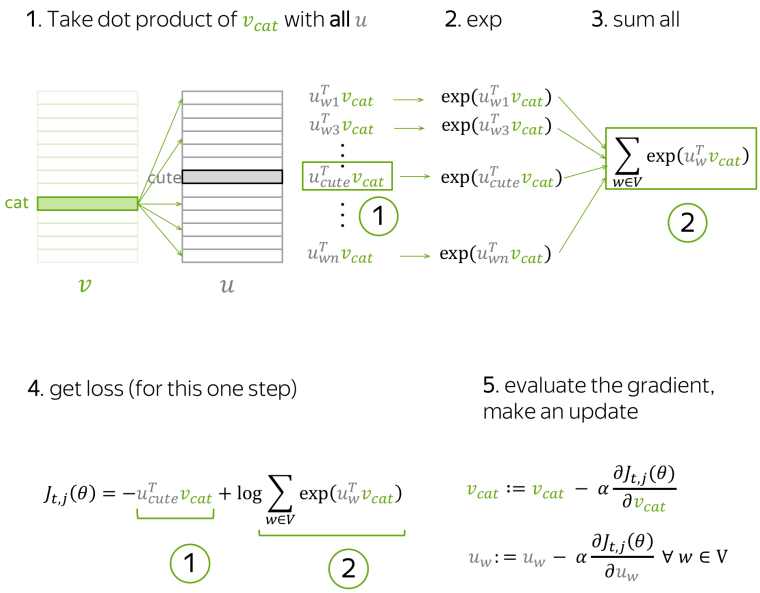
$$J_{t,j}(\theta) = -\log P(cute|cat) = -\log\frac{\exp u_{cute}^T v_{cat}}{\sum_{w\in Voc}\exp u_w^T v_{cat}} = -u_{cute}^T v_{cat} + \log\sum_{w\in Voc}\exp u_w^T v_{cat}.$$

Note which parameters are present at this step:

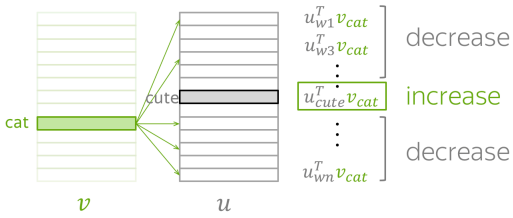
- from vectors for central words, only v_{cat} ;
- from vectors for context words, all u_w (for all words in the vocabulary).

Only these parameters will be updated at the current step.

Below is the schematic illustration of the derivations for this step.



By making an update to minimize $J_{t,j}(\theta)$, we force the parameters to **increase** similarity (dot product) of v_{cat} and u_{cute} and, at the same time, to **decrease** similarity between v_{cat} and u_w for all other words in the vocabulary.

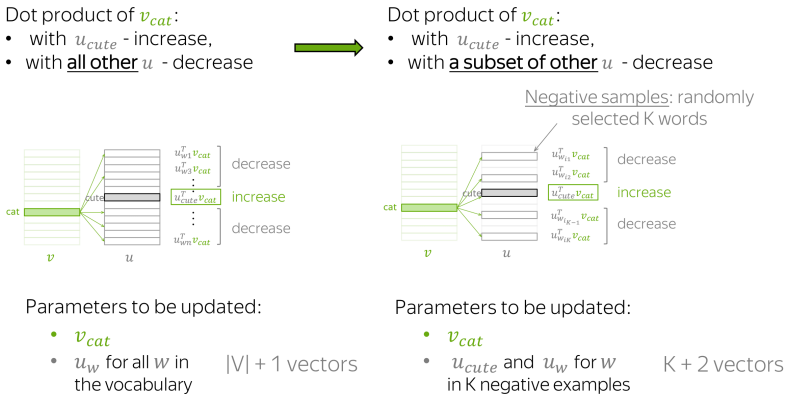


This may sound a bit strange: why do we want to decrease similarity between v_{cat} and all other words, if some of them are also valid context words (e.g., *grey, playing, in* on our example sentence)? But do not worry: since we make updates for each context word (and for all central words in your text), **on average over all updates** our vectors will learn the distribution of the possible contexts.

Faster Training: Negative Sampling

In the example above, for each pair of a central word and its context word, we had to update all vectors for context words. This is highly inefficient: for each step, the time needed to make an update is proportional to the vocabulary size.

But why do we have to consider all context vectors in the vocabulary at each step? For example, imagine that at the current step we consider context vectors not for all words, but only for the current target (cute) and several randomly chosen words. The figure shows the intuition.



As before, we are increasing similarity between v_{cat} and u_{cute}. What is different, is that now we decrease similarity between v_{cat} and context vectors **not for all** words, but only with a **subset of K "negative" examples**.

Since we have a large corpus, on average over all updates we will update each vector sufficient number of times, and the vectors will still be able to learn the relationships between words quite well.

Formally, the new loss function for this step is:

$$J_{t,j}(\theta) = -\log \sigma(u_{cute}^T v_{cat}) - \sum_{w \in \{w_{i_1}, \dots, w_{i_K}\}} \log \sigma(-u_w^T v_{cat}),$$

where $w_{i_1}, \dots, w_{i_K}$ are the K negative examples chosen at this step and $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function.

Note that $\sigma(-x) = \frac{1}{1+e^x} = \frac{1 \cdot e^{-x}}{(1+e^x) \cdot e^{-x}} = \frac{e^{-x}}{1+e^{-x}} = 1 - \frac{1}{1+e^x} = 1 - \sigma(x)$. Then the loss can also be written as:

$$J_{t,j}(\theta) = -\log \sigma(u_{cute}^T v_{cat}) - \sum_{w \in \{w_{i_1}, \dots, w_{i_K}\}} \log(1 - \sigma(u_w^T v_{cat})).$$

The Choice of Negative Examples

Each word has only a few "true" contexts. Therefore, randomly chosen words are very likely to be "negative", i.e. not true contexts. This simple idea is used not only to train Word2Vec efficiently but also in many other applications, some of which we will see later in the course.

Word2Vec randomly samples negative examples based on the empirical distribution of words. Let $U(w)$ be a unigram distribution of words, i.e. $U(w)$ is the frequency of the word w in the text corpus. Word2Vec modifies this distribution to sample less frequent words more often: it samples proportionally to $U^{3/4}(w)$.

Once we have built the data for the positive examples, i.e the words in the neighborhood of the target word, we need to build a data set with negative examples. For each word in the corpus, the probability of sampling a negative context word is defined as follows:

$$P(w_i) = \frac{|w_i|^{\frac{3}{4}}}{\sum_{n} 3}$$

Word2Vec variants: Skip-Gram and CBOW

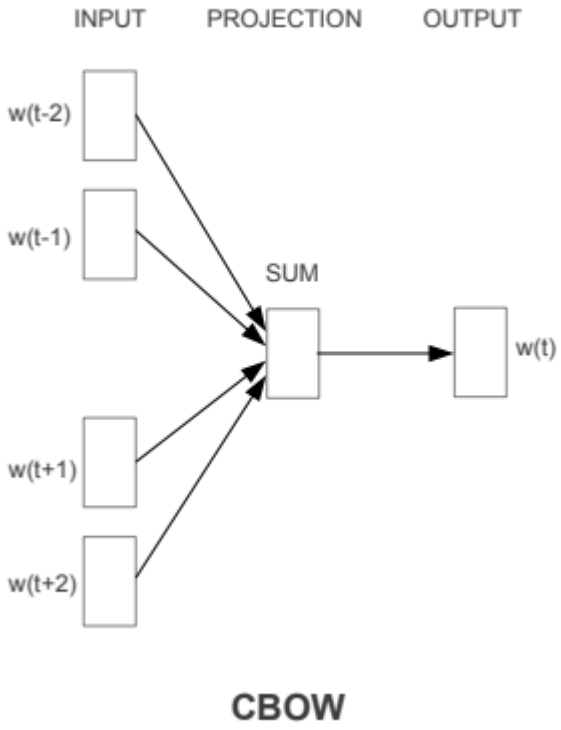
There are two Word2Vec variants: Skip-Gram and CBOW.

Skip-Gram is the model we considered so far: it predicts context words given the central word. Skip-Gram with negative sampling is the most popular approach.

