AICOSS - AI Special Program

Text Mining Theory

Content

- **Introduction to Text Mining**
    - Terms
    - Applications
    - Challenges
    - Performance Aims
- **Gathering Text Data**
- **Preprocessing Text**
    - Tokenization
    - Normalization
    - Stop Word Removal
- **Vectorizing Text (Feature Generation)**
    - Frequency-based Approaches
    - Prediction-based Approaches

Content

- **Introduction to Text Mining**
  - Terms
  - Applications
  - Challenges
  - Performance Aims
- **Gathering Text Data**
- **Preprocessing Text**
  - Tokenization
  - Normalization
  - Stop Word Removal
- **Vectorizing Text (Feature Generation)**
  - Frequency-based Approaches
  - Prediction-based Approaches

Terms

Text Mining

- Umbrella term enclosing activities that aim to discover previously unknown information in (mostly) unstructured natural text.

Unstructured Data

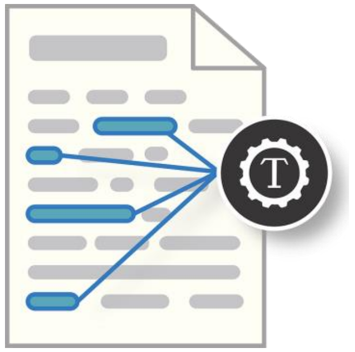
- Text data is the primary example of unstructured data which adds new challenges to the ML pipeline and the data you have seen so far (e.g., tabular data) → heavy focus on preprocessing.

Text Mining Activities

- Information Retrieval (IR): Search for relevant information in documents (does not only include text). Search for relevant documents itself.
- Natural Language Processing (NLP): Aims to “understand” text to identify and structure relevant information. Includes the generation of text (e.g., chat bots).
- Data Mining (DM): Classic approach to discover unknown patterns in structured information.

Applications

- Web-search Engines: Retrieving relevant information from web-pages in response to text-based user queries.
- Intelligent Personal Assistants: Reacting and analyzing natural language requests in order to provide relevant information/answers.
- Machine Translations: Translating texts from one language to another. Aims to maintain the natural flow of the target language instead of purely relying on literal word-by-word translations.
- Spelling Correction
- Topic Detection
- Sentiment Analysis
- Spam Detection



Challenges

Language Dependency

- One of the major issues in regards to text mining → methods/techniques/algorithms almost always have to be altered to accommodate the target language.
- Different meaning of words in different languages: “Brief” → German? English?

Domain Dependency

- Texts can have different style, meaning, and purpose in different settings (e.g., legal texts vs. classic literature vs. holiday post cards)
- Often domain knowledge and context is needed to properly understand texts.

Ambiguity

- Quantifiers: “I didn’t buy a house.”
- Word Sense: “I went to the bank.”
- Idioms: “To get cold feet.”

Performance Aims

Effectiveness

- Due to mentioned challenges, Text Mining is rarely free of errors.
- Circumventing those challenges in order to maximize effectiveness is a primary goal of any text mining related approach.

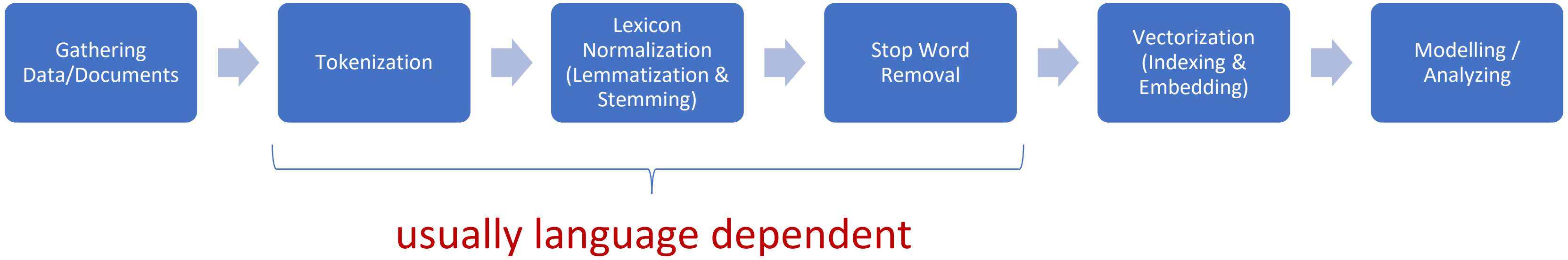
Efficiency

- Text mining often has to deal with huge amounts of data.
- However, especially in practical applications, it is vital to manage available resources carefully and to optimize run-time and storage needs (e.g., response times for intelligent assistants).

