DEPAUL UNIVERSITY

DSC478: Programming Machine Learning Applications

Roselyne Tchoua

rtchoua@depaul.edu

School of Computing, CDM, DePaul University

Text Categorization

- Assigning documents to a fixed set of categories
- Applications:
  - Web pages
    - Recommending pages
    - Classification hierarchies
    - Categorizing bookmarks
  - Newsgroup Messages /News Feeds / Micro-blog Posts
    - Recommending messages, posts, tweets, etc.
    - Message filtering
  - News articles
    - Personalized news
    - Sentiment analysis
  - Email messages
    - Routing
    - Folderizing
    - Spam filtering

Learning for Text Categorization

- Text Categorization is an application of classification

- Typical Learning Algorithms:
  - **Bayesian (naïve)**
  - **Relevance Feedback (Rocchio)**
  - **Nearest Neighbor**
  - **Neural network**
  - **Support Vector Machines (SVM)**

Recall … Similarity Metric

- Nearest neighbor method depends on a similarity (or distance) metric
- Simplest for continuous *m*-dimensional instance space is *Euclidean distance*
- Simplest for *m*-dimensional binary instance space is *Hamming distance* (number of feature values that differ)
- For text, cosine similarity of **TF-IDF** weighted vectors is typically most effective

Basic Text Processing

- Parse documents to recognize structure and meta-data
  - e.g., title, date, other fields, html tags, etc.
- Scan for word tokens
  - lexical analysis to recognize keywords, numbers, special characters, etc.
- Stopword removal
  - common words such as “the”, “and”, “or” which are not semantically meaningful in a document
- Stem words
  - morphological processing to group word variants (e.g., “compute”, “computer”, “computing”, “computes”, … can be represented by a single **stem** “comput” in the index)
- Assign weight to words
  - using frequency in documents and across documents
- Store Index
  - Stored in a Term-Document Matrix (“inverted index”) which stores each document as a vector of keyword weights

Basic Text Representation

- **Bag-of-words**: represent a document as a "bag-of-the-words" within it with their corresponding frequency
- Used for feature engineering in text data
  - Use 0 for words from the corpustvhat do not appear in a doc.
  - Corpus: all the text data/words in all your documents

"**John**","likes","to","watch","movies","Mary","likes","**movies**","**too**"
"Mary","**also**","likes","to","watch","**football**","**games**"

BoW1 = {"John":**1**,"likes":**2**,"to":**1**,"watch":**1**,"movies":**2**,"Mary":**1**,"too":**1**};
BoW2 = {"Mary":1,"also":1,"likes":1,"to":1,"watch":1,"football":1,"games":1};

(1) [**1**, **2**, **1**, **1**, **2**, **1**, **1**, **0**, **0**, **0**]
(2) [**0**, 1, 1, 1, **0**, 1, **0**, **1**, **1**, **1**]

also, *football* and games dot not appear in 1st doc.
John, *movies* and too dot not appear in 2nd doc.

Zipf Law & Why the Log Function?

- Zipf's law is an empirical law that states that frequency distribution of data in social and physical sciences (e.g., some corpus* of natural language utterances) often follows a Power-law distribution. More specifically, **the frequency of a term is inversely proportional to some power of its rank**.
  - **Most frequent word will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc**.
  - In practice, this means less frequent data are lumped together at one end of the frequency spectrum, whose frequencies aren't too distinguishable. Applying log to the frequencies, "smooths" out the lumping
- It's often desirable to have scoring functions that are **additive**.

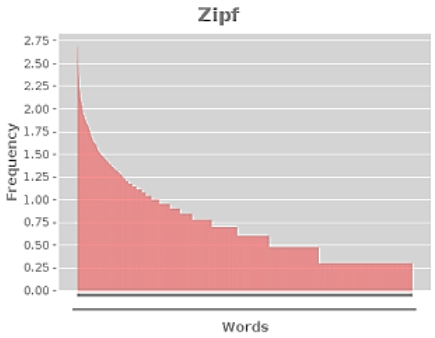
*** corpus: body of text you are analyzing**

Zipf Law & Why the Log Function?

- Zipf's law is an empirical law that states that frequency distribution of data in social and physical sciences (e.g., some corpus* of natural language utterances) often follows a Power-law distribution. More specifically, the frequency of a term is inversely proportional to some power of its rank.
- It's often desirable to have scoring functions that are additive.

- *N* be the number of elements;
- *k* be their rank;
- *s* be the value of the exponent characterizing the distribution.
- S=1 English Language

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N} (1/n^s)}$$



*** corpus: body of text you are analyzing**

TF*IDF Weights

- tf x idf measure:
  - term frequency (tf)
  - inverse document frequency (idf) -- a way to deal with the problems of the Zipf distribution
    - There are always a few very frequent tokens that are not good discriminators ("**stopwords**").
      - English examples: *to, from, on, and, the, ...*
    - There are always a large number of tokens that occur once and can mess up algorithms.
    - **Medium frequency words most descriptive**
  - Want to weight terms highly if they are
    - frequent in relevant documents … **BUT**
    - infrequent in the collection as a whole
- Goal: assign a tf x idf weight to each term in each document

