THINKFUL

Text Vectorization & Feature Engineering

DATA SCIENCE

Warm Up

- How many ways can you think of to convert text data into numbers that can be computed upon?

Agenda

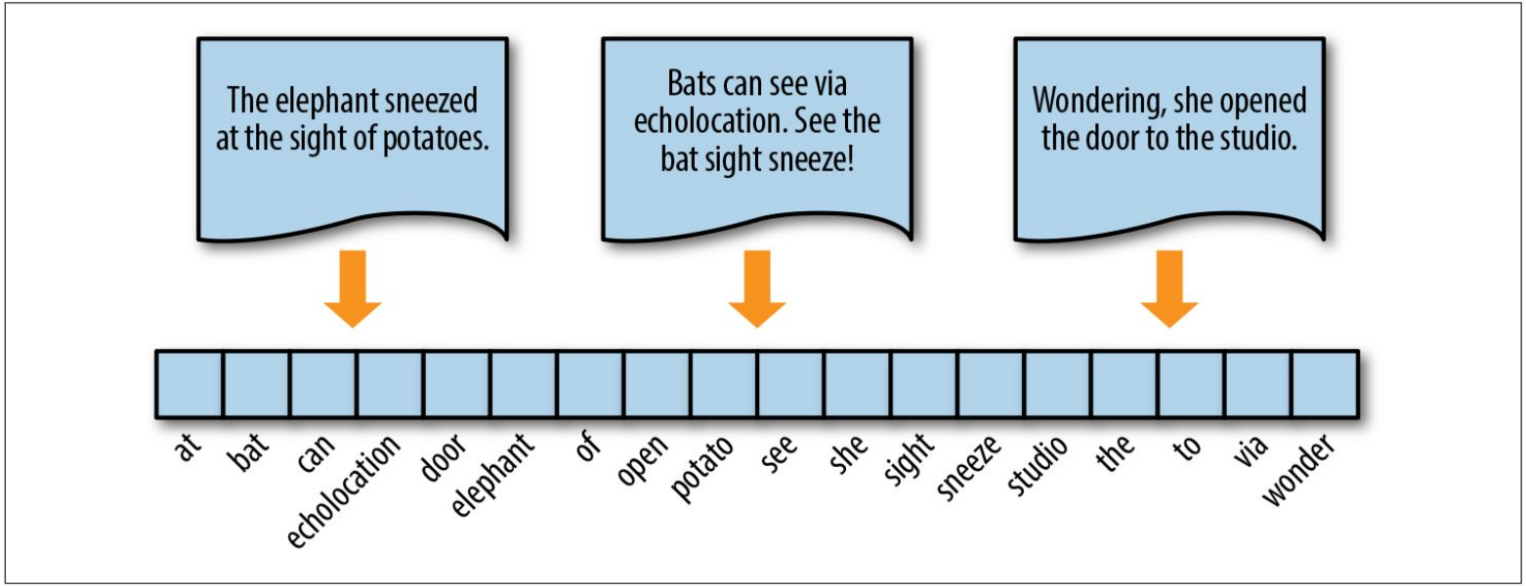
- Feature Engineering for Text Data
- Bag of Words Vectorization
- Vectorization Methods
- Count Vectorization
- One Hot Vectorization
- TF-IDF Vectorization
- Distributed Representation Vectorization
- Important Considerations for Vectorization

Feature Engineering For Text Data

- Machine learning algorithms operate on a numeric feature space, expecting input as a two-dimensional array where rows are instances and columns are features.
- In order to perform machine learning on text, we need to transform our documents into numeric vector representations.
- This process is called feature extraction and engineering or, more simply, vectorization.

Bag of Words (BOW) Vectorization

- Bag of Words vectorization represents each document in the corpus as a vector whose length is equal to the vocabulary of the corpus.