Robustness

- Text mining often needs to handle texts with unknown properties.
- Techniques and methods related to text mining should be effective across different domains, genres, and topics.

Classic Text Mining Pipeline



Pipeline Approach

- Classic and standard approach to tackle text mining problems.
- Sequentially apply a set of text analysis algorithms to the input texts.

Alternative Approaches

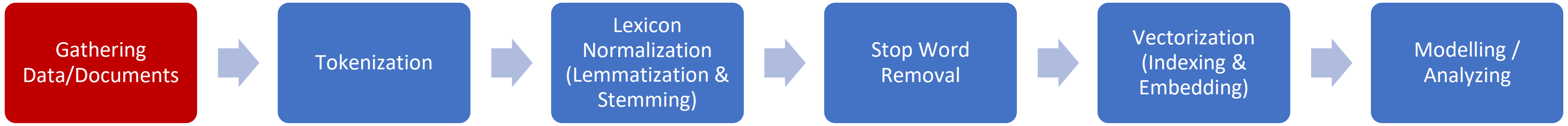
- Joint Models: Realize multiple analysis steps at the same time.
- Neural Networks: Often work best with plain input texts.

Content

- **Introduction to Text Mining**
  - Terms
  - Applications
  - Challenges
  - Performance Aims
- **Gathering Text Data**
- **Preprocessing Text**
  - Tokenization
  - Normalization
  - Stop Word Removal
- **Vectorizing Text (Feature Generation)**
  - Frequency-based Approaches
  - Prediction-based Approaches

Text Mining Terminology

- First step of the text mining pipeline is to gather and parse text from a document.



- Where to get those documents?
  - Freely available resources (e.g., Kaggle)
  - Scraping textual information (e.g., from the web)
- In terms of text mining, what exactly is a document?

Example: Tweets by Donald Trump

- Imagine you have a collection of tweets by Donald Trump for which you want to perform a sentiment analysis.
- In this scenario each tweet would be a single **document**.
- The entirety of all those documents (tweets) is called **corpus**.
- Documents are usually comprised of multiple **terms/tokens**.
- The smallest unit of a token is a **character**.
- Tokens can include alpha-numeric characters as well as special characters (e.g., punctuation, @, #)
- Tokens are usually separated by white spaces (language dependent)

Documents

Can come in various types:
- HTML, LaTeX, PowerPoint, …
- Tweets, E-Mails, Protocols, …

Need to have a defined unit:
- A file?
- A tweet?
- An e-mail with 5 attachments?
- A group of files?

Can impose complications in regard to different languages:
- A single document containing terms of several languages
- Multiple languages for different documents (e.g., German e-mail with English attachment)

Web Documents

Web-Crawling / Scraping:

- Process of systematically indexing web documents
  - Fetching the web page, parsing it, extracting data from it
- Can be used to retrieve textual information from a web-page and linked sub-pages (e.g., a corpus of tweets by Donald Trump)
- Might violate the terms of use of some web pages → could constitute copyright infringement

The web and its challenges:

- Unstructured
- Unusual and diverse documents
- Unusual and diverse users, queries, information needs
- Vast and non-static (repeatability)
- Diverse quality of information (correctness, age)

Established Python libraries:

- Beautiful Soup
- Scrapy

Content

- **Introduction to Text Mining**
  - Terms
  - Applications
  - Challenges
  - Performance Aims
- **Gathering Text Data**
- **Preprocessing Text**
  - Tokenization
  - Normalization
  - Stop Word Removal
- **Vectorizing Text (Feature Generation)**
  - Frequency-based Approaches
  - Prediction-based Approaches

Tokenization

Gathering Data/Documents → **Tokenization** → Lexicon Normalization (Lemmatization & Stemming) → Stop Word Removal → Vectorization (Indexing & Embedding) → Modelling / Analyzing

- Input: **Document**
  - "I would like to study computer science."
- Output: **Tokens**
  - I
  - would
  - like
  - to
  - study
  - computer
  - science
- A token is a sequence of characters in a document.
- Each such taken is a candidate for an index entry after further processing.
- Usually tokens are complete words. However, it is also possible to tokenize sentences, characters or *n*-grams (more on that later).