TF*IDF Weights

$$w_{ik} = tf_{ik} * \log(N / n_k)$$

$T_k = \text{term } k \text{ in document } D_i$

$tf_{ik} = \text{frequency of term } T_k \text{ in document } D_i$

$idf_k = \text{inverse document frequency of term } T_k \text{ in } C$

$N = \text{total number of documents in the collection } C$

$n_k = \text{the number of documents in } C \text{ that contain } T_k$

$$idf_k = \log\left(\frac{N}{n_k}\right)$$

Inverse Document Frequency

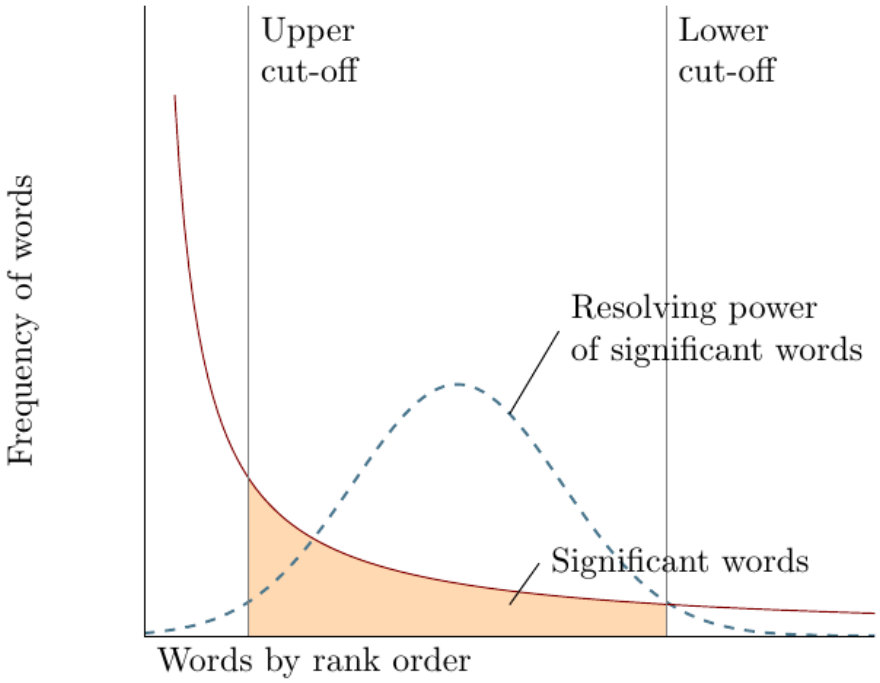
IDF provides high values for rare words and low values for common words

$$\log\left(\frac{10000}{10000}\right) = 0$$

$$\log\left(\frac{10000}{5000}\right) = 0.301$$

$$\log\left(\frac{10000}{20}\right) = 2.698$$

$$\log\left(\frac{10000}{1}\right) = 4$$



TF * IDF Example

The initial Term x Doc matrix (Inverted Index)

	Doc 1	Doc 2	Doc 3	Doc 4	Doc 5	Doc 6
T1	0	2	4	0	1	0
T2	1	3	0	0	0	2
T3	0	1	0	2	0	0
T4	3	0	1	5	4	0
T5	0	4	0	0	0	1
T6	2	7	2	1	3	0
T7	1	0	0	5	5	1
T8	0	1	1	0	0	3

df	idf = log2(N/df)
3	1.00
3	1.00
2	1.58
4	0.58
2	1.58
5	0.26
4	0.58
3	1.00

Documents represented as vectors of words

tf x idf Term x Doc matrix

	Doc 1	Doc 2	Doc 3	Doc 4	Doc 5	Doc 6
T1	0.00	2.00	4.00	0.00	1.00	0.00
T2	1.00	3.00	0.00	0.00	0.00	2.00
T3	0.00	1.58	0.00	3.17	0.00	0.00
T4	1.74	0.00	0.58	2.90	2.32	0.00
T5	0.00	6.34	0.00	0.00	0.00	1.58
T6	0.53	1.84	0.53	0.26	0.79	0.00
T7	0.58	0.00	0.00	2.92	2.92	0.58
T8	0.00	1.00	1.00	0.00	0.00	3.00

TF*IDF Limitations

- TF-IDF comes with some limitations and the biggest one is that it relies on a **clean set of data** (eg. minimal spelling errors, acronyms etc.) that is large enough, and that your dataset adheres to *Zipf's Law*.
- Its a huge assumption which doesn't really hold all the time, because real world data is complex
  - e.g., in scientific data, some rare words are the most important even though they appear only once, have "weird" characters and are easy to misspell etc.
  - e.g., what about tweets?

Using Rocchio Method

- Rocchio method is typically used for relevance feedback in information retrieval
  - **Relevance feedback** is a feature of some information retrieval systems. The idea behind relevance feedback is to take the results that are initially returned from a given query, to gather user feedback, and to use information about whether or not those results are relevant to perform a new query.