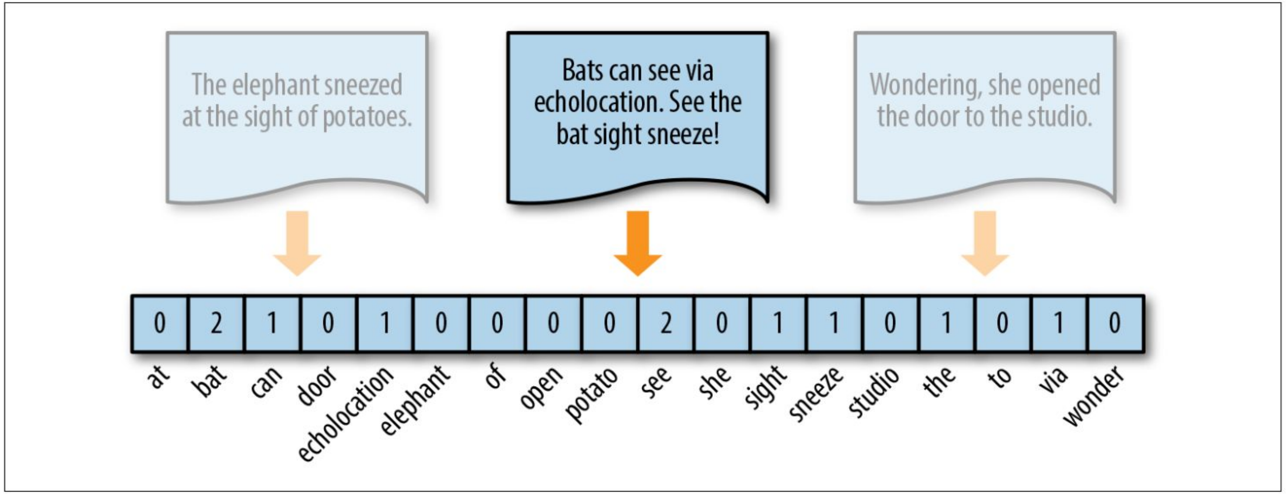


Vectorization Methods

- What should each element in the document vector be?
- There are a few different approaches, each of which extends or modifies the base bag of words model to describe semantic space.
- Count/Frequency Vectorization
- One Hot Vectorization
- TF-IDF Vectorization
- Distributed Representation Vectorization

Count Vectorization

- The simplest vector encoding method is to simply fill in the vector with the frequency of each word as it appears in the document.
- Can either be a straight integer encoding or a normalized encoding where each word is weighted by the total number of words in the document.



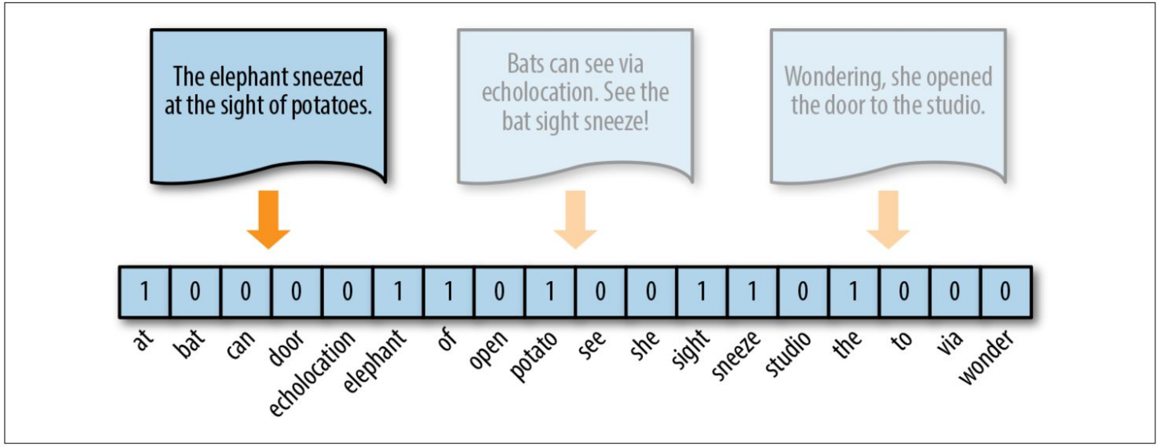
Count Vectorization

- We can use Scikit-Learn's `CountVectorizer` to perform count vectorization on a list of tokenized, normalized, and cleaned documents.
- After instantiating the CountVectorizer, we call the fit_transform method, pass it our list of documents, and save the results in a data frame.