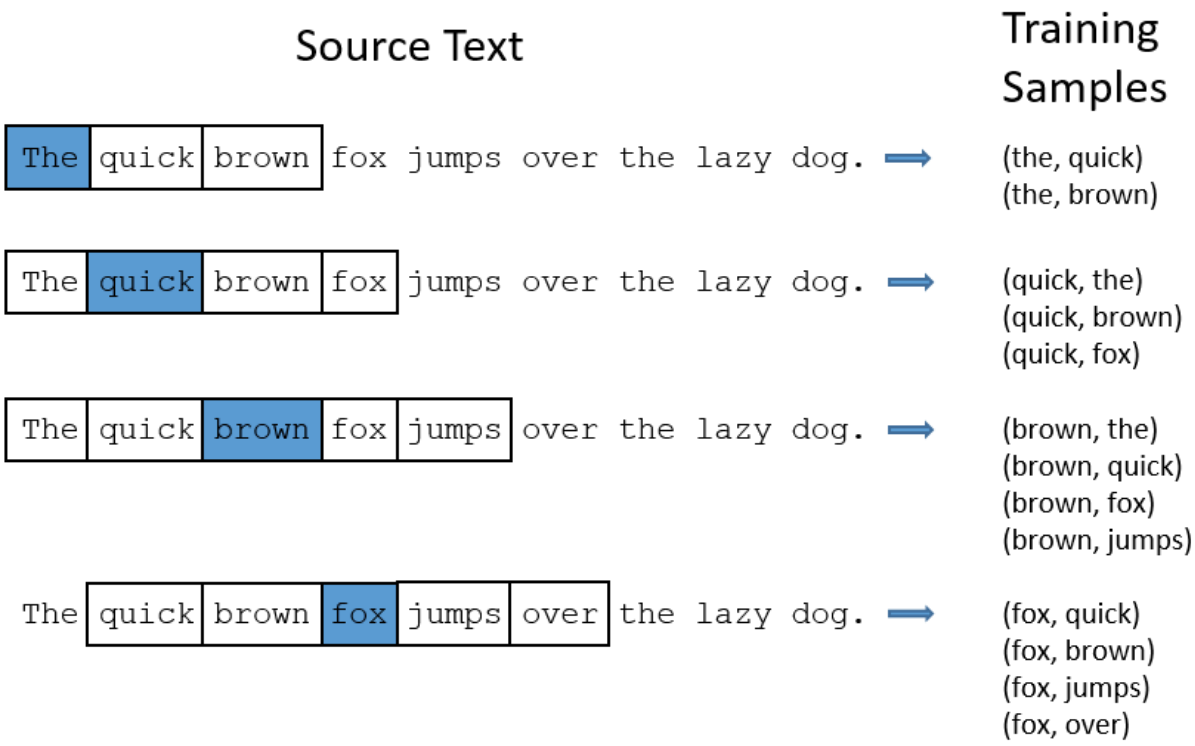
CBOW (Continuous Bag-of-Words) predicts the central word from the sum of context vectors. This simple sum of word vectors is called "bag of words", which gives the name for the model.

The Continuous Bag of Words (CBOW) Model

- The CBOW model architecture tries to predict the current target word (the center word) based on the source context words (surrounding words).
- Considering a simple sentence, *"the quick brown fox jumps over the lazy dog"*, this can be pairs of (context_window, target_word) where if we consider a context window of size 2, we have examples like ([quick, fox], brown), ([the, brown], quick), ([the, dog], lazy) and so on. Thus the model tries to predict the target_word based on the context_window words.



Skip-gram, Why More Efficient than CBOW?



Robust Word2Vec Models with Gensim

- While our implementations are decent enough, they are not optimized enough to work well on large corpora.
- The **gensim** framework, created by Radim Řehůřek consists of a robust, efficient and scalable implementation of the Word2Vec model.
- We will leverage the same on Bible corpus.
- In our workflow, we will tokenize our normalized corpus and then focus on the following four parameters in the Word2Vec model to build it.
  - vector_size: The word embedding dimensionality
  - window: The context window size
  - min_count: The minimum word count
  - sample: The downsample setting for frequent words
- After building our model, we will use our words of interest to see the top similar words for each of them.

In [ ]:
```
!pip install gensim
```

- Let's now load up our other corpus based on The King James Version of the Bible using nltk and pre-process the text.

In [37]:
```
import pandas as pd
import numpy as np
import re
import nltk
import matplotlib.pyplot as plt
pd.options.display.max_colwidth = 200
%matplotlib inline
```

In [38]:
```
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /home/hpshin/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[38]:
```
True
```

In [39]:
```
from nltk.corpus import gutenberg
from string import punctuation

wpt = nltk.WordPunctTokenizer()
stop_words = nltk.corpus.stopwords.words('english')

def normalize_document(doc):
    # lower case and remove special characters\whitespaces
    doc = re.sub(r'[^a-zA-Z\s]', '', doc, re.I|re.A)
    doc = doc.lower()
    doc = doc.strip()
    # tokenize document
    tokens = wpt.tokenize(doc)
    # filter stopwords out of document
    filtered_tokens = [token for token in tokens if token not in stop_words]
    # re-create document from filtered tokens
    doc = ' '.join(filtered_tokens)
    return doc

normalize_corpus = np.vectorize(normalize_document)
```

In [40]:
```
bible = gutenberg.sents('bible-kjv.txt')
#print(bible)
remove_terms = punctuation + '0123456789'

norm_bible = [[word.lower() for word in sent if word not in remove_terms] for
norm_bible = [' '.join(tok_sent) for tok_sent in norm_bible]
norm_bible = filter(None, normalize_corpus(norm_bible))
norm_bible = [tok_sent for tok_sent in norm_bible if len(tok_sent.split()) > 2

print('Total lines:', len(bible))
print('\nSample line:', bible[10])
print('\nProcessed line:', norm_bible[10])
```

```
Total lines: 30103

Sample line: ['1', ':', '6', 'And', 'God', 'said', ',', 'Let', 'there', 'be',
'a', 'firmament', 'in', 'the', 'midst', 'of', 'the', 'waters', ',', 'and', 'l
et', 'it', 'divide', 'the', 'waters', 'from', 'the', 'waters', '.']

Processed line: god said let firmament midst waters let divide waters waters
```

In [12]:
```
# MODULE SLIGHTLY CHANGED
# https://radimrehurek.com/gensim/models/word2vec.html
# https://github.com/piskvorky/gensim/wiki/Migrating-from-Gensim-3.x-to-4


from gensim.models import word2vec

# tokenize sentences in corpus
wpt = nltk.WordPunctTokenizer()
tokenized_corpus = [wpt.tokenize(document) for document in norm_bible]

# Set values for various parameters
feature_size = 100    # Word vector dimensionality
window_context = 30          # Context window size
min_word_count = 1   # Minimum word count
sample = 1e-3   # Downsample setting for frequent words

w2v_model = word2vec.Word2Vec(tokenized_corpus, vector_size=feature_size,
                              window=window_context, min_count=min_word_count,
                              sample=sample, epochs=50)