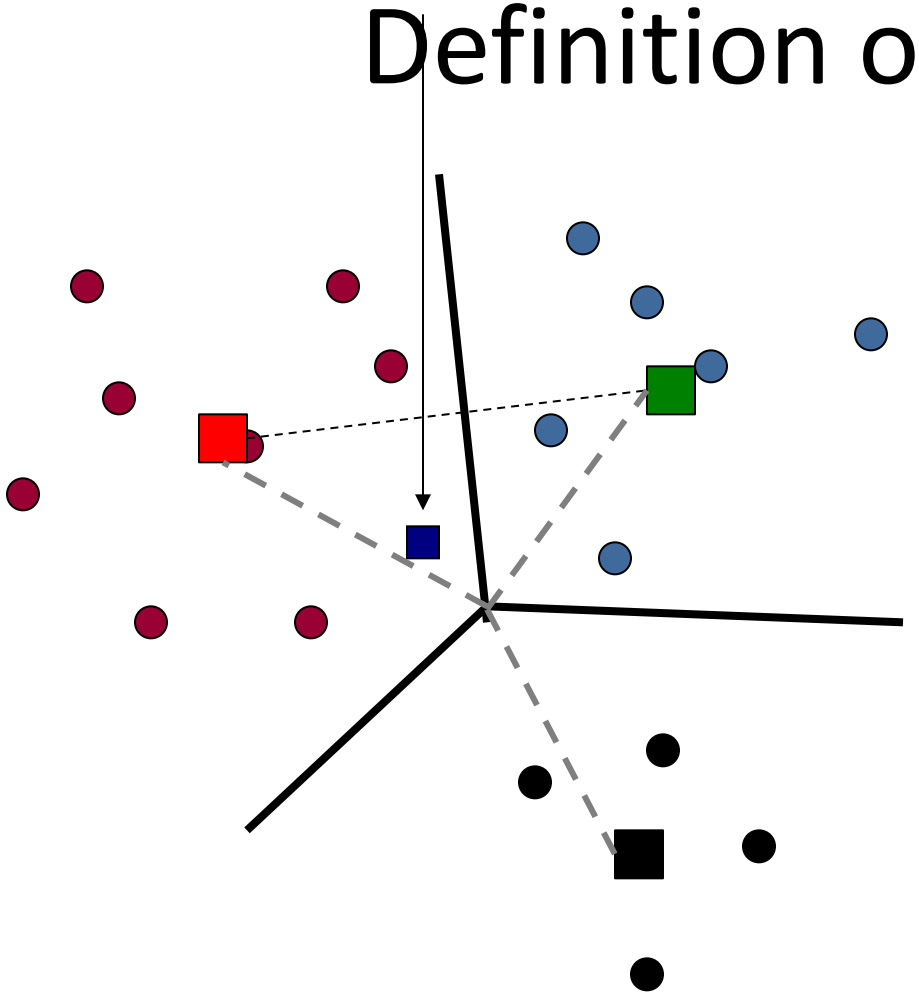
$$\vec{q}_m = \alpha \vec{q}_0 + \beta \frac{1}{|D_r|} \sum_{\vec{d}_j \in D_r} \vec{d}_j - \gamma \frac{1}{|D_{nr}|} \sum_{\vec{d}_j \in D_{nr}} \vec{d}_j$$

- q_m = modified query vector; q_0 = original query vector; α,β,γ: weights (hand-chosen or set empirically); D_r = set of known relevant doc vectors; D_{nr} = set of known irrelevant doc vectors
- New query moves toward relevant documents and away from irrelevant documents

Using Rocchio Method

- Rocchio method can be adapted for text categorization
- Use standard TF/IDF weighted vectors to represent text documents
- For each category, compute a ***prototype*** vector by summing the vectors of the training documents in the category
- Assign test documents to the category with the closest prototype vector based on cosine similarity.

Definition of a Centroid



$$\vec{\mu}(c) = \frac{1}{|D_c|} \sum_{d \in D_c} \vec{v}(d)$$

Government

Science

Arts

- Where Dc is the set of all documents that belong to class c and v(d) is the vector space representation of d.
- Note that centroid will in general not be a unit vector even when the inputs are unit vectors.

Rocchio Text Categorization Algorithm (Training)

Assume the set of categories is $\{c_1, c_2, \ldots c_n\}$

For i from 1 to n let $\mathbf{p}_i = <0, 0, \ldots, 0>$ (*init. prototype vectors*)

For each training example $<x, c(x)> \in D$

Let $\mathbf{d}$ be the TF/IDF term vector for doc x

Let $i = j$ where $c_j = c(x)$

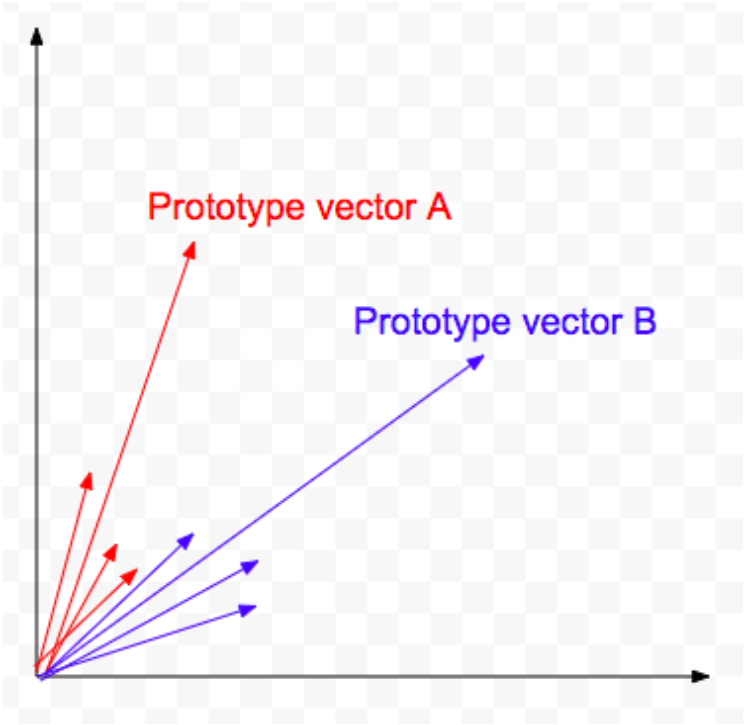
(*sum all the document vectors in c_i to get $\boldsymbol{p}_i$*)