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
vectors = vectorizer.fit_transform(documents)

count = pd.DataFrame(vectors.toarray(),
                     columns=vectorizer.get_feature_names())
```

One Hot Vectorization

- Count vectorization - tokens that occur very frequently are considered much more significant than less frequent tokens, which may not be desirable.
- A solution to this problem is one hot encoding, which simply assigns a 1 if the token exists in the document and a 0 otherwise.



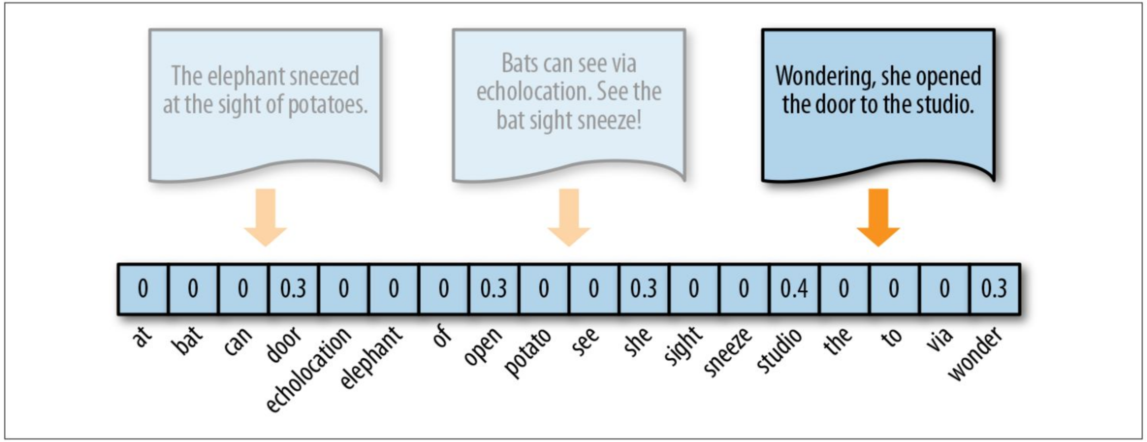
One Hot Vectorization

- We can use the same `CountVectorizer` from Scikit-Learn to perform one hot vectorization - we just need to set its `binary` parameter equal to `True`.
- We can then proceed just like we did before, calling the fit_transform, passing it the list of cleaned documents, and loading the results into a data frame.

```
vectorizer = CountVectorizer(binary=True)
vectors = vectorizer.fit_transform(documents)

one_hot = pd.DataFrame(vectors.toarray(),
                       columns=vectorizer.get_feature_names())
```

TF-IDF Vectorization

- The bag of words representations covered so far describe a document in isolation, not taking into account the context of the corpus.
- Term Frequency-Inverse Document Frequency (TF-IDF) vectorization considers the relative frequency or rareness of tokens in the document against their frequency in other documents.



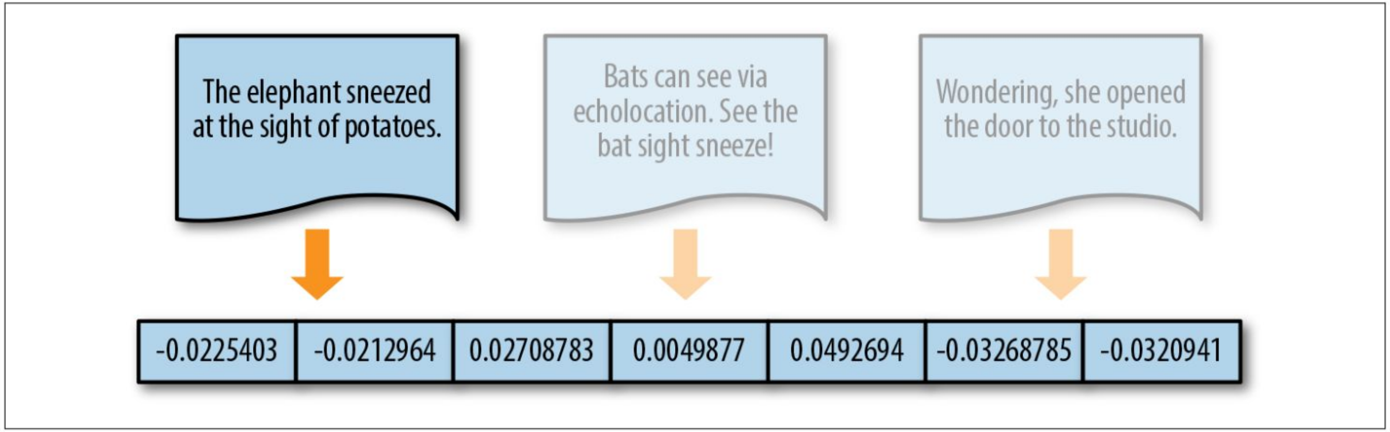
TF-IDF Vectorization

- To perform TF-IDF vectorization in Python, we need to import Scikit-Learn's `TfidfVectorizer` and use it in place of the `CountVectorizer`.
- We can then call the `fit_transform` method, pass it our list of documents, and load the results into a data frame just like we did with the other methods.

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
vectors = vectorizer.fit_transform(documents)

tfidf = pd.DataFrame(vectors.toarray(),
                     columns=vectorizer.get_feature_names())
```