Tokenization Issues – General

- What are valid tokens?
  - ***“Finland’s capital*”** → ***Finland***? ***Finlands***? ***Finland’s***?
  - “***Hewlett-Packard***” → ***Hewlett*** and ***Packard*** as two tokens?
    - General hyphen issue: split, merge, drop, keep?
      - ***state-of-the-art***
      - ***co-education***
      - ***lower-case***, lower case, lowercase
      - ***Neu-Ulm***
  - “***San Francisco***” → one or two tokens?
    - Morpheme analysis might help to make decision

Tokenization Issues – Numbers

- Should numbers be tokenized?

- ***10/12/20***
- ***Dec. 10, 2020***
- ***55 B.C.***
- ***(+49) 123 456 7890***
- ***Your PGP key is 123a3df125cb33e***

  - Often have embedded whitespace
  - Can be useful depending on the problem (e.g., error codes, logs)

Tokenization Issues – Language

- French
  - ***L'ensemble*** → one or two tokens?
    - **L**, **L'**, **Le** ?
- German
  - ***Lebensversicherungsgesellschaft*** → usually **compound splitting** helps a lot
    - **Leben**, **Versicherung**, **Gesellschaft**
- Chinese & Japanese
  - No spaces between words → unique tokenization not always guaranteed
- Japanese
  - Multiple intermingled alphabets → Katakana, Hiragana, Kanji, Romaji
- Arabic & Hebrew
  - Words written from right to left, numbers written from left to right

Lexicon Normalization

Gathering Data/Documents → Tokenization → Lexicon Normalization (Lemmatization & Stemming) → Stop Word Removal → Vectorization (Indexing & Embedding) → Modelling / Analyzing

- Normalization is the process of transforming tokens into a standard format
- This approach can help to reduce the complexity of the vocabulary and positively affect later modelling phases
- However, too much normalization can lead to loss of information (and, therefore, reduce the performance of some analysis tasks) → amount/type of normalization is dependent on the use-case
- Tokenization and Normalization often done concurrently

Special Character Normalization

- Removing punctuation
  - ***U.S.A.*** → ***USA***
- Deleting hyphens
  - ***anti-discriminatory*** → ***antidiscriminatory***
- Deleting accents
  - ***résumé*** → ***resume***
- Changing umlauts
  - ***Tübingen*** → ***Tuebingen***

- Alternative approach to umlauts and accents:
  - normalize de-accented tokens
    - ***Tuebingen***, ***Tübingen*** → ***Tubingen***

Normalization cont.

- Case folding
  - Reducing all letters to lower case
  - Always a good idea?
    - ***Fed*** (US central bank) vs. ***fed***
    - ***CAT*** (Caterpillar Inc.) vs. ***cat***
  - Despite possible ambiguity, almost always best to lower case everything
- Date formats
  - ***1.3.20***, ***01/03/20*** → ***01.03.20***
- Synonyms (advanced)
  - ***car*** → ***automobile*** (synonym)
- Correcting spelling mistakes, alternative spellings, and homophones (advanced)
  - ***donut***, ***dougnut*** → ***doughnut***
  - ***color*** → ***colour***
  - ***I would like to eat a carat*** → ***I would like to eat a carrot***

Stemming

- Reducing terms to their word **stem** (also called **base** or **root** form)
- Stemming suggests crude affix chopping
- Language dependent
- ***automate***, ***automatic***, ***automation*** → ***automat***

for example compressed and compression are both reduced to their stem compress

→

for exampl compress and compress ar both reduc to their stem compress

- Word meaning/spelling can become incorrect
  - ***example*** → ***exampl***

Porter Stemmer

- Also called Porter's algorithm
- Most common stemmer for English language
- Uses conventions (rules) & 5 phases of reduction

- Typical rules
  - ***sses*** → ***ss***
  - ***ies*** → i
  - ***ational*** → ***ate***
  - ***tional*** → ***tion***

- Rules are sensitive to word length
  - ***replacement*** → ***replac***
  - ***cement*** → ***cement***

Stemming cont.