Let $\mathbf{p}_i = \mathbf{p}_i + \mathbf{d}$

Rocchio Text Categorization Algorithm (Testing)

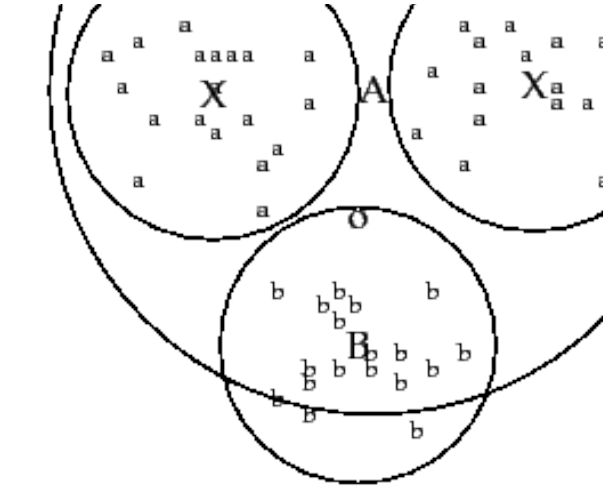
Given test document *x*

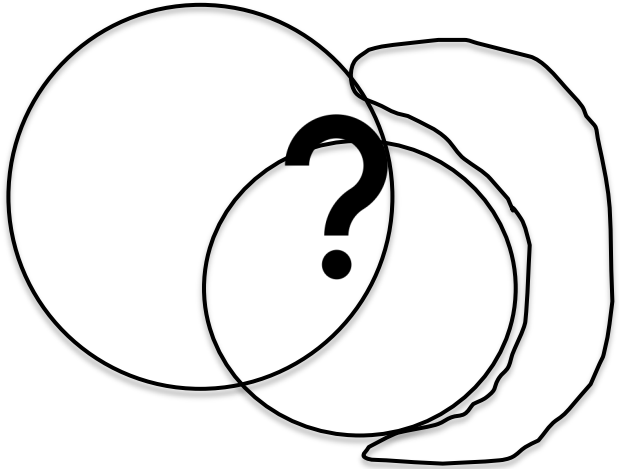
Let **d** be the TF/IDF term vector for *x*
Let *m* = –2 (*init. maximum cosSim*)
For *i* from 1 to *n*:
    (*compute similarity to prototype vector*)
    Let *s* = cosSim(**d**, $\mathbf{p}_i$)
    if *s* > *m*
        let *m* = *s*
        let *r* = c_i (*update most similar class prototype*)
Return class *r*



Rocchio Properties

- Does not guarantee a consistent hypothesis
  - Why? **Example maybe equally close to either prototype**.
- Forms a simple generalization of the examples in each class (a *prototype*)
- Prototype vector does not need to be averaged or otherwise normalized for length since cosine similarity is insensitive to vector length
- Classification is based on similarity to class prototypes
- Rocchio classification is simple and efficient, but inaccurate if classes are not approximately spheres with similar radii
- Rocchio also struggle with polymorphic (disjunctive categories see image)

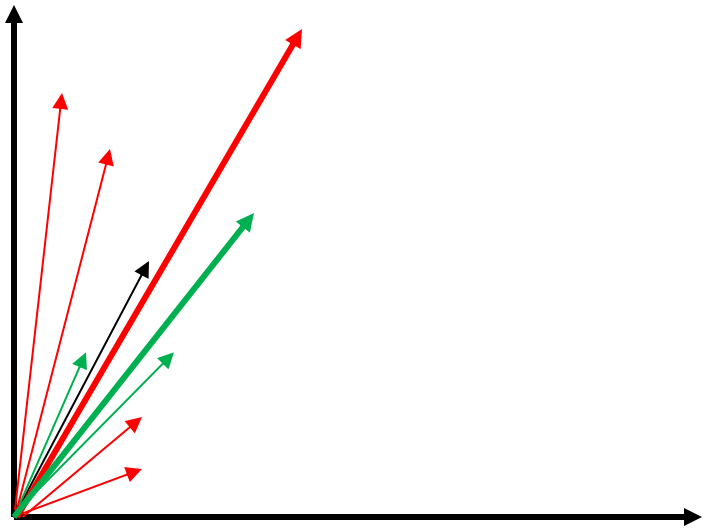




Top figure: https://nlp.stanford.edu/IR-book/html/htmledition/rocchio-classification-1.html

Rocchio Properties

- Does not guarantee a consistent hypothesis
  - Why? **Example maybe equally close to either prototype**.
- Forms a simple generalization of the examples in each class (a *prototype*)
- Prototype vector does not need to be averaged or otherwise normalized for length since cosine similarity is insensitive to vector length
- Classification is based on similarity to class prototypes
- Rocchio classification is simple and efficient, but inaccurate if classes are not approximately spheres with similar radii
- Rocchio also struggle with polymorphic (disjunctive categories see image)



In this example, the category for the new document (in black will be red instead of green based on cosine similarity to the prototypes (big red and green arrows))