Distributed Representations

- When document similarity is important, we must encode our text data along a continuous scale with a distributed representation.
- In the resulting vector, each document is represented in a feature space with word similarities embedded based on how the representation was trained and not directly tied to the document itself.



Word2Vec & Doc2Vec

- Word2Vec is a word embedding model that trains word representations based on either a continuous bag-of-words (CBOW) or skip-gram model, such that words are embedded in space along with similar words based on their context.
- Doc2Vec is an extension of Word2Vec that learns fixed-length feature representations from variable length documents, attempts to inherit the semantic properties of words, and takes into consideration the ordering of words within a narrow context.
- The Gensim library has implementations of both of these, and we will use Doc2Vec to vectorize our text.

Doc2Vec Vectorization

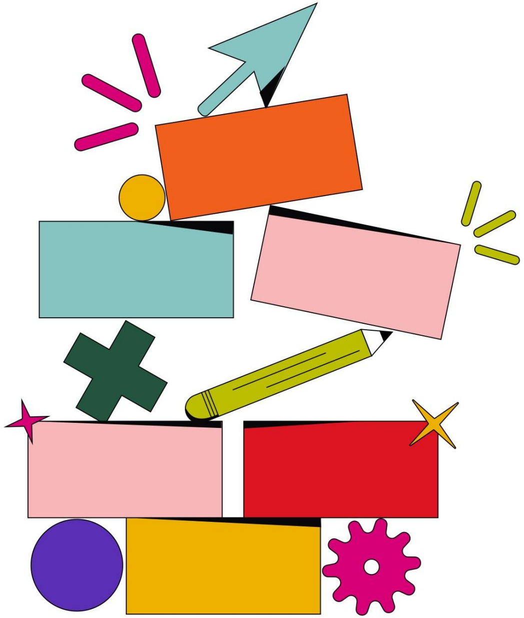
- To perform this type of vectorization, we need to import Gensim's `Doc2Vec` and `TaggedDocument` functions.
- First, we need to convert our list of documents into a list of `TaggedDocument` objects.
- Then, we can call the `Doc2Vec` function, pass it the converted documents, and load the results into a data frame as follows.

```
from gensim.models.doc2vec import Doc2Vec, TaggedDocument

documents = [TaggedDocument(doc, [i])
             for i, doc in enumerate(documents)]

model = Doc2Vec(documents)

doc2vec = pd.DataFrame([[document]+list(model[document])
                        for document in range(len(docs))]).drop(0, axis=1)
```

Considerations For Vectorization

- Which stop words to include and which to filter out.
- Whether we should we vectorize based on individual terms or n-grams.
  - Vectorizing based on individual terms leaves out potentially important word combinations and phrases, but vectorizing based on n-grams makes the data very sparse and potentially more difficult to model (especially with a limited amount of data).
- Whether we should remove infrequent words and if so, what the threshold should be.
- What vectorization approach aligns best with our data and our goals.

Questions?



THINKFUL

Summary

Brief review, should call back to the objective and make the direct connection for how the objective has now been achieved.

- Feature engineering for text data.
- An overview of the different text vectorization methods.
- How to perform each vectorization method in Python.
- Some important considerations for vectorizing text data.

Assignment

1. See Jupyter Notebook.

THINKFUL

Thank You

Text Vectorization and Feature Engineering

Warm Up

- How many ways can you think of to convert text data into numbers that can be computed upon?