- Other stemmers
  - **Lovins stemmer**
    - Single-pass, longest suffix removal, about 250 rules
  - Full morphological analysis → very high effort, modest benefits

- Is stemming worth the effort?
  - (again) very language dependent
  - English: rather mixed results
    - ***operative*** (dentristy) → ***oper***
    - ***operational*** (research) → ***oper***
    - ***operating*** (systems) → ***oper***
  - Often better suited for other languages (e.g., Spanish, German, Finish)

Lemmatization

- Finding the non-inflected form (i.e., **lemma**) of a term
- Lemmatization suggests proper reduction to the dictionary form of a token instead of simply chopping affixes
  - ***am***, ***are***, ***is*** → ***be***
  - ***car***, ***cars***, ***car's*** → ***car***

the boy's cars are different colors → ***the boy car be different color***

- Lemmatization retains correct spelling and can mostly convey the original meaning of a text

Lemmatization cont.

- To perform proper lemmatization each token must be labeled as a corresponding part of speech (e.g., noun, verb, adjective, etc.)
  - “***He rose to the occasion***”
    - ***rose*** (verb) → ***rise***
    - ***rose*** (noun) → ***rose***
- Additionally, it is possible to indicate various grammatical categories
  - Tense (e.g., past, present, future)
    - ***was***, ***were***, ***am***, ***is*** → ***be***
  - Number (singular, plural)
    - ***window***, ***windows*** → ***window***
  - Case (e.g., nominative, genitive)
    - ***Andy’s***, ***Andy*** → ***Andy***
- Process is called POS-Tagging which in itself is a text analysis process
- If POS-Tagging is not possible, lemmatization applies the rules of a default tag
  - **English**: reasonable results when tagging all tokens as **verb** by default

Stop Words

Gathering Data/Documents → Tokenization → Lexicon Normalization (Lemmatization & Stemming) → **Stop Word Removal** → Vectorization (Indexing & Embedding) → Modelling / Analyzing

- A stop word is a token that is very frequent in texts, but does not contribute much value (in terms of understanding the meaning of the text)
- Usually stop words are very common terms
  - ***the***, ***a***, ***and***, ***be***
- Stop words can be excluded to reduce vocabulary and problem size
  - Little semantic impact
  - There can be quite a few of them in a single document

Stop Words cont.

- Apart from shown common terms, stop words are application specific
  - There is no universal list of stop words
- However, in some applications, stop word removal can be disadvantageous
  - Stop words might be needed for:
    - Phrases: “King of Denmark”
    - Relational queries: “flights to London”
    - Song titles, citations, etc.: “Let it be”, “To be or not to be”

Content

- **Introduction to Text Mining**
  - Terms
  - Applications
  - Challenges
  - Performance Aims
- **Gathering Text Data**
- **Preprocessing Text**
  - Tokenization
  - Normalization
  - Stop Word Removal
- **Vectorizing Text (Feature Generation)**
  - Frequency-based Approaches
  - Prediction-based Approaches

Feature Generation

Gathering Data/Documents → Tokenization → Lexicon Normalization (Lemmatization & Stemming) → Stop Word Removal → Vectorization (Indexing & Embedding) → Modelling / Analyzing

- Result of preprocessing phase: list of cleaned tokens for each document
  - Input: “***European Countries are sadly getting clobbered by the China Virus. The Fake News does not like reporting this!***”
  - Output: [‘***europe***’, ‘***country***’, ‘***sad***’, ‘***get***’, ‘***clobber***’, ‘***china***’, ‘***virus***’, ‘***fake***’, ‘***news***’, ‘***like***’, ‘***report***’]

- Issue: categorical (textual) values are not suited for machine learning and analysis tasks
- Categorical values need to be transformed into numerical values
  - Depending on the context, different terms for this **vectorization** process
    - Query search: **indexing**
    - Machine learning: **feature generation**
    - Neural Networks: **embedding**

Vectorization: One Hot Encoding (OHE)

- Basic and intuitive approach
- Boolean vector representation
  - **1** → term is present in a document
  - **0** → term is not present in a document
- Example:

Doc1: "***Mike likes cats***."

Doc2: "***Sandy likes dogs***. ***Sandy and Mike like each other***."

	mike	like	cat	sandy	dog	each	other
Doc1	1	1	1	0	0	0	0
Doc2	1	1	0	1	1	1	1

- Issues:
  - Vectors can become very sparse (many **0** entries)
    - Curse of dimensionality
    - Increased memory demands
  - Does not consider semantic relation between terms
  - Does not consider frequency of terms in documents

Vectorization: Frequency Vectors

- **OHE** only provides binary information (term present, not present)
- Extension to this approach: taking **frequency** of terms into account
- **Bag-of-Words (BOW)**
  - Still no semantic relation between words (word order does not matter) → words are thrown into a fictional bag
  - However, frequency gives possible insight into more important terms (the higher the frequency, the more important)

Doc1: "***Mike likes cats***."