Scikit-Learn Example

Examples

```
>>> from sklearn.neighbors.nearest_centroid import NearestCentroid
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> clf = NearestCentroid()
>>> clf.fit(X, y)
NearestCentroid(metric='euclidean', shrink_threshold=None)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

KNN for Text

Training:

For each each training example $\langle x, c(x) \rangle \in D$

Compute the corresponding TF-IDF vector, $\mathbf{d}_x$, for document x

Test instance *y*:

Compute TF-IDF vector **d** for document *y*

For each $\langle x, c(x) \rangle \in D$

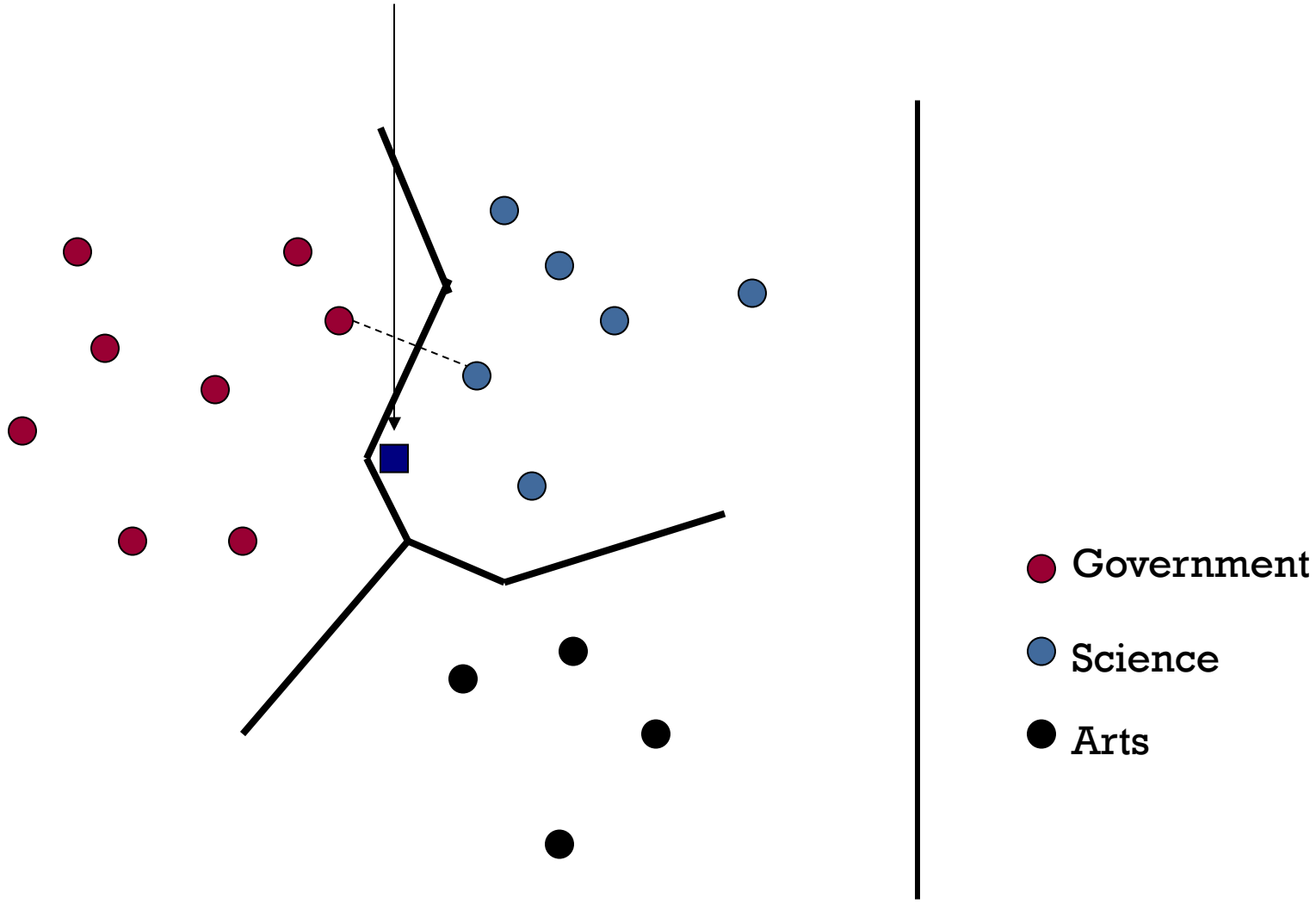
Let $s_x = \text{cosSim}(\mathbf{d}, \mathbf{d}_x)$

Sort examples, *x*, in *D* by decreasing value of s_x

Let *N* be the first *k* examples in D. (*get most similar neighbors*)

Return the majority class of examples in *N*

KNN for Text



KNN Discussion

- No feature selection necessary
- No training necessary
- Scales well with large number of classes
  - Don't need to train n classifiers for n classes
- Done naively, very expensive at test time
- In most cases it's more accurate than Rocchio/ KNN can handle non-spherical and other complex classes better than Rocchio.

Scikit-Learn Example