High Level Agenda

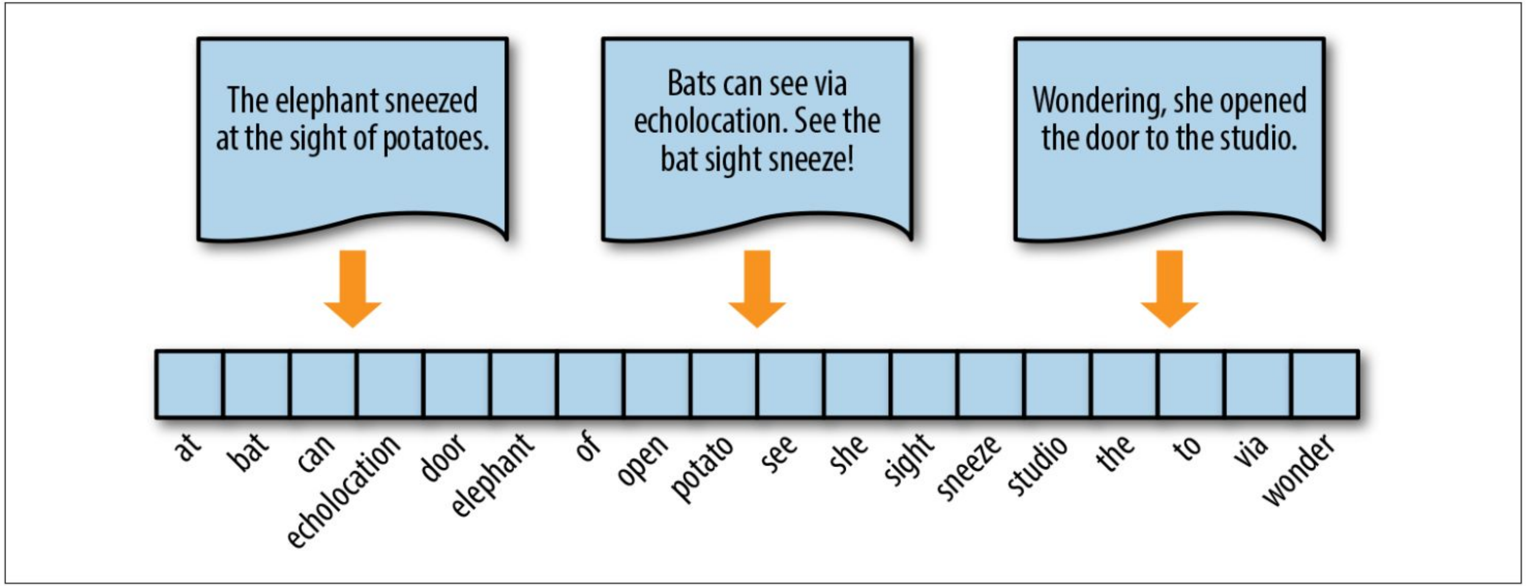
- Feature Engineering for Text Data
- Bag of Words Vectorization
- Vectorization Methods
- Count Vectorization
- One Hot Vectorization
- TF-IDF Vectorization
- Distributed Representation Vectorization
- Important Considerations for Vectorization

Feature Engineering for Text Data

- Machine learning algorithms operate on a numeric feature space, expecting input as a two-dimensional array where rows are instances and columns are features.
- In order to perform machine learning on text, we need to transform our documents into numeric vector representations.
- This process is called feature extraction and engineering or, more simply, vectorization.

Bag of Words (BOW) Vectorization

- Bag of Words vectorization represents each document in the corpus as a vector whose length is equal to the vocabulary of the corpus.