# view similar words based on gensim's model
similar_words = {search_term: [item[0] for item in w2v_model.wv.most_similar([s
                  for search_term in ['god', 'jesus', 'noah', 'egypt', 'john',
similar_words
```

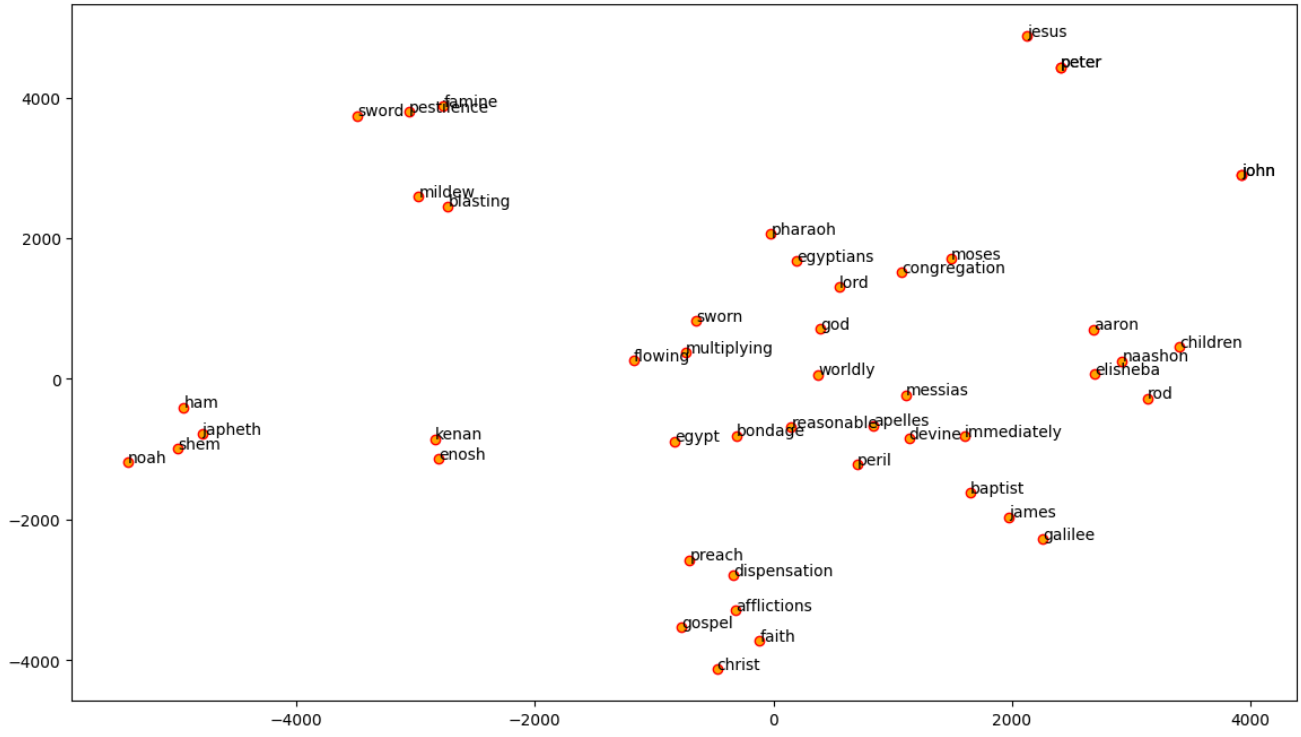
Out[12]:
```
{'god': ['lord', 'worldly', 'sworn', 'multiplying', 'reasonable'],
 'jesus': ['messias', 'peter', 'apelles', 'immediately', 'john'],
 'noah': ['shem', 'ham', 'japheth', 'kenan', 'enosh'],
 'egypt': ['pharaoh', 'egyptians', 'bondage', 'rod', 'flowing'],
 'john': ['baptist', 'james', 'devine', 'peter', 'galilee'],
 'gospel': ['christ', 'faith', 'afflictions', 'preach', 'dispensation'],
 'moses': ['congregation', 'naashon', 'children', 'elisheba', 'aaron'],
 'famine': ['pestilence', 'peril', 'sword', 'mildew', 'blasting']}
```

- The similar words here definitely are more related to our words of interest and this is expected given that we ran this model for more number of iterations which must have yield better and more contextual embeddings.
- Do you notice any interesting associations?

NOAH AND HIS SONS

- Let's also visualize the words of interest and their similar words using their embedding vectors after reducing their dimensions to a 2-D space with t-SNE.

In [13]:
```
from sklearn.manifold import TSNE

words = sum([[k] + v for k, v in similar_words.items()], [])
wvs = w2v_model.wv[words]

tsne = TSNE(n_components=2, random_state=0, n_iter=10000, perplexity=2)
np.set_printoptions(suppress=True)
T = tsne.fit_transform(wvs)
labels = words

plt.figure(figsize=(14, 8))
plt.scatter(T[:, 0], T[:, 1], c='orange', edgecolors='r')
for label, x, y in zip(labels, T[:, 0], T[:, 1]):
    plt.annotate(label, xy=(x+1, y+1), xytext=(0, 0), textcoords='offset points
```



- We can clearly see based on what I depicted earlier that noah and his sons are quite close to each other based on the word embeddings from our model!

Applying Word2Vec features for Machine Learning Tasks

- Let's try to apply word2vec features to actual machine learning task like clustering.
- To start with, we will build a simple Word2Vec model on the corpus and visualize the embeddings.

In [14]:
```
corpus = ['The sky is blue and beautiful.',
          'Love this blue and beautiful sky!',
          'The quick brown fox jumps over the lazy dog.',
          "A king's breakfast has sausages, ham, bacon, eggs, toast and beans",
          'I love green eggs, ham, sausages and bacon!',
          'The brown fox is quick and the blue dog is lazy!',
          'The sky is very blue and the sky is very beautiful today',
          'The dog is lazy but the brown fox is quick!'
]
labels = ['weather', 'weather', 'animals', 'food', 'food', 'animals', 'weather

corpus = np.array(corpus)
corpus_df = pd.DataFrame({'Document': corpus,
                          'Category': labels})
corpus_df = corpus_df[['Document', 'Category']]
corpus_df
```

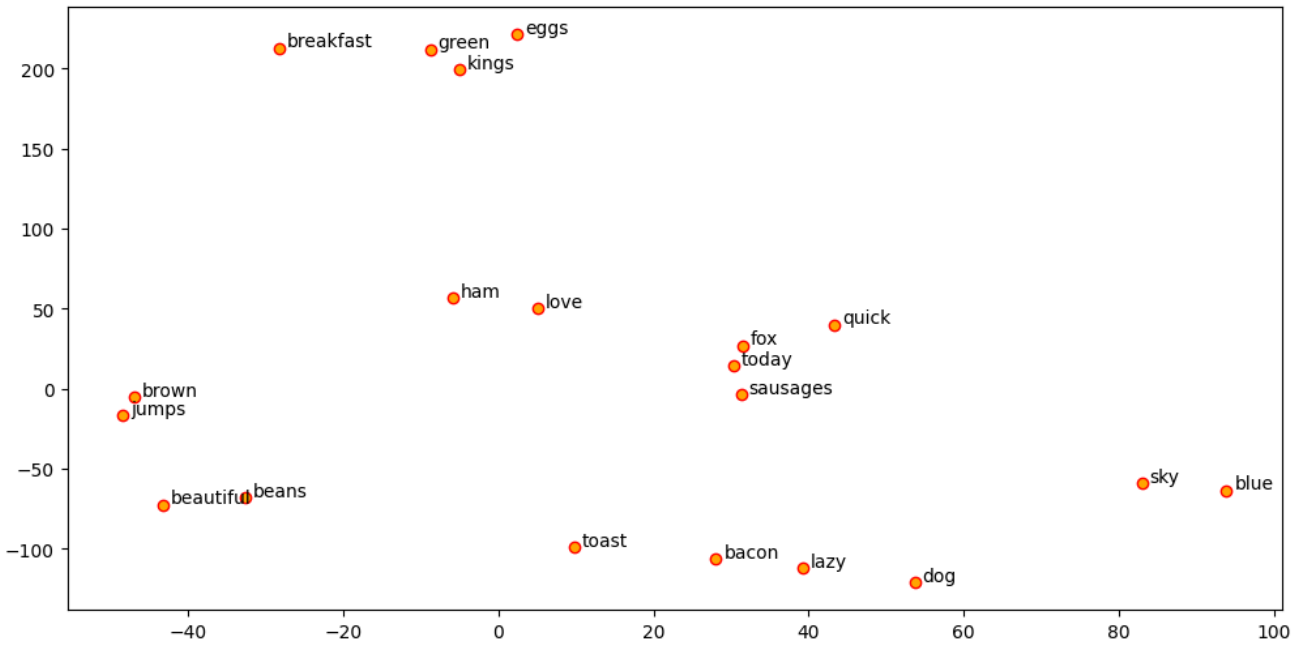
Out[14]:

	Document	Category
0	The sky is blue and beautiful.	weather
1	Love this blue and beautiful sky!	weather
2	The quick brown fox jumps over the lazy dog.	animals
3	A king's breakfast has sausages, ham, bacon, eggs, toast and beans	food
4	I love green eggs, ham, sausages and bacon!	food
5	The brown fox is quick and the blue dog is lazy!	animals
6	The sky is very blue and the sky is very beautiful today	weather
7	The dog is lazy but the brown fox is quick!	animals

In [15]:
```
norm_corpus = normalize_corpus(corpus)
norm_corpus
```

Out[15]:
```
array(['sky blue beautiful', 'love blue beautiful sky',
       'quick brown fox jumps lazy dog',
       'kings breakfast sausages ham bacon eggs toast beans',
       'love green eggs ham sausages bacon',
       'brown fox quick blue dog lazy', 'sky blue sky beautiful today',
       'dog lazy brown fox quick'], dtype='<U51')
```

In [19]:
```
# build word2vec model
wpt = nltk.WordPunctTokenizer()
tokenized_corpus = [wpt.tokenize(document) for document in norm_corpus]