Examples

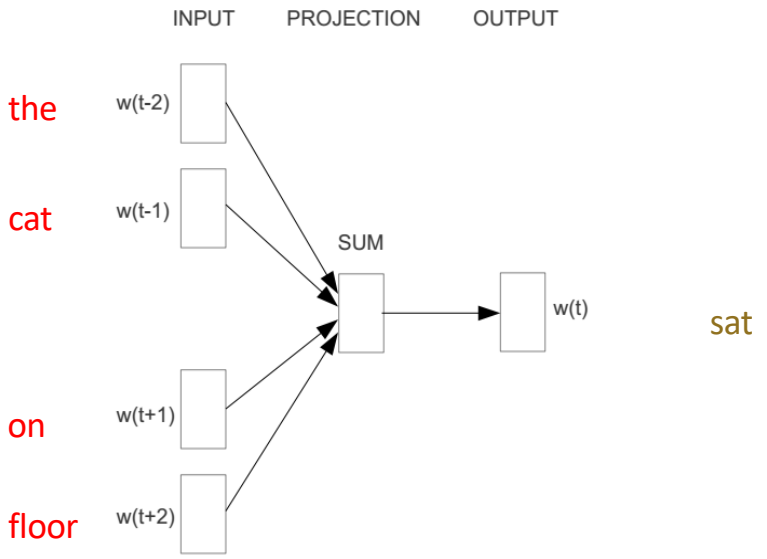
```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import KNeighborsClassifier
>>> neigh = KNeighborsClassifier(n_neighbors=3)
>>> neigh.fit(X, y)
KNeighborsClassifier(...)
>>> print(neigh.predict([[1.1]]))
[0]
>>> print(neigh.predict_proba([[0.9]]))
[[0.66666667 0.33333333]]
```

Word Embedding (Neural Network)

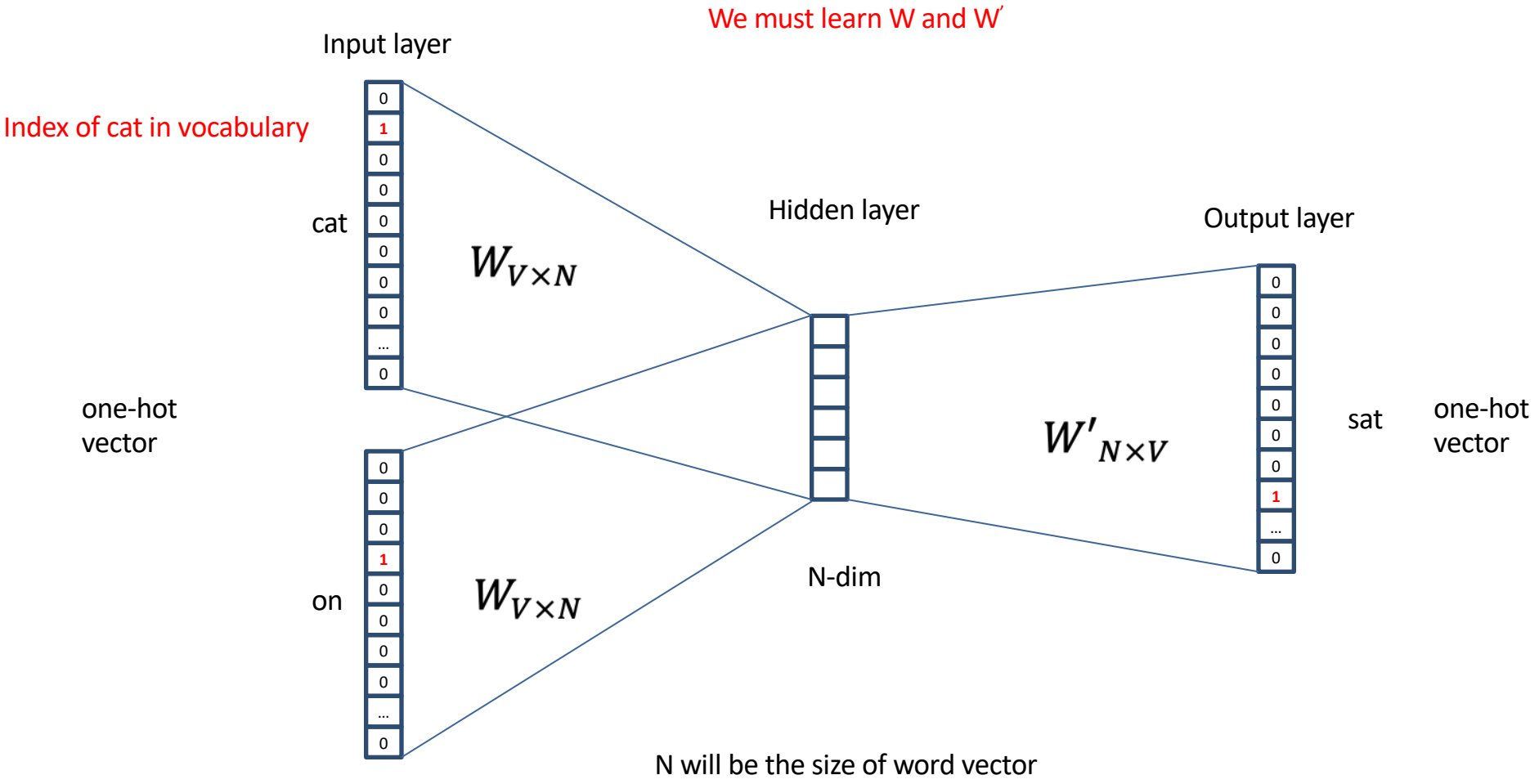
- Represent each word with a low-dimensional vector
- Word similarity = vector similarity
- Key idea: **Predict surrounding words of every word**
- Faster and can easily incorporate a new sentence/document or add a word to the vocabulary

e.g., “The cat sat on floor”

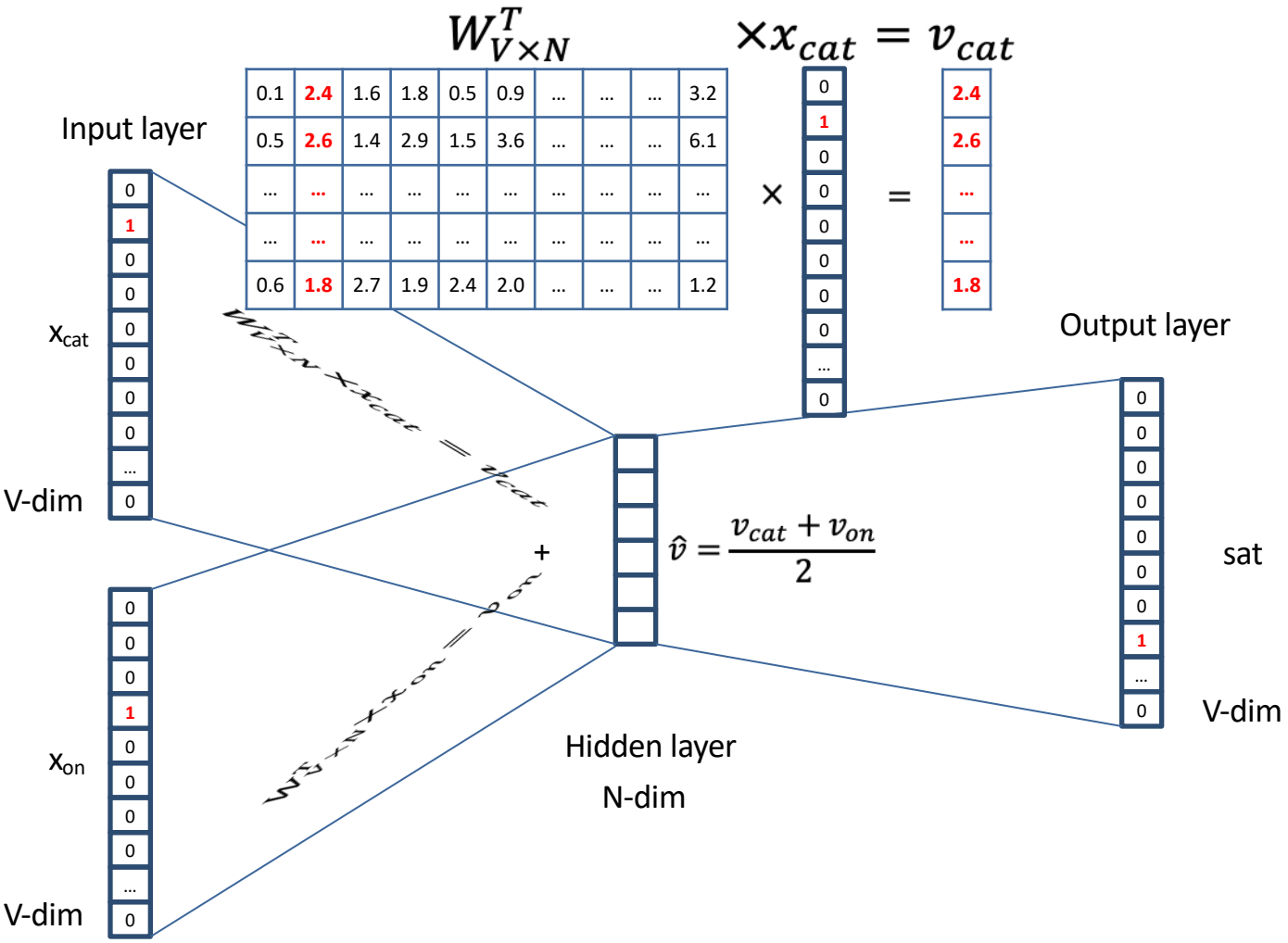
Window size = 2



Word2Vec


We must learn W and W′
Input layer
Index of cat in vocabulary
cat
one-hot vector
on
$W_{V\times N}$
$W_{V\times N}$
Hidden layer
N-dim
$W'_{N\times V}$
Output layer
sat
one-hot vector
N will be the size of word vector

Word2Vec


$W^T_{V\times N}$
$\times x_{cat} = v_{cat}$
Input layer
x_{cat}
V-dim
x_{on}
V-dim
Hidden layer
N-dim
$\hat{v} = \frac{v_{cat} + v_{on}}{2}$
Output layer
sat
V-dim

Applications

Word Analogies

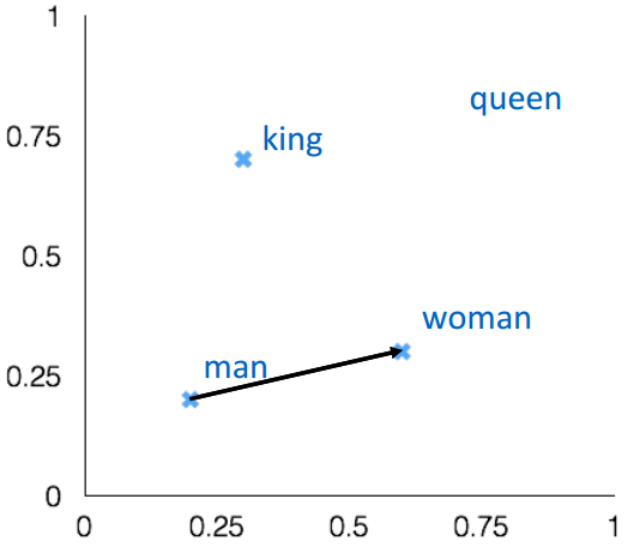