Doc2: "***Sandy likes dogs***. ***Sandy and Mike like each other***."

	mike	like	cat	sandy	dog	each	other
Doc1	1	1	1	0	0	0	0
Doc2	1	2	0	2	1	1	1

Bag-of-Words cont.

- BOW represents a feature vector without order
- Possible extension to this approach: indexing ***n*-grams** instead of single terms
  - multiple variations: **bi-grams**, **tri-grams**, etc.
  - allows to keep spatial information between terms

Doc1: "***Mike likes cats***."

Doc2: "***Sandy likes dogs***. ***Sandy and Mike like each other***."

bi-grams	mike like	like cat	sandy like	like dog	dog sandy	sandy mike	like each	each other
Doc1	1	1	0	0	0	0	0	0
Doc2	1	0	1	1	1	1	1	1

- retains spatial relationship between terms (e.g., ***like*** always follows a name)
- Does not counteract sparseness

TF-IDF

- Yet another issue: Frequency-based approaches do not take into account the importance of a word
  - Rare terms are more informative than frequent terms (recall frequent stop words (e.g., ***a***, ***the***, ***this***) without adding to the meaning of a document)
  - To measure importance of a term it is necessary to normalize term frequencies (of a single document) with respect to the entire corpus of documents
    - Terms that are rare across all documents are more important
- **Term Frequency – Inverse Document Frequency** (**tf-idf**) is a measure to identify the importance/relevance of a single term across a collection of documents
  - Especially important for information retrieval tasks and query searches

Part I: Term Frequency (TF)

- Term frequency *tf(t, d)* is the number of times a term *t* occurs in a document *d*
  - Also called raw term frequency (similar to bag-of-words)
- Other definitions possible: Boolean term frequency (similar to one-hot-encoding)
  - $tf(t,d) = \begin{cases} 1 & \text{if } t \epsilon d, \\ 0 & \text{otherwise} \end{cases}$
- Issue: relevance/importance of term is not linear to raw term frequency
  - a document containing the term ***dog*** 20 times is not 20 times more relevant than a document containing the term ***dog*** just once
- Solution: log-frequency weighting
  - $weight(t,d) = \begin{cases} 1 + \log(tf(t,d)), & \text{if } tf(t,d) > 0 \\ 0, & \text{otherwise} \end{cases}$

Part II: Inverse Document Frequency (IDF)

- Let *N* be the total number of documents
- Document frequency *df(t)* is the number of documents that contain a certain term *t*
- The multiplicative inverse of *df(t)* is called **inverse document frequency** *idf(t)*:
  - $idf(t) = log\frac{N}{df(t)}$
- *idf(t)* is a measurement that weighs the rareness of a term within a collection of documents

IDF Example

Doc1: "***Mike likes cats***."

Doc2: "***Sandy likes dogs***. ***Sandy and Mike like each other***."

Doc3: ***"Mike and Sandy like each others dog."***

- $N = 3$

	mike	like	cat	sandy	dog	each	other
df(t)	3	3	1	2	2	2	2
idf(t)	0	0	0.48	0.18	0.18	0.18	0.18

- The term ***cat*** is very rare across all documents and receives the highest weight *idf(t)*
- The terms ***mike*** and ***like*** contribute no informative value (all 3 documents make a statement about something Mike likes)

Part III: TF-IDF

- The product of *tf(t, d)* and *idf(t)* is called Term Frequency – Inverse Document Frequency *tf-idf(t, d)*:

  - $tf.idf(t,d) = tf(t,d) \cdot log \frac{N}{df(t)} = tf(t,d) \cdot idf(t)$

- One of the most popular weighting schemes for information retrieval
- Value increases with number of term occurrences within a single document
- Value increases with the rarity of the term in the entire corpus

TF-IDF Example

Doc1: “***Mike likes cats***.”

Doc2: “***Sandy likes dogs***. ***Sandy and Mike like each other***.”