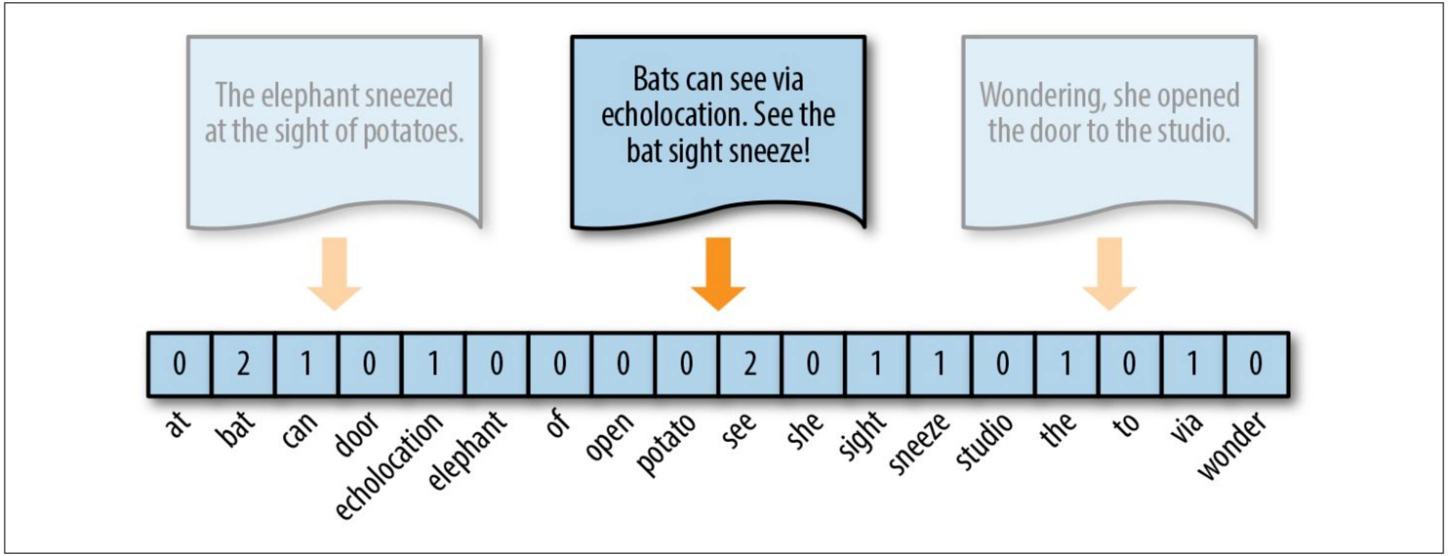
Vectorization Methods

- What should each element in the document vector be?
- There are a few different approaches, each of which extends or modifies the base bag of words model to describe semantic space.

- Count/Frequency Vectorization
- One Hot Vectorization
- TF-IDF Vectorization
- Distributed Representation Vectorization

Count Vectorization

- The simplest vector encoding method is to simply fill in the vector with the frequency of each word as it appears in the document.
- Can either be a straight integer encoding or a normalized encoding where each word is weighted by the total number of words in the document.