# Set values for various parameters
feature_size = 10    # Word vector dimensionality
window_context = 10          # Context window size
min_word_count = 1   # Minimum word count
sample = 1e-3   # Downsample setting for frequent words

w2v_model = word2vec.Word2Vec(tokenized_corpus, vector_size=feature_size,
                              window=window_context, min_count = min_word_count
                              sample=sample, epochs=100)


# visualize embeddings
from sklearn.manifold import TSNE

words = w2v_model.wv.index_to_key
wvs = w2v_model.wv[words]

tsne = TSNE(n_components=2, random_state=0, n_iter=5000, perplexity=2)
np.set_printoptions(suppress=True)
T = tsne.fit_transform(wvs)
labels = words

plt.figure(figsize=(12, 6))
plt.scatter(T[:, 0], T[:, 1], c='orange', edgecolors='r')
for label, x, y in zip(labels, T[:, 0], T[:, 1]):
    plt.annotate(label, xy=(x+1, y+1), xytext=(0, 0), textcoords='offset points
```



- Remember that our corpus is extremely small so to get meaninful word embeddings and for the model to get more context and semantics, more data helps.
- Now what is a word embedding in this scenario? It's typically a dense vector for each word as depicted in the following example for the word **sky**.

In [20]:
```
w2v_model.wv['sky']
```

Out[20]:
```
array([-0.00550217,  0.00225759,  0.05222183,  0.08953774, -0.09186712,
       -0.07102545,  0.06703831,  0.09108812, -0.05248048, -0.03727873],
      dtype=float32)
```

- Now suppose we wanted to cluster the eight documents from our toy corpus, we would need to get the document level embeddings from each of the words present in each document.
- One strategy would be to average out the word embeddings for each word in a document.
- This is an extremely useful strategy and you can adopt the same for your own problems.
- Let's apply this now on our corpus to get features for each document.

In [24]:
```
def average_word_vectors(words, model, vocabulary, num_features):

    feature_vector = np.zeros((num_features,),dtype="float64")
    nwords = 0.

    for word in words:
        if word in vocabulary:
            nwords = nwords + 1.
            feature_vector = np.add(feature_vector, model.wv[word])

    if nwords:
        feature_vector = np.divide(feature_vector, nwords)

    return feature_vector


def averaged_word_vectorizer(corpus, model, num_features):
    vocabulary = set(model.wv.index_to_key)
    features = [average_word_vectors(tokenized_sentence, model, vocabulary, nu
                    for tokenized_sentence in corpus]
    return np.array(features)


# get document level embeddings
w2v_feature_array = averaged_word_vectorizer(corpus=tokenized_corpus, model=w2
                                             num_features=feature_size)
pd.DataFrame(w2v_feature_array)
```

Out[24]:

	0	1	2	3	4	5	6	7	8	
0	-0.002563	-0.007418	0.034811	0.026904	-0.053587	-0.035993	0.061499	0.014958	-0.047006	-0.0592
1	0.003990	-0.017028	0.047207	-0.004556	-0.022574	-0.019438	0.034795	0.022913	-0.040348	-0.0274
2	-0.025970	0.044765	-0.027590	-0.042492	0.038190	0.001035	0.020798	0.014793	-0.009800	0.0260
3	-0.004875	-0.023161	0.012324	0.017017	0.023751	0.013767	0.019178	-0.004341	0.008625	0.0135
4	0.002887	-0.026936	0.028622	0.022390	0.038068	0.033622	0.020794	0.024426	-0.022648	0.0369
5	0.000605	0.036747	-0.027230	-0.022999	0.024138	-0.010912	0.013114	0.026146	-0.036314	-0.0007
6	-0.000502	-0.007114	0.035939	0.018142	-0.055835	-0.030537	0.061412	0.022556	-0.058082	-0.0340
7	-0.013937	0.047206	-0.023748	-0.040617	0.038326	-0.009574	0.009073	0.029035	-0.026323	0.0177

- Now that we have our features for each document, let's cluster these documents using the **Affinity Propagation**(https://towardsdatascience.com/unsupervised-machine-learning-affinity-propagation-algorithm-explained-d1fef85f22c8 (https://towardsdatascience.com/unsupervised-machine-learning-affinity-propagation-algorithm-explained-d1fef85f22c8)) algorithm, which is a clustering algorithm based on the concept of "message passing" between data points and does not need the number of clusters as an explicit input which is often required by partition-based clustering algorithms.

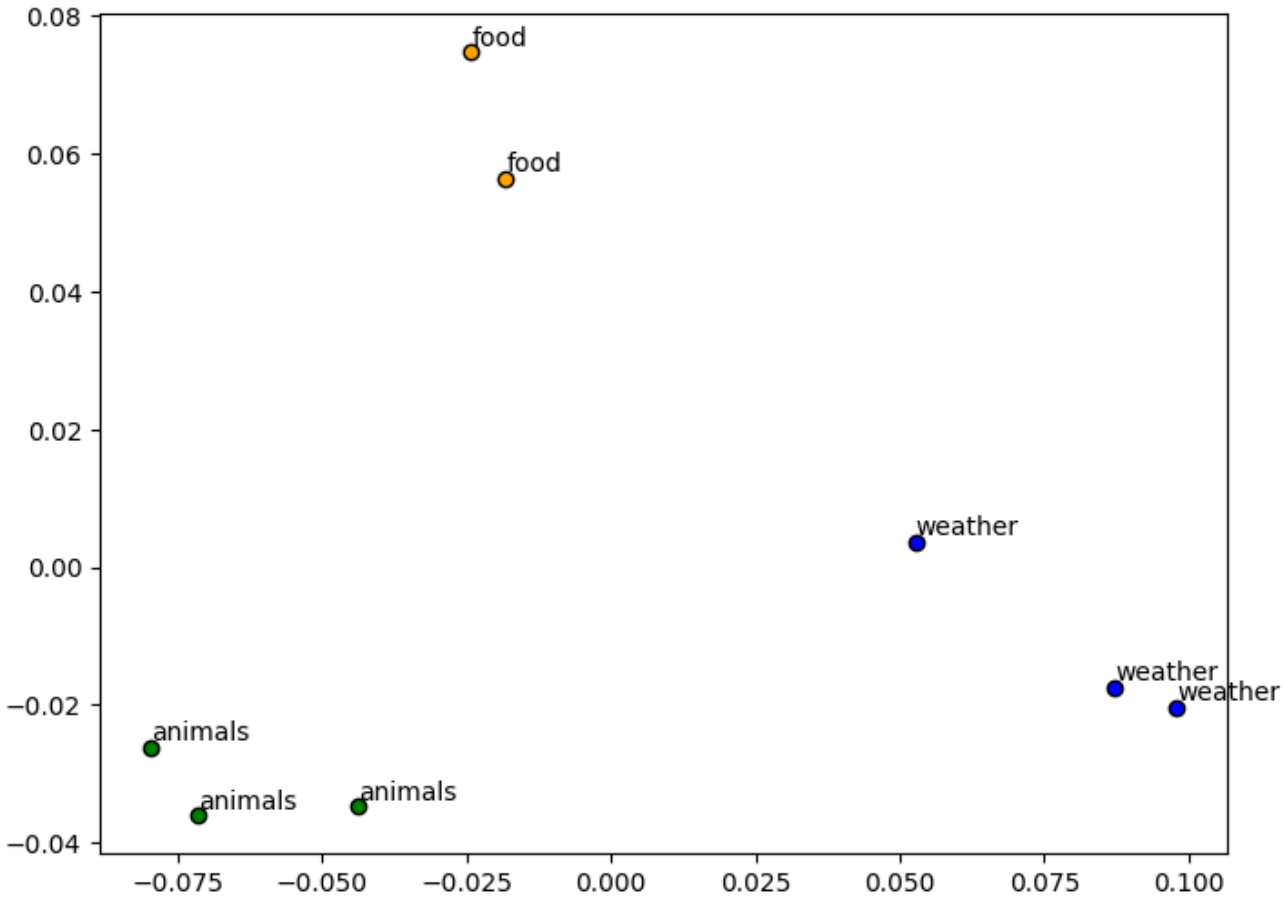
In [25]:
```
from sklearn.cluster import AffinityPropagation