Doc3: ***“Mike and Sandy like each others dog.”***

- $N = 3$

	mike	like	cat	sandy	dog	each	other
idf(t)	0	0	0.48	0.18	0.18	0.18	0.18
tf(t, Doc2)	1	2	0	2	1	1	1
tf-idf(t, Doc2)	0	0	0	0.36	0.18	0.18	0.18

- ***Sandy*** is the term that holds the most information in document 2
  - terms with high **tf-idf** values can be interpreted as salient keywords to a document

Content

Motivation

- Frequency-based vectorization methods are a good fit for a variety of analysis tasks (e.g., query search, text classification)
- However, despite the improvements made – going from one-hot-encoded vectors to tf-idf representations – frequency based vectorization still suffers from a variety of issues:
  - Sparse vector representations: dealing with vectors that only have few non-zero values
    - Especially hurtful in applications where artificial neural networks are utilized → larger input vectors lead to larger amount of weights to be trained → more computational resources required
  - Lack of meaningful relation between words: apart from little spatial relations, it is not possible to retrieve semantic relations between words (e.g., identifying words that are similar to each other)
    - Many modern NLP applications (again mostly driven by ANNs) require knowledge about the relation between words to perform properly (e.g. machine translations)
- Solution: **(Dense) Embeddings**
  - Translating large sparse vectors into a lower-dimensional space, preserving semantic relationships

Embeddings Intuition

- To understand the idea behind embeddings, let us consider an easy example:
  - We have 5 documents each consisting of a single word only:

D1: male	D2: female	D3: blue	D4: yellow	D5: red

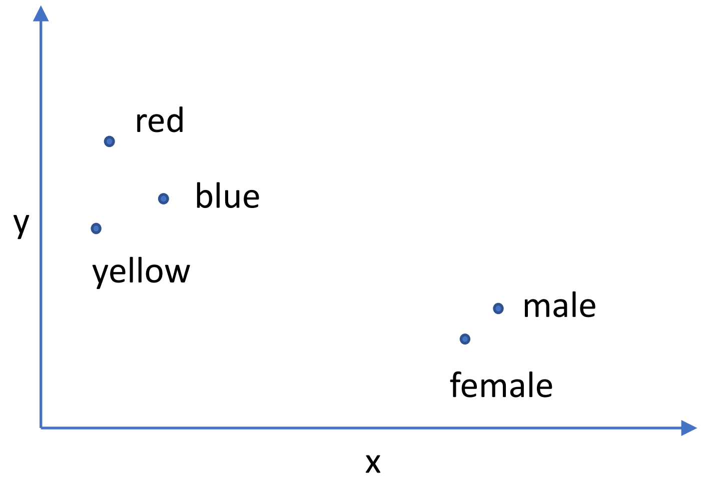
  - A (sparse) one-hot-encoding for those documents

	male	female	blue	yellow	red
D1	1	0	0	0	0
D2	0	1	0	0	0
D3	0	0	1	0	0
D4	0	0	0	1	0
D5	0	0	0	0	1

  - Each document (word) is represented by a 5-dimensional vector
  - Position in the vector space does not give any inside into the semantic relation of the documents (e.g., colors, sex)

Embeddings Intuition cont.

- The general goal is to find a representation that puts those terms with semantic relationships closer to each other in the vector space
- The following image shows how the same 5 documents are embedded into a 2-dimensional vector space



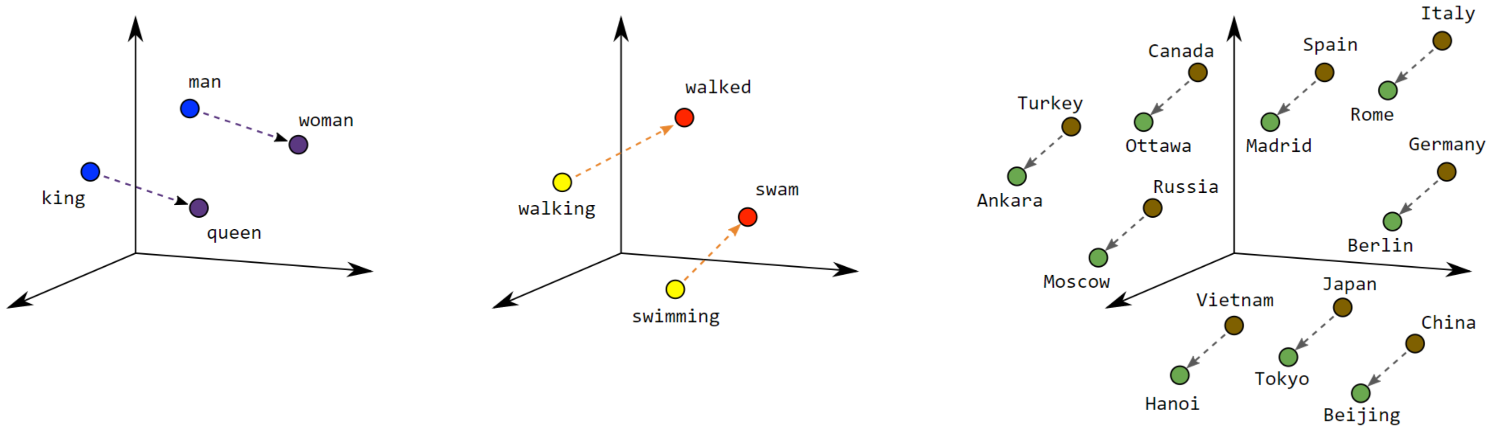
- Instead of utilizing a sparse 5-dimensional representation, the words can be represented in a dense 2-dimensional embedding (x, y coordinates) → **dimensionality reduction**
- Transferring that idea to larger problems → for example, imagine a corpus which contains 10k different words → 10k-dimensional representations could be reduced to 50 dimensions (features) while retaining semantic information among terms