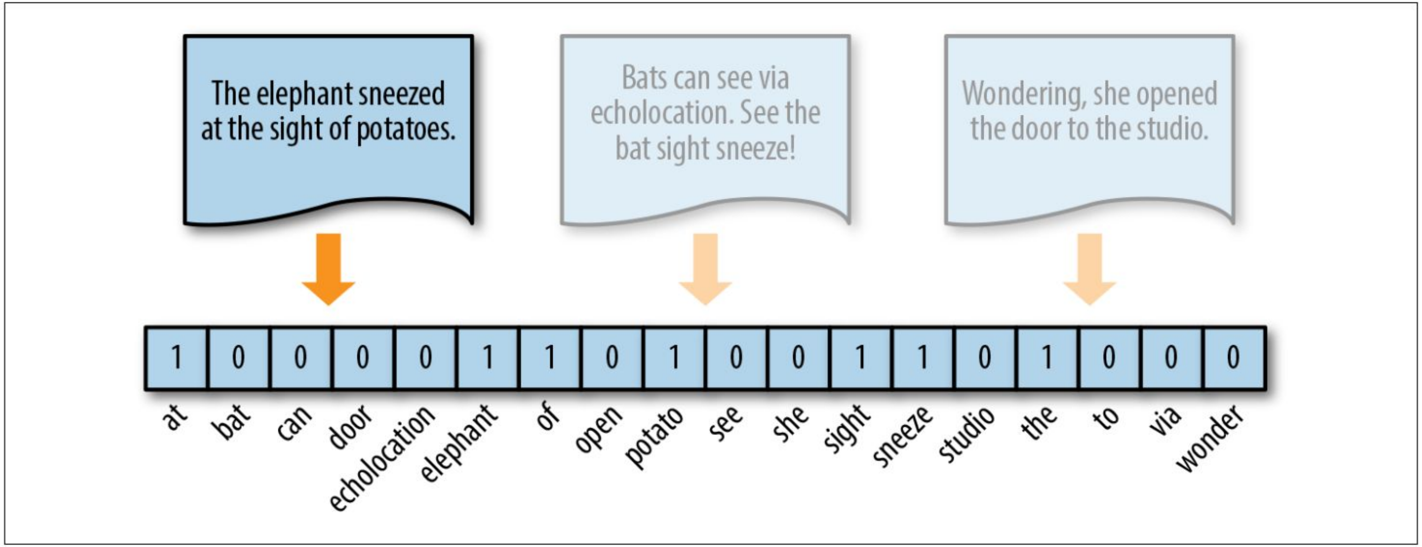
Count Vectorization

- We can use Scikit-Learn's `CountVectorizer` to perform count vectorization on a list of tokenized, normalized, and cleaned documents.
- After instantiating the CountVectorizer, we call the fit_transform method, pass it our list of documents, and save the results in a data frame.

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
vectors = vectorizer.fit_transform(documents)

count = pd.DataFrame(vectors.toarray(),
                     columns=vectorizer.get_feature_names())
```

One Hot Vectorization

- Count vectorization - tokens that occur very frequently are considered much more significant than less frequent tokens, which may not be desirable.
- A solution to this problem is one hot encoding, which simply assigns a 1 if the token exists in the document and a 0 otherwise.



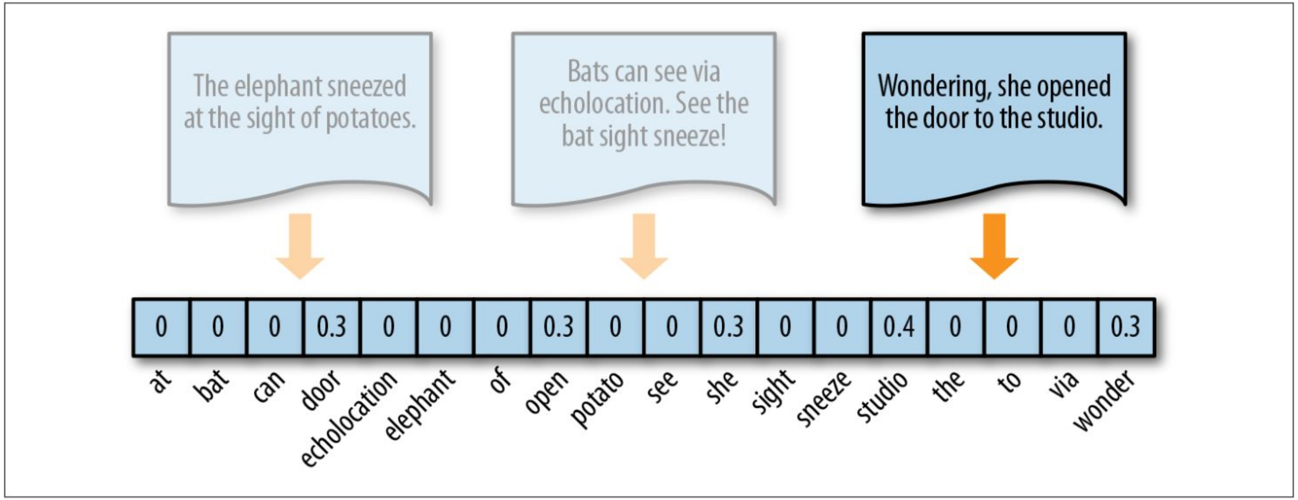
One Hot Vectorization

- We can use the same `CountVectorizer` from Scikit-Learn to perform one hot vectorization - we just need to set its `binary` parameter equal to `True`.
- We can then proceed just like we did before, calling the fit_transform, passing it the list of cleaned documents, and loading the results into a data frame.

```
vectorizer = CountVectorizer(binary=True)
vectors = vectorizer.fit_transform(documents)