Test for linear relationships, examined by Mikolov et al. (2014)

a:b :: c:?

$$d = \arg\max_x \frac{(w_b - w_a + w_c)^T w_x}{||w_b - w_a + w_c||}$$

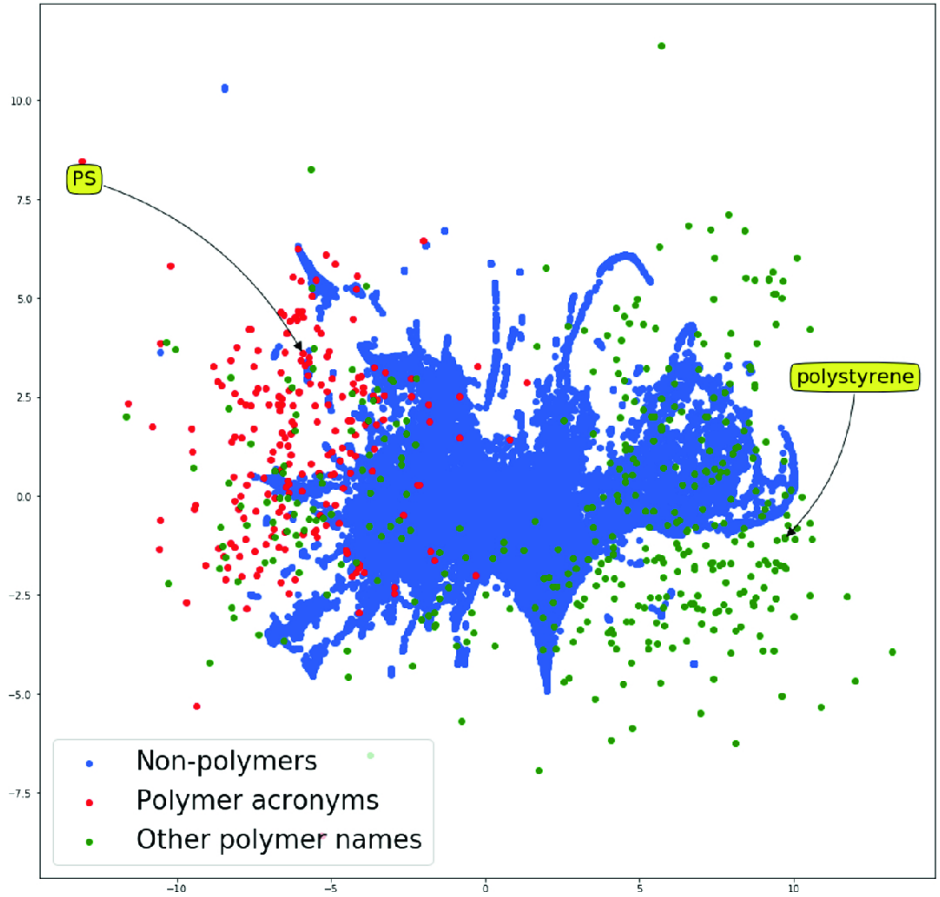
man:woman :: king:?

+	king	[0.30 0.70]
-	man	[0.20 0.20]
+	woman	[0.60 0.30]
	queen	[0.70 0.80]



Applications

- Visualizing polymer names, acronyms and all the other words in 100 documents
- Visualization done with tsne
  - https://projector.tensorflow.org/



Tchoua, Roselyne B., et al. "Creating training data for scientific named entity recognition with minimal human effort." *International Conference on Computational Science*. Springer, Cham, 2019.

Word Embedding to Sentences & Paragraphs

- Simple approach: take avg of the word2vecs of its words
- Another approach: Paragraph vector (2014, Quoc Le, Mikolov)
  - Extend word2vec to text level
  - Also, two models: add paragraph vector as the input
- Other Applications:
  - Search, e.g., query expansion
  - Sentiment analysis
  - Classification
  - Clustering

Gensim and FastText

- Easiest way to use it is via the Gensim libarary for Python

  https://radimrehurek.com/gensim/models/word2vec.html

Word2vec supports several word similarity tasks out of the box:

```
model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
[('queen', 0.50882536)]
model.doesnt_match("breakfast cereal dinner lunch";.split())
'cereal'
model.similarity('woman', 'man')
0.73723527
```

- FastText in Python (and C):
  https://github.com/facebookresearch/fastText/tree/master/python

More NN-Based Word Representation

- Word2Vec: 2 basic neural network models:
  - Continuous Bag of Word (CBOW): use a window of word to predict the middle word
  - Skip-gram (SG): use a word to predict the surrounding ones in window. Usually better at predicting rare words
- FastText improves Word2Vec with integration of character n-gram embedding
- (Large) Pretrained word vectors:
  - Glove: https://nlp.stanford.edu/projects/glove/
  - ElMo: https://allennlp.org/elmo
  - BERT: https://github.com/google-research/bert
  - …

Limitations

- Can take a long time to train
- Needs a lot of (labelled) data to get insight
  - Easier to get for “regular” corpus
  - Not so simple for scientific text
- Hard to understand (recall black box)
  - OpenAI GPT-2 language model
  - https://openai.com/blog/better-language-models/
  - **Read their “*Release Strategy*” for more info**
    - **Ethical concerns**