Embeddings cont.

- What exactly are those features (x and y in our example)?
  - No inherent interpretation possible
  - Usually a **neural network** will come up with the important relations between terms (feature selection)
  - → word embeddings can be learned by training a neural network
  - → Embedding size (number of features) is arbitrary
    - Should be large enough to encode meaningful semantic relations
    - Should be small enough so that training times are kept at a minimum
  - Arguably, one the most popular techniques for word embeddings in NLP applications is Word2Vec, introduced in 2013 by Mikolov, Tomas et. al. (Google)

Embedding Examples

- Image shows examples of different semantic relationships from real embeddings
- Semantics are encoded by position in a multi-dimensional vector space (distance and direction)



Male-Female Verb Tense Country-Capital

Word2Vec

- The main idea of Word2Vec is to map semantically similar terms to geometrically close embedding vectors
- It relies on the distributional hypothesis, stating that terms which have the same neighboring words tend to be semantically similar
- E.g., the terms ***blue***, ***red***, and ***yellow*** are semantically related, because they often appear close to the word ***color***
- Word2Vec can exploit such contextual information by training a neural network that can distinguish between randomly grouped words from frequently co-occurring words
- The technique comes in two flavors:
  - Continues Bag-of-Words (CBOW)
    - Predicting a word given its context
  - Skip-gram
    - Predicting a context given a word

Continuous Bag-of-Words (CBOW)

- Consider a sliding window over a text taken from the course description for Intelligent Systems:
- ***"… amount of information available digitally …"***
- The central word ***information*** is the **target word**
- The two terms preceding and following it are the **context words**
- CBOW aims to predict the target word given the context words
- On the right you can see the schematic representation of the CBOW network used to learn this representation
- Input Layer: Context words $x_{1k} \dots x_{Ck}$ as one-hot-encoded vectors with length V (in our case 5 → vocabulary size); C equals to the amount of context words (in our case 4)
- Hidden Layer: (also called Projection Layer) with N dimensions (i.e., embedding size)
- Output layer: contains (softmax activated) possibility distribution for each term of the vocabulary
- How to achieve embedding? → Multiplying one-hot-encoded term with trained weight matrix W (embedding matrix)

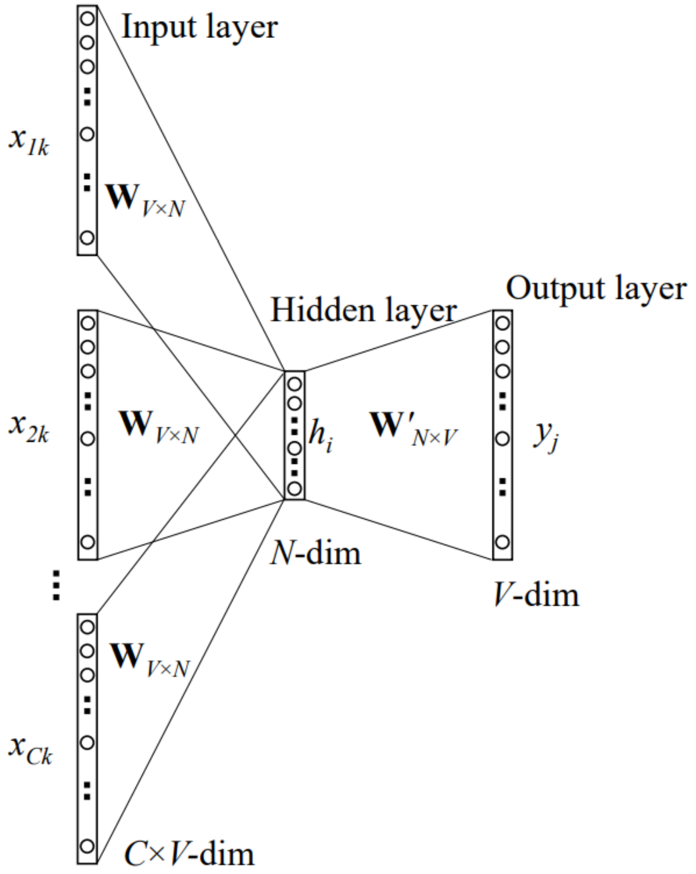


Image Source: Rong, 2014 – word2vec Parameter Learning Explained

Skip-gram

- In some sense, turns CBOW architecture around
- Skip-gram tries to predict context words given a target word
- Input: one-hot-encoded vector of size V → ([0, 0, 1, 0, 0]) (***information***)
- Output: Instead of one probability distribution, now $C = 4$ distributions of predicted context words (in our case, hopefully: ***amount***, ***of***, ***available***, ***digitally***)
- Training objective is to minimize the total prediction error for all context words in the output layer
- Embedding is still achieved by multiplying one-hot-encoded target word with trained W

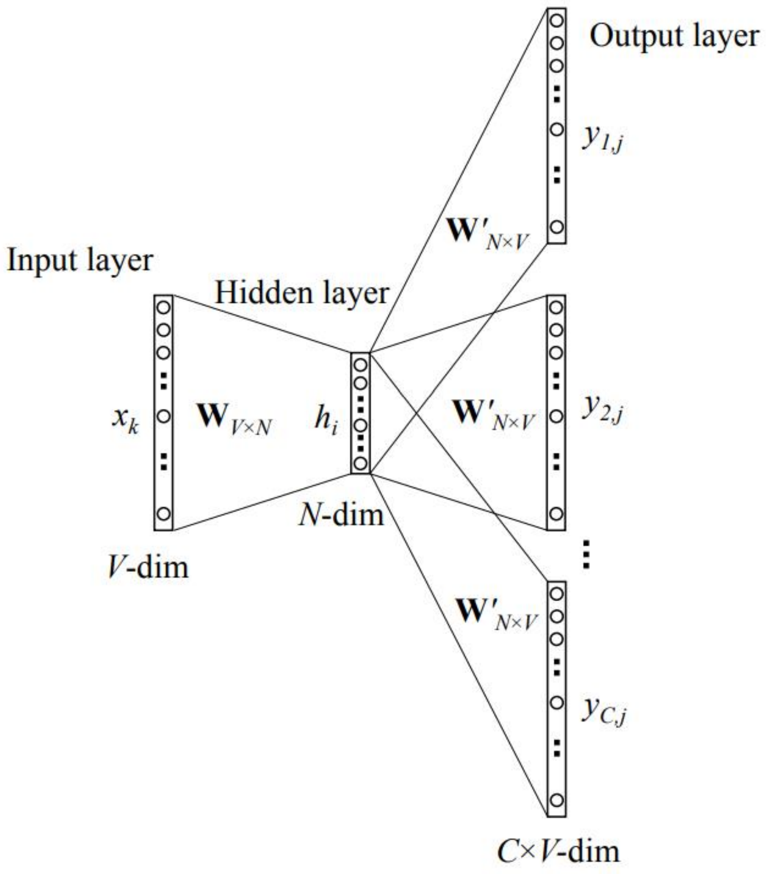


Image Source: Rong, 2014 – word2vec Parameter Learning Explained

Embeddings

- According to Tomas Mikolov, both approaches valid → application dependent
  - CBOW trains faster; CBOW comes up with better embeddings for frequent/common terms
  - Skip-gram works particularly well for smaller datasets; Skip-gram also represents rare words better
- Luckily, for standard applications, you don't have to train an embedding network yourself → pre-trained embeddings available
- Apart from Word2Vec, many other predictive approaches to word embeddings possible:
  - Facebook's **fastText** (especially interesting for non-English corpuses)
  - **GloVe** (Global Vectors) → embeddings are learned using matrix factorization techniques