ap = AffinityPropagation()
ap.fit(w2v_feature_array)
cluster_labels = ap.labels_
cluster_labels = pd.DataFrame(cluster_labels, columns=['ClusterLabel'])
pd.concat([corpus_df, cluster_labels], axis=1)
```

Out[25]:

	Document	Category	ClusterLabel
0	The sky is blue and beautiful.	weather	1
1	Love this blue and beautiful sky!	weather	1
2	The quick brown fox jumps over the lazy dog.	animals	2
3	A king's breakfast has sausages, ham, bacon, eggs, toast and beans	food	0
4	I love green eggs, ham, sausages and bacon!	food	0
5	The brown fox is quick and the blue dog is lazy!	animals	2
6	The sky is very blue and the sky is very beautiful today	weather	1
7	The dog is lazy but the brown fox is quick!	animals	2

- We can see that our algorithm has clustered each document into the right group based on our Word2Vec features.
- Pretty neat! We can also visualize how each document in positioned in each cluster by using Principal Component Analysis (PCA) to reduce the feature dimensions to 2-D and then visualizing the same (by color coding each cluster).

In [26]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=2, random_state=0)
pcs = pca.fit_transform(w2v_feature_array)
labels = ap.labels_
categories = list(corpus_df['Category'])
plt.figure(figsize=(8, 6))

for i in range(len(labels)):
    label = labels[i]
    color = 'orange' if label == 0 else 'blue' if label == 1 else 'green'
    annotation_label = categories[i]
    x, y = pcs[i]
    plt.scatter(x, y, c=color, edgecolors='k')
    plt.annotate(annotation_label, xy=(x+1e-4, y+1e-3), xytext=(0, 0), textcoo
```

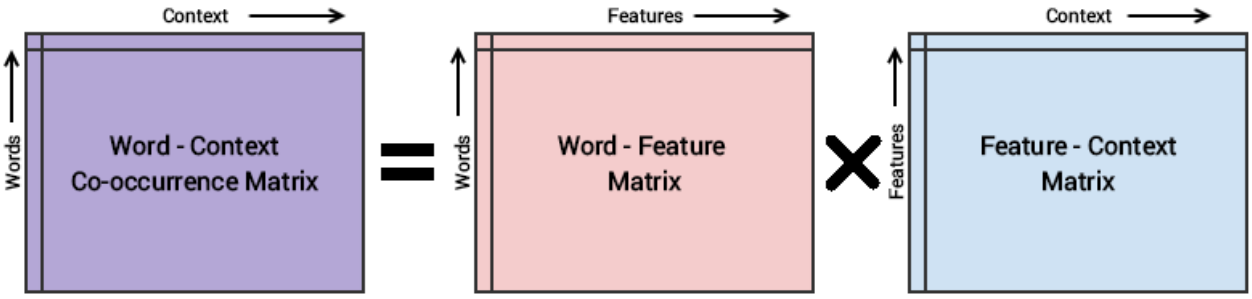


- Everything looks to be in order as documents in each cluster are closer to each other and far apart from other clusters.

The GloVe Model

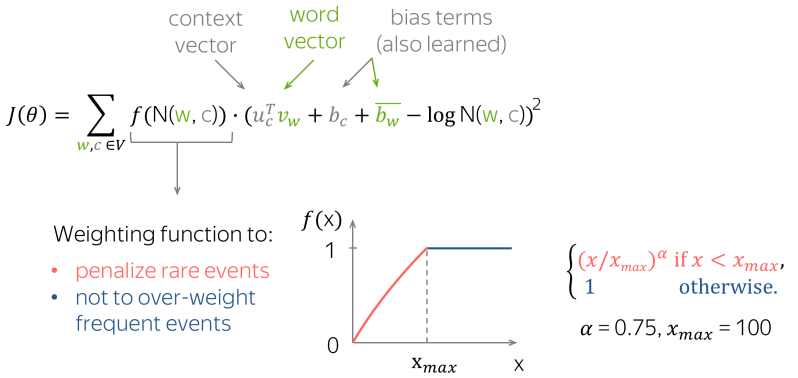
- The GloVe model stands for Global Vectors which is an unsupervised learning model which can be used to obtain dense word vectors similar to Word2Vec.
- However the technique is different and training is performed on an aggregated **global word-word co-occurrence matrix**, giving us a vector space with meaningful sub-structures.
- This method was invented in Stanford by Pennington et al. and I recommend you to read the original paper on GloVe, 'GloVe: Global Vectors for Word Representation' by Pennington et al. which is an excellent read to get some perspective on how this model works.

- The basic methodology of the GloVe model is to first create a huge word-context co-occurence matrix consisting of (word, context) pairs such that each element in this matrix represents how often a word occurs with the context (which can be a sequence of words). The idea then is to apply matrix factorization to approximate this matrix as depicted in the following figure.



The GloVe model is a combination of count-based methods and prediction methods (e.g., Word2Vec). Model name, GloVe, stands for "Global Vectors", which reflects its idea: the method uses global information from corpus to learn vectors.

As we saw earlier, the simplest count-based method uses co-occurrence counts to measure the association between word w and context c: N(w, c). GloVe also uses these counts to construct the loss function:



Similar to Word2Vec, we also have different vectors for central and context words - these are our parameters. Additionally, the method has a scalar bias term for each word vector.

What is especially interesting, is the way GloVe controls the influence of rare and frequent words: loss for each pair (w, c) is weighted in a way that

- rare events are penalized,
- very frequent events are not over-weighted.

- Given the words i = ice and j = steam, we want to study a ratio of co-occurrence probabilities with some probe word k = solid. We can expect the co-occurrence between the word i and the word k (Pik) being great over Pjk. There, the ratio Pik Pjk should be great.
- We describe this ratio by the following formula :

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

Which, as shown by the paper, can be simplified as :

$$w_i^T \tilde{w}_k + b_i + \tilde{b}_k = log(X_{ik})$$

Which resolution is closely related to LSA, and older method.

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Co-occurrence probabilities and ratio with the words ice and steam in function of solid, gas, water and fashion. Since ice is more related to solid than steam, the co-occurrence ratio is high. This is the contrary considering the probe word steam. They are both equally related to water (highly) and to fashion (vaguely) so the co-occurrence ratio is approximately equals to 1. (https://www.aclweb.org/anthology/D14-1162/ (https://www.aclweb.org/anthology/D14-1162/))

Applying GloVe features for Machine Learning Tasks

- Let's try and leverage GloVe based embeddings for our document clustering task.
- The very popular spacy framework comes with capabilities to leverage GloVe embeddings based on different language models.
- You can also get pre-trained word vectors and load them up as needed using gensim or spacy.
- We will first install spacy and use the en_vectors_web_lg model which consists of 300-dimensional word vectors trained on Common Crawl with GloVe.

In [28]:
```
#  python -m spacy download en_core_web_lg
import spacy

nlp = spacy.load('en_core_web_sm')

total_vectors = len(nlp.vocab.vectors)
print('Total word vectors:', total_vectors)
```

```
---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
Cell In[28], line 4
      1 #  python -m spacy download en_core_web_lg
      2 import spacy
----> 4 nlp = spacy.load('en_core_web_sm')
      6 total_vectors = len(nlp.vocab.vectors)
      7 print('Total word vectors:', total_vectors)

File ~/anaconda3/envs/book/lib/python3.9/site-packages/spacy/__init__.py:51,
in load(name, vocab, disable, enable, exclude, config)
     27 def load(
     28     name: Union[str, Path],
     29     *,
   (...)
     34     config: Union[Dict[str, Any], Config] = util.SimpleFrozenDict(),
     35 ) -> Language:
     36     """Load a spaCy model from an installed package or a local path.
     37
     38     name (str): Package name or model path.
   (...)
     49     RETURNS (Language): The loaded nlp object.
     50     """
---> 51     return util.load_model(
     52         name,
     53         vocab=vocab,
     54         disable=disable,
     55         enable=enable,
     56         exclude=exclude,
     57         config=config,
     58     )

File ~/anaconda3/envs/book/lib/python3.9/site-packages/spacy/util.py:472, in
load_model(name, vocab, disable, enable, exclude, config)
    470 if name in OLD_MODEL_SHORTCUTS:
    471     raise IOError(Errors.E941.format(name=name, full=OLD_MODEL_SHORTC
UTS[name]))  # type: ignore[index]
--> 472 raise IOError(Errors.E050.format(name=name))

OSError: [E050] Can't find model 'en_core_web_sm'. It doesn't seem to be a Py
thon package or a valid path to a data directory.
```

In [ ]:
```
unique_words = list(set([word for sublist in [doc.split() for doc in norm_corpu

word_glove_vectors = np.array([nlp(word).vector for word in unique_words])
pd.DataFrame(word_glove_vectors, index=unique_words)
```

We can now use t-SNE to visualize these embeddings similar to what we did using our Word2Vec embeddings.