one_hot = pd.DataFrame(vectors.toarray(),
                       columns=vectorizer.get_feature_names())
```

TF-IDF Vectorization

- The bag of words representations covered so far describe a document in isolation, not taking into account the context of the corpus.
- Term Frequency-Inverse Document Frequency (TF-IDF) vectorization considers the relative frequency or rareness of tokens in the document against their frequency in other documents.



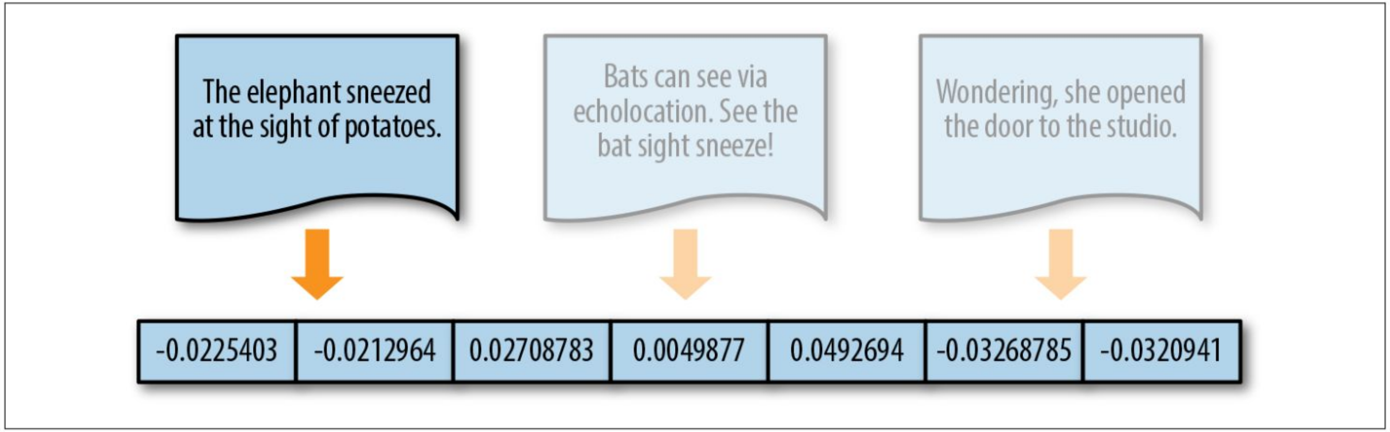
TF-IDF Vectorization

- To perform TF-IDF vectorization in Python, we need to import Scikit-Learn's `TfidfVectorizer` and use it in place of the `CountVectorizer`.
- We can then call the `fit_transform` method, pass it our list of documents, and load the results into a data frame just like we did with the other methods.

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
vectors = vectorizer.fit_transform(documents)

tfidf = pd.DataFrame(vectors.toarray(),
                     columns=vectorizer.get_feature_names())
```

Distributed Representations

- When document similarity is important, we must encode our text data along a continuous scale with a distributed representation.
- In the resulting vector, each document is represented in a feature space with word similarities embedded based on how the representation was trained and not directly tied to the document itself.



Word2Vec and Doc2Vec

- Word2Vec is a word embedding model that trains word representations based on either a continuous bag-of-words (CBOW) or skip-gram model, such that words are embedded in space along with similar words based on their context.
- Doc2Vec is an extension of Word2Vec that learns fixed-length feature representations from variable length documents, attempts to inherit the semantic properties of words, and takes into consideration the ordering of words within a narrow context.
- The Gensim library has implementations of both of these, and we will use Doc2Vec to vectorize our text.

Doc2Vec Vectorization

- To perform this type of vectorization, we need to import Gensim's `Doc2Vec` and `TaggedDocument` functions.
- First, we need to convert our list of documents into a list of `TaggedDocument` objects.
- Then, we can call the `Doc2Vec` function, pass it the converted documents, and load the results into a data frame as follows.

```
from gensim.models.doc2vec import Doc2Vec, TaggedDocument

documents = [TaggedDocument(doc, [i])
             for i, doc in enumerate(documents)]

model = Doc2Vec(documents)

doc2vec = pd.DataFrame([[document]+list(model[document])
                        for document in range(len(docs))]).drop(0, axis=1)
```

Important Considerations for Vectorization

- Which stop words to include and which to filter out.

- Whether we should we vectorize based on individual terms or n-grams.
  - Vectorizing based on individual terms leaves out potentially important word combinations and phrases, but vectorizing based on n-grams makes the data very sparse and potentially more difficult to model (especially with a limited amount of data).

- Whether we should remove infrequent words and if so, what the threshold should be.

- What vectorization approach aligns best with our data and our goals.

Questions?

Recap

In this session, we covered:

- Feature engineering for text data.
- An overview of the different text vectorization methods.
- How to perform each vectorization method in Python.
- Some important considerations for vectorizing text data.

Assignment

- See Jupyter Notebook.