In [ ]:
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, random_state=0, n_iter=5000, perplexity=3)
np.set_printoptions(suppress=True)
T = tsne.fit_transform(word_glove_vectors)
labels = unique_words

plt.figure(figsize=(12, 6))
plt.scatter(T[:, 0], T[:, 1], c='orange', edgecolors='r')
for label, x, y in zip(labels, T[:, 0], T[:, 1]):
    plt.annotate(label, xy=(x+1, y+1), xytext=(0, 0), textcoords='offset points
```

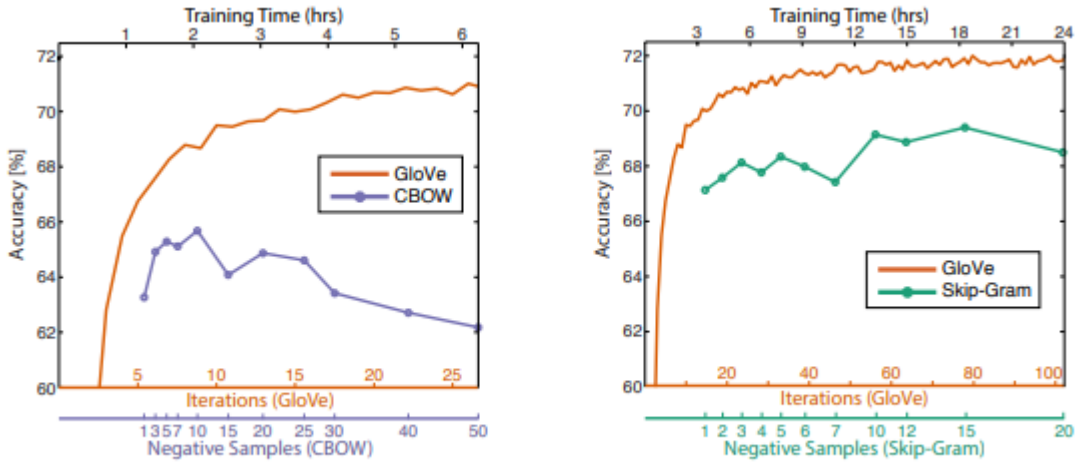
The beauty of spacy is that it will automatically provide you the averaged embeddings for words in each document without having to implement a function like we did in Word2Vec. We will leverage the same to get document features for our corpus and use k-means clustering to cluster our documents.

from sklearn.cluster import KMeans

doc_glove_vectors = np.array([nlp(str(doc)).vector for doc in norm_corpus])

km = KMeans(n_clusters=3, random_state=0) km.fit_transform(doc_glove_vectors) cluster_labels = km.labels_ cluster_labels = pd.DataFrame(cluster_labels, columns=['ClusterLabel']) pd.concat([corpus_df, cluster_labels], axis=1)

We see consistent clusters similar to what we obtained from our Word2Vec model which is good! The GloVe model claims to perform better than the Word2Vec model in many scenarios as illustrated in the following graph from the original paper by Pennington el al.



(a) GloVe vs CBOW (b) GloVe vs Skip-Gram

Convert Glove vector to word2vec (https://radimrehurek.com/gensim/scripts/glove2word2vec.htm

In [ ]:
```
from gensim.test.utils import datapath, get_tmpfile
from gensim.models import KeyedVectors
from gensim.scripts.glove2word2vec import glove2word2vec

glove_file = datapath('test_glove.txt')
tmp_file = get_tmpfile("test_word2vec.txt")
 _ = glove2word2vec(glove_file, tmp_file)

model = KeyedVectors.load_word2vec_format(tmp_file)
```

More Example (https://web.stanford.edu/class/cs224n/materials/Gensim%20word%20vector%20visualization.html)

FastText

- One major draw-back for word-embedding techniques like word2vec and glove was its inability to deal with out of corpus words.
- These embedding techniques treat word as the minimal entity and try to learn their respective embedding vector. Hence in case there is a word that does not appear in the corpus word2vec or glove fails to get their vectorized representation. However fasttext follows the same skipgram and cbow model like word2vec.

How FastText is better:

- It treats each word as composed of n-grams. That is let us say value of n is 3 for the word 'India' we have '<in', 'ind', 'ndi', 'di>' as the n-gram representation.
- And for the word 'India' we can infer the whole vector as sum of the vector representation all the character n-grams.(Here it is assumed that the hyperparameter [minn] and [maxn] value is 3,where 'minn' and 'maxn' are the smallest and largest ngram respectively). The symbols '<' and '>' are special symbols and are appended to show the start and end of the token.(P.S < her > and 'her' are not the same.)
- Fasttext can generate embedding for the words that does not appear in the training corpus.
- This can be done by adding the character n-gram of all the n-gram representations.
- For example,let's say there is a word 'commonly' in the testing dataset,but doesn't have any representation in the training set.
- But training set has vector representation of all its n-grams.
- So we can just average the vectorized representation of all its constituent n-grams.word. On the otherhand for a random word 'fgghoio' we can get the representation of by average all ngram characters(i.e 'f'+ 'g' + 'g' + 'h' + 'o' + 'i' + 'o' here we have to keep the hyperparameter minn as 1).

Applying FastText features for Machine Learning Tasks

- The gensim package has nice wrappers providing us interfaces to leverage the FastText model available under the gensim.models.fasttext module.
- Let's apply this once again on our Bible corpus and look at our words of interest and their most similar words.

In [29]:
```
import pandas as pd
import numpy as np
import re
import nltk
import matplotlib.pyplot as plt
pd.options.display.max_colwidth = 200
%matplotlib inline
```

In [30]:
```
from nltk.corpus import gutenberg
from string import punctuation

wpt = nltk.WordPunctTokenizer()
stop_words = nltk.corpus.stopwords.words('english')

def normalize_document(doc):
    # lower case and remove special characters\whitespaces
    doc = re.sub(r'[^a-zA-Z\s]', '', doc, re.I|re.A)
    doc = doc.lower()
    doc = doc.strip()
    # tokenize document
    tokens = wpt.tokenize(doc)
    # filter stopwords out of document
    filtered_tokens = [token for token in tokens if token not in stop_words]
    # re-create document from filtered tokens
    doc = ' '.join(filtered_tokens)
    return doc

normalize_corpus = np.vectorize(normalize_document)
```

In [31]:
```
bible = gutenberg.sents('bible-kjv.txt')
remove_terms = punctuation + '0123456789'

norm_bible = [[word.lower() for word in sent if word not in remove_terms] for s
norm_bible = [' '.join(tok_sent) for tok_sent in norm_bible]
norm_bible = filter(None, normalize_corpus(norm_bible))
norm_bible = [tok_sent for tok_sent in norm_bible if len(tok_sent.split()) > 2]

print('Total lines:', len(bible))
print('\nSample line:', bible[10])
print('\nProcessed line:', norm_bible[10])
```

```
Total lines: 30103

Sample line: ['1', ':', '6', 'And', 'God', 'said', ',', 'Let', 'there', 'be',
'a', 'firmament', 'in', 'the', 'midst', 'of', 'the', 'waters', ',', 'and', 'l
et', 'it', 'divide', 'the', 'waters', 'from', 'the', 'waters', '.']

Processed line: god said let firmament midst waters let divide waters waters
```

In [32]:
```
from gensim.models.fasttext import FastText

wpt = nltk.WordPunctTokenizer()
tokenized_corpus = [wpt.tokenize(document) for document in norm_bible]

# Set values for various parameters
feature_size = 100    # Word vector dimensionality
window_context = 50          # Context window size
min_word_count = 5   # Minimum word count
sample = 1e-3   # Downsample setting for frequent words

# sg decides whether to use the skip-gram model (1) or CBOW (0)
ft_model = FastText(tokenized_corpus, vector_size=feature_size, window=window_c
                    min_count=min_word_count,sample=sample, sg=1, epochs=50)


# view similar words based on gensim's FastText model
similar_words = {search_term: [item[0] for item in ft_model.wv.most_similar([se
                  for search_term in ['god', 'jesus', 'noah', 'egypt', 'john',
similar_words
```

Out[32]:
```
{'god': ['lord', 'therefore', 'unto', 'jesus', 'christ'],
 'jesus': ['christ', 'god', 'disciples', 'faith', 'grace'],
 'noah': ['methuselah', 'flood', 'shem', 'milcah', 'mahalaleel'],
 'egypt': ['land', 'pharaoh', 'egyptians', 'israel', 'lice'],
 'john': ['baptist', 'peter', 'galilee', 'baptize', 'baptized'],
 'gospel': ['preached', 'preach', 'christ', 'preaching', 'faith'],
 'moses': ['aaron', 'commanded', 'congregation', 'spake', 'tabernacle'],
 'famine': ['pestilence', 'dearth', 'sword', 'die', 'egypt']}
```

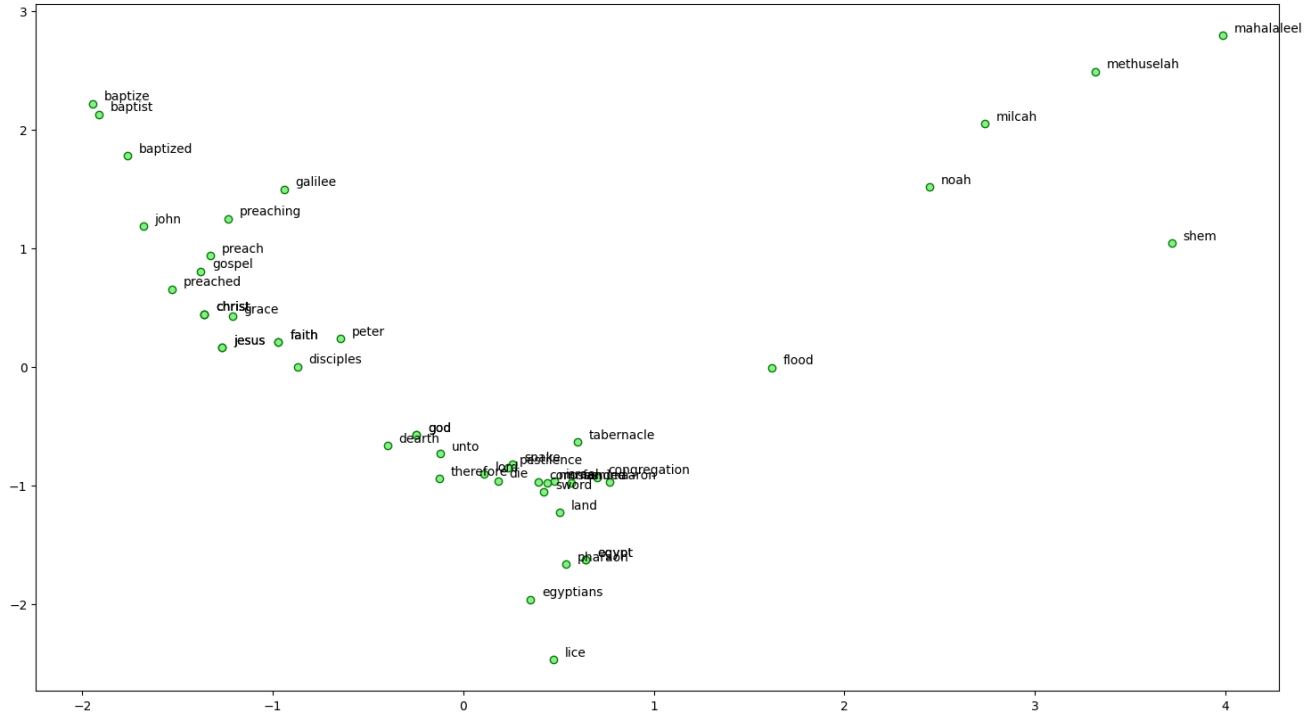
- You can see a lot of similarity in the results with our Word2Vec model with relevant similar words for each of our words of interest.
- Do you notice any interesting associations and similarities?

MOSES & AARON

TABERNACLE OF MOSES

- Let's now use Principal Component Analysis (PCA) to reduce the word embedding dimensions to 2-D and then visualize the same.

In [33]:
```
from sklearn.decomposition import PCA

words = sum([[k] + v for k, v in similar_words.items()], [])
wvs = ft_model.wv[words]

pca = PCA(n_components=2)
np.set_printoptions(suppress=True)
P = pca.fit_transform(wvs)
labels = words

plt.figure(figsize=(18, 10))
plt.scatter(P[:, 0], P[:, 1], c='lightgreen', edgecolors='g')
for label, x, y in zip(labels, P[:, 0], P[:, 1]):
    plt.annotate(label, xy=(x+0.06, y+0.03), xytext=(0, 0), textcoords='offset
```



- We can see a lot of interesting patterns! Noah, his son Shem and grandfather Methuselah are close to each other.
- We also see God associated with Moses and Egypt where it endured the Biblical plagues including famine and pestilence.
- Also Jesus and some of his disciples are associated close to each other.

- To access any of the word embeddings you can just index the model with the word as follows.

In [34]:
```
ft_model.wv['jesus']
```

Out[34]:
```
array([-0.21410207,  0.44561526, -0.1301552 ,  0.30315176,  0.2249915 ,
        0.19641946,  0.25907385,  0.21431054, -0.07418279, -0.26137155,
       -0.34035408, -0.06096689, -0.3440477 ,  0.37155524,  0.3456482 ,
       -0.32816324,  0.09839344, -0.01660585,  0.21343961, -0.00275812,
       -0.24593773, -0.06078429, -0.24258041, -0.11905143,  0.187347  ,
       -0.17225918,  0.2181918 , -0.21946089,  0.14004439,  0.35555184,
        0.27357373,  0.30776376,  0.15737171,  0.02487491,  0.09142949,
       -0.07084188,  0.04799145, -0.21592353, -0.4972383 , -0.0236594 ,
        0.3820593 , -0.00270047,  0.31200227,  0.07356375,  0.4042165 ,
        0.32702845,  0.13249317, -0.38389492,  0.3621398 ,  0.08467998,
        0.7645856 ,  0.0137694 ,  0.39625594,  0.67632014,  0.11796288,
        0.22214502,  0.36213407, -0.23399659,  0.17291893, -0.59768397,
        0.33454192,  0.08517095, -0.13195841,  0.27994925,  0.05856017,
       -0.10243247,  0.12820646,  0.03529444,  0.05228662,  0.04882605,
        0.27263704, -0.03657197,  0.04613392, -0.16623606, -0.69763297,
        0.03132856, -0.43083394,  0.1939759 , -0.47189543, -0.1207964 ,
        0.11327155, -0.14173634,  0.01996687,  0.19283177,  0.3114154 ,
        0.35132396,  0.18835379, -0.18932691,  0.04909254, -0.49333173,
        0.05465135, -0.10819462, -0.09423175, -0.06061858, -0.22236331,
        0.13477711, -0.29637215, -0.04496021,  0.26653707,  0.27178374],
      dtype=float32)
```

- Having these embeddings, we can perform some interesting natural language tasks.
- One of these would be to find out similarity between different words (entities).

In [35]:
```
print(ft_model.wv.similarity(w1='god', w2='satan'))
print(ft_model.wv.similarity(w1='god', w2='jesus'))
```

```
0.28516117
0.6709814
```

- We can see that ‘god’ is more closely associated with ‘jesus’ rather than ‘satan’ based on the text in our Bible corpus. Quite relevant!

- Considering word embeddings being present, we can even find out odd words from a bunch of words as follows.

In [36]:
```
st1 = "god jesus satan john"
print('Odd one out for [',st1, ']:',
      ft_model.wv.doesnt_match(st1.split()))
st2 = "john peter james judas"
print('Odd one out for [',st2, ']:',
      ft_model.wv.doesnt_match(st2.split()))
```

```
Odd one out for [ god jesus satan john ]: satan
Odd one out for [ john peter james judas ]: judas
```

Evaluation of Word Embeddings

How can we understand that one method for getting word embeddings is better than another? There are two types of evaluation (not only for word embeddings): intrinsic and extrinsic.

Intrinsic Evaluation: Based on Internal Properties

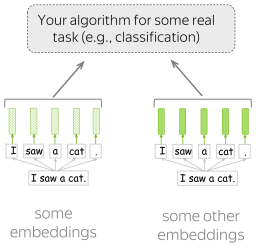
This type of evaluation looks at the internal properties of embeddings, i.e. how well they capture meaning. Specifically

How well do embeddings capture **meaning**?

Extrinsic Evaluation: On a Real Task

This type of evaluation tells which embeddings are better for the task you really care about (e.g., text classification, coreference resolution, etc.).

In this setting, you have to train the model/algorithm for the real task several times: one model for each of the embeddings you want to evaluate. Then, look at the quality of these models to decide which embeddings are better.



Model with which embeddings performs better?

Train the same model several times: one model for each embedding set

For the same dataset, you can get representations using different word embeddings

How to Choose?

One thing you have to get used to is that there is no perfect solution and no right answer for all situations: it always depends on many things.

Intrinsic:

- usually fast
- does not tell what is better in practice

Extrinsic:

- tells directly what is better in practice
- training several real-task models is expensive

Regarding evaluation, you usually care about quality of the task you want to solve. Therefore, you are likely to be more interested in extrinsic evaluation. However, real-task models usually require a lot of time and resources to train, and training several of them may be too expensive.

Analysis and Interpretability

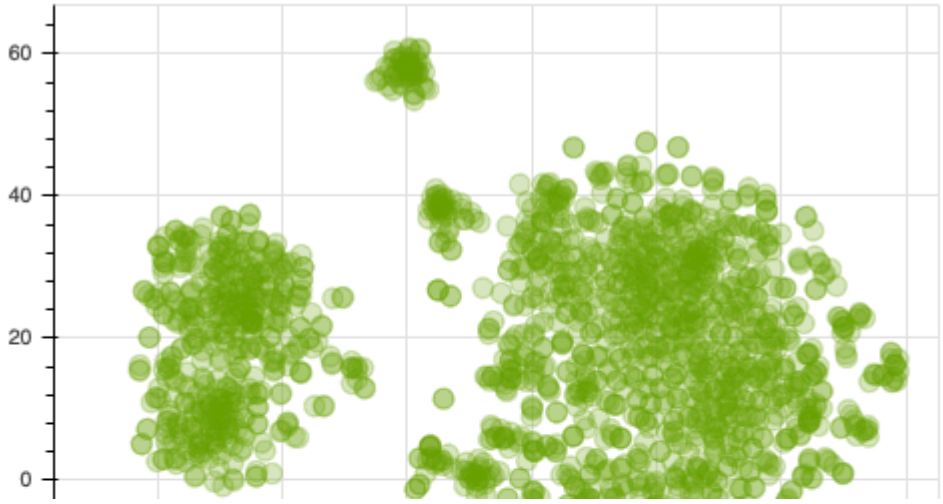
Take a Walk Through Space... Semantic Space!

Semantic spaces aim to create representations of natural language that capture meaning. We can say that (good) word embeddings form semantic space and will refer to a set of word vectors in a multi-dimensional space as "semantic space".

Below is shown semantic space formed by GloVe vectors trained on twitter data (taken from gensim). Vectors were projected to two-dimensional space using t-SNE; these are only the top-3k most frequent words.

How to: Walk through semantic space and try to find:

- language clusters: Spanish, Arabic, Russian, English. Can you find more languages?
- clusters for: food, family, names, geographical locations. What else can you find?



Nearest Neighbors

During your walk through semantic space, you probably noticed that the points (vectors) which are nearby usually have close meaning. Sometimes, even rare words are understood very well. Look at the example: the model understood that words such as leptodactylidae or litoria are close to frog.

Closest to frog:

frogs
toad
litoria
leptodactylidae
rana
lizard
eleutherodactylus

Word Similarity Benchmarks

"Looking" at nearest neighbors (by cosine similarity or Euclidean distance) is one of the methods to estimate the quality of the learned embeddings. There are several word similarity benchmarks (test sets). They consist of word pairs with a similarity score according to human judgments. The quality of embeddings is estimated as the correlation between the two similarity scores (from model and from humans).

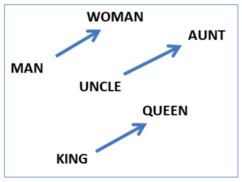
word pair		score
vulgarism	profanity	9.62
subdividing	separate	8.67
friendships	brotherhood	7.5
exceedance	probability	5.0
assigned	allow	3.5
marginalize	interact	2.5
misleading	beat	1.25
radiators	beginning	0

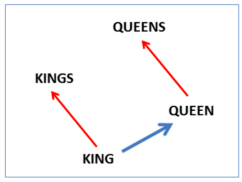
Linear Structure

While similarity results are encouraging, they are not surprising: all in all, the embeddings were trained specifically to reflect word similarity. What is surprising, is that many semantic and syntactic relationships between words are (almost) linear in word vector space.

semantic: $v(\text{king}) - v(\text{man}) + v(\text{woman}) \approx v(\text{queen})$

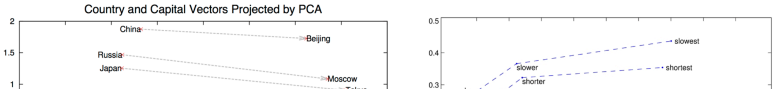
syntactic: $v(\text{kings}) - v(\text{king}) + v(\text{queen}) \approx v(\text{queens})$





For example, the difference between king and queen is (almost) the same as between man and woman. Or a word that is similar to queen in the same sense that kings is similar to king turns out to be queens. The man-woman ≈ king-queen example is probably the most popular one, but there are also many other relations and funny examples.

Below are examples for the country-capital relation and a couple of syntactic relations.



Word Analogy Benchmarks

These near-linear relationships inspired a new type of evaluation: word analogy evaluation.

Analogy: **a** is to **a*** as **b** is to __

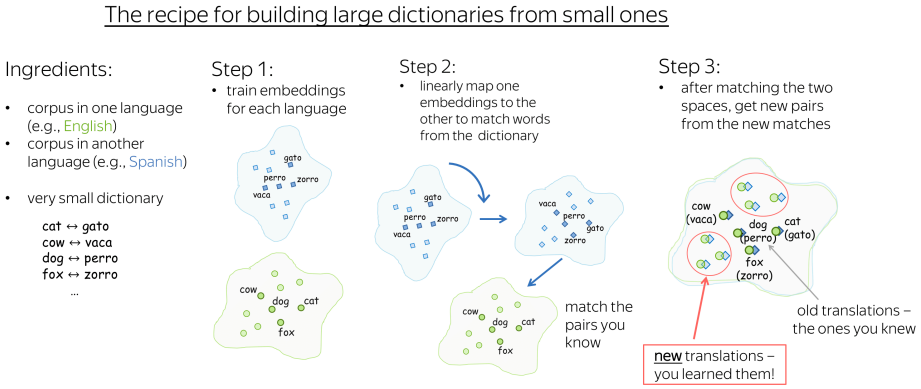
Task: $v(\mathbf{a}^*) - v(\mathbf{a}) + v(\mathbf{b}) \approx ?$ → • find the closest vector
• check if it corresponds to the correct word

Given two word pairs for the same relation, for example (man, woman) and (king, queen), the task is to check if we can identify one of the words based on the rest of them. Specifically, we have to check if the closest vector to king - man + woman corresponds to the word queen.

Now there are several analogy benchmarks; these include the standard benchmarks (MSR + Google analogy test sets) and BATS (the Bigger Analogy Test Set).

Similarities across Languages

We just saw that some relationships between words are (almost) linear in the embedding space. But what happens across languages? Turns out, relationships between semantic spaces are also (somewhat) linear: you can linearly map one semantic space to another so that corresponding words in the two languages match in the new, joint semantic space.



The figure above illustrates the approach proposed by Tomas Mikolov et al. in 2013 not long after the original Word2Vec. Formally, we are given a set of word pairs and their vector representations $\{x_i, z_i\}_{i=1}^{n}$, where x_i and z_i are vectors for i-th word in the source language and its translation in the target. We want to find a transformation matrix W such that Wz_i approximates x_i : "matches" words from the dictionary. We pick such that

$$W = \arg\min_{W} \sum_{i=1}^{n} \| Wz_i - x_i \|^2,$$

and learn this matrix by gradient descent.

In the original paper, the initial vocabulary consists of the 5k most frequent words with their translations, and the rest is learned.

In [ ]: