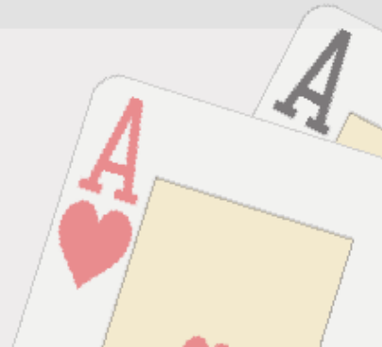
AI Games course

Certificate 2, session 3

Predictive Text game.

AIgaming.com



Recap of last session

- Language Identification task

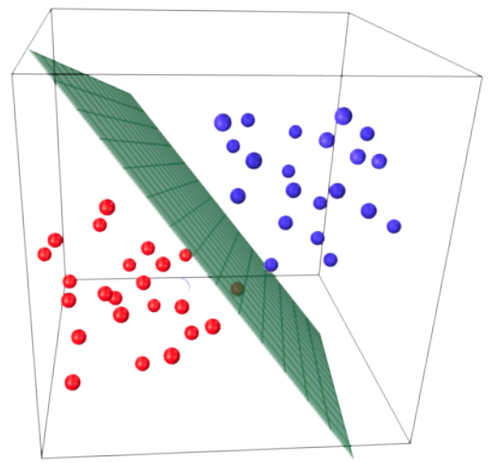
“gelukkige verjaardag” → *Dutch*

“Gratulerer med dagen” → *Norwegian*

- using ngrams as features

‘how to make an AI gaming bot’ → *‘how’, ‘ow ‘, ‘w t’, ‘ to’, ‘to ’, ‘o m’, ‘ ma’, ‘mak’, ‘ake’, ‘ke ’, ‘e a’, ‘ an’, ‘an ’, ’n A’, ‘ AI’, ‘AI ’* etc.

- and Logistic Regression as the classifier



Predictive Text game

smartypants-defbot vs housebot-practise
Id:71892 Started:15:03 25/02/2018 for 1

smartypants-defbot	housebot-practise
1. in must have been a bitter thing for him of have it lay down his great power of last.	1. of must have been a bitter thing for him in have to lay down his great power is last.
2. Many a modern highway crosses deep valleys over great viaducts, and without viaducts our railways could scarcely have passed over the mighty mountain ranges.	2. Many a modern highway crosses deep valleys over great viaducts, and without viaducts our railways could scarcely have passed over the mighty mountain ranges.
3. Now the muscles that move our bones are only one it three kinds of muscle of our bodies.	3. Now the muscles that move our bones are only one of three kinds to muscle __ our bodies.
4. More than this, after training, persons such to athletes, acrobats of dancers perform with ease the most complicated and graceful	4. More than this, after training, persons such __ athletes, acrobats __ dancers perform with ease the most complicated and graceful

smartypants-defbot stopped game, therefore housebot-practise won

Given a text fragment in English with some of the words omitted, fill in the gaps using language modelling techniques.

Omitted are two-letter words: *of*, *to*, *in*, *it*, *if*, *is*, *by*, *he*, *on*, *we*, *as*, *be*, *up*, *at* etc.

There are finitely many of them (this list would probably cover 95%)
➡ multi-class classification!

Predictive Text game

OBJECTIVE

This game has two different variations.

- You are given a list of sentences and all of the two letter words have been removed.

1. __ the board __ __ __ one inch thick, __ pushes the log forward with the lever until __ __ one inch beyond the line __ the saw, which __ then set __ work.

2. Francis was making himself supreme; his word was law.

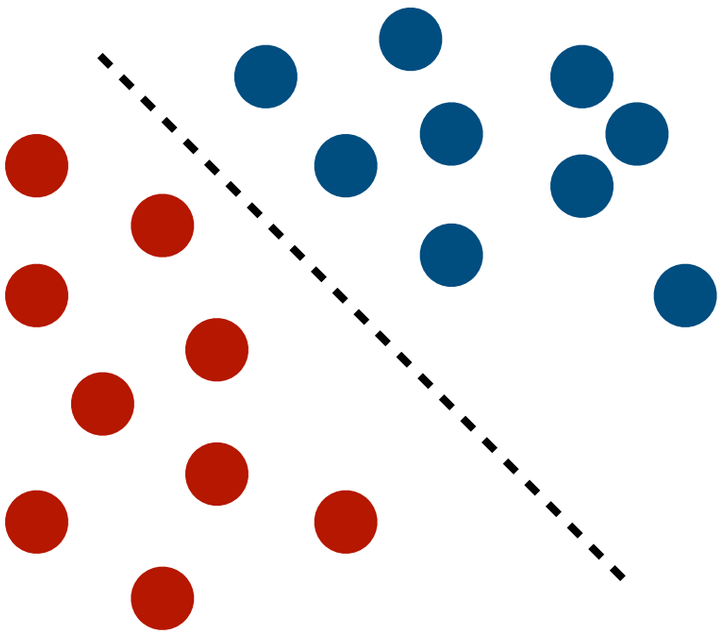
3. This much __ know __ the relics __ have found; but where that ancient race came from __ what became __ __ __ __ not know.

4. __ __ built __ the cantilever principle.

5. This dissolves out the perishable green material found __ plant tissue and leaves behind nothing but pulp, which __ nearly pure cellulose.

Discriminative vs Generative models

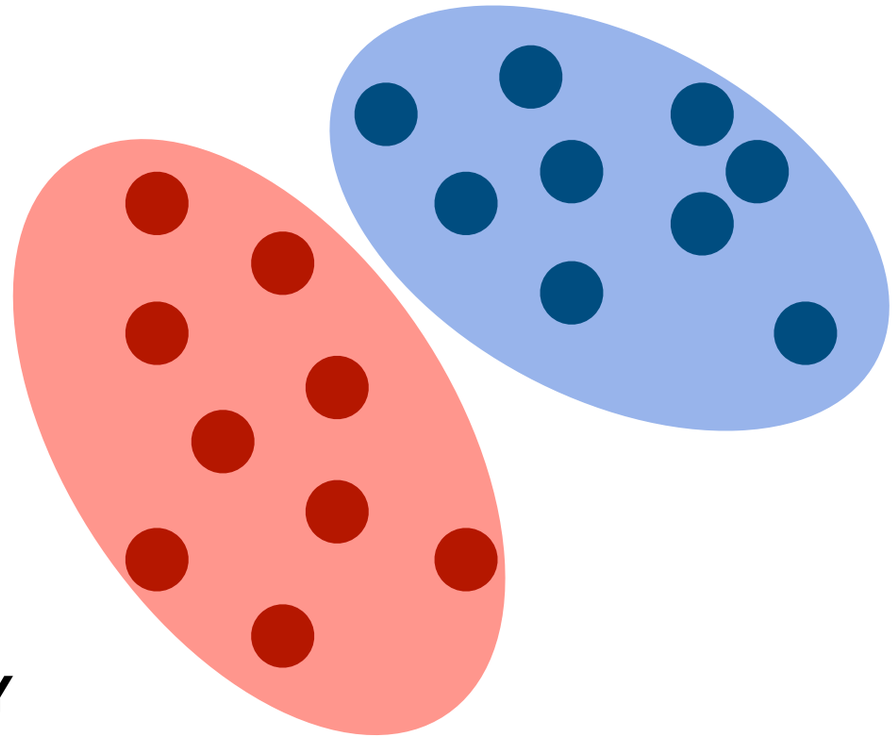
Discriminative



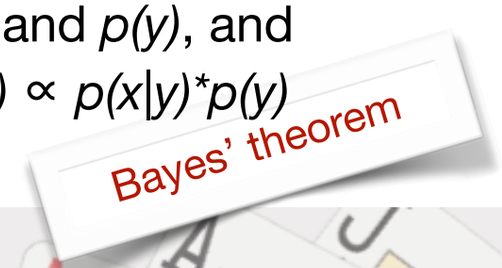
learn $p(y|x)$

Given data **X** and labels **Y**

Generative



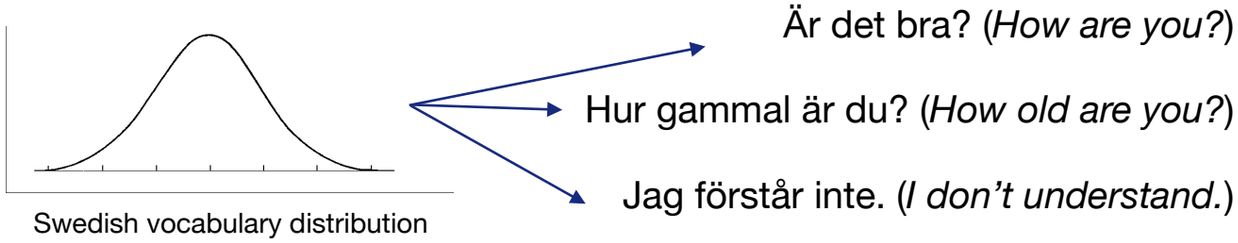
learn $p(x|y)$ and $p(y)$, and derive $p(y|x) \propto p(x|y)*p(y)$



i.i.d. assumption

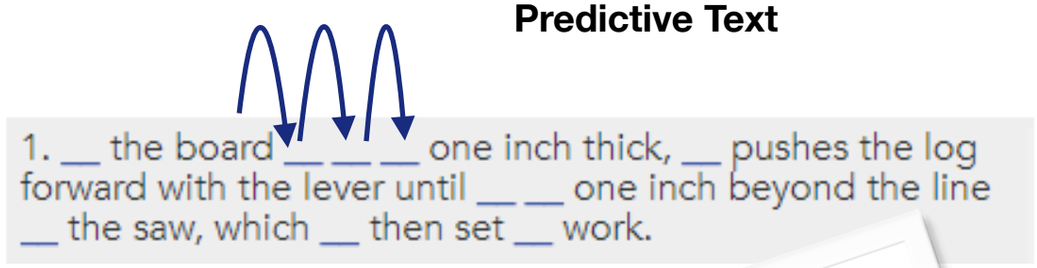
- In many classification scenarios, and in many ML algorithms, one makes the i.i.d. assumption: data points (instances) are independently and identically distributed https://en.wikipedia.org/wiki/Independent_and_identically_distributed_random_variables
  - instances are independent of each other, and
  - they have the same underlying distribution.
- But this is not always the case!

Language Identification



VS

Predictive Text



Types of classifiers

	Descriminative	Generative
Simple predictions	Logistic regression ✔	*Naive Bayes*
Structured predictions	*CRFs*	**Markov Models**

Sequence modelling

- Given a sentence:

  `start` *I want to go to London* `stop`

- the probability of this sentence is:

  $p(\texttt{start}, w_1, w_2, w_3, w_4, w_5, w_6, \texttt{stop}) = \prod_{1 \le i \le n+1} \mathit{freq}(w_i | w_1, \ldots, w_{i-1})$

- or, to reduce computation, we can introduce a *history window* ***k:***

  $p(\texttt{start}, w_1, w_2, w_3, w_4, w_5, w_6, \texttt{stop}) = \prod_{1 \le i \le n+1} \mathit{freq}(w_i | w_{i-k}, \ldots, w_{i-1})$

Ex: window of size *2*

p(`start`, I, want, to, go, to, London, `stop`) = *freq*(I | `start`) * *freq*(want | `start`,I) * *freq*(to | I,want) * *freq*(go | want,to) * *freq*(to | to, go) * *freq*(London | go,to) * *freq*(`stop` | to,London)

Sequence modelling

- **Q:** What if a part of input is missing:

  `start` *I want to go* __ *London* `stop`

- **A:** in place of an unknown word try putting *all possible options*, and choose the one with the highest overall probability

generally this means any word from the vocabulary

but in case of Predictive Text it is w ∈ {*of*, *to*, *in*, *it*, *if*, *is*, *by*, *he*, *on*, *we*, *as*, *be*, *up*, *at*}

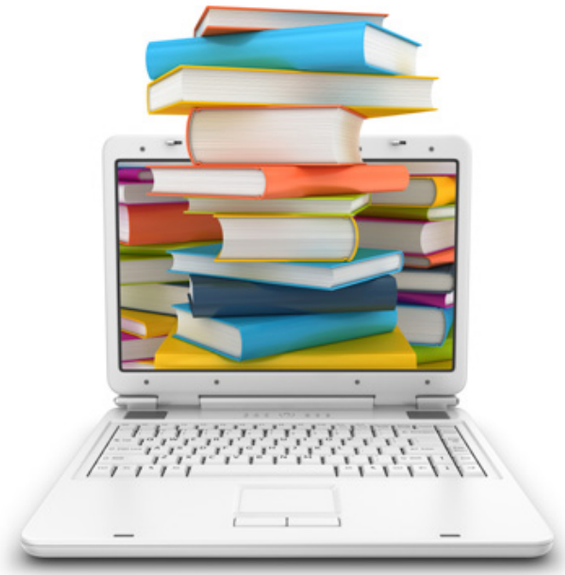
Ex:

p(`start`, I, want, to, go, **w**, London, `stop`) = ***argmax*** $_{\mathbf{w} \in \{of, to, in, \ldots\}}$ $\prod_{1 \le i \le n+1}$ *freq*($w_i \mid w_{i-2}, w_{i-1}$)

Sequence frequencies

Q: But how do we know the frequencies?

A: External corpus!



- There are numerous English corpora: https://en.wikipedia.org/wiki/List_of_text_corpora.

- In principle, the bigger the corpus, the better: better modelling + fewer chances that a sequence was seen.

English corpus

- We have taken a fragment of NOW corpus: www.corpusdata.org/now_corpus.asp of around 2 mio words
- From github download *corpus.txt*

```
import re

# https://www.corpusdata.org/now_corpus.asp -> samples -> text
file = open('corpus.txt', 'r')
input = file.read()

corpus = re.split(" ", input)
corpus_size = len(corpus)
print(corpus_size) # 1959580
```


Estimating relative frequencies

- *freq*($w_i|w_{1,...,}w_{i-1}$) = *count*($w_{1,...,}w_{i-1},w_i$) / *count*($w_{1,...,}w_{i-1}$)
- *freq*(w) = *count*(w) / total *# words*

Smoothing:

What if some of the subsequences are not seen in the data?

$p(\texttt{start},w_1,...,w_n,\texttt{stop}) = \prod_{1 \le i \le n+1} freq(w_i|w_1,...,w_{i-1})$

If for some *i* *freq*($w_i|w_{1,...,}w_{i-1}$) is *0*, the whole product becomes *0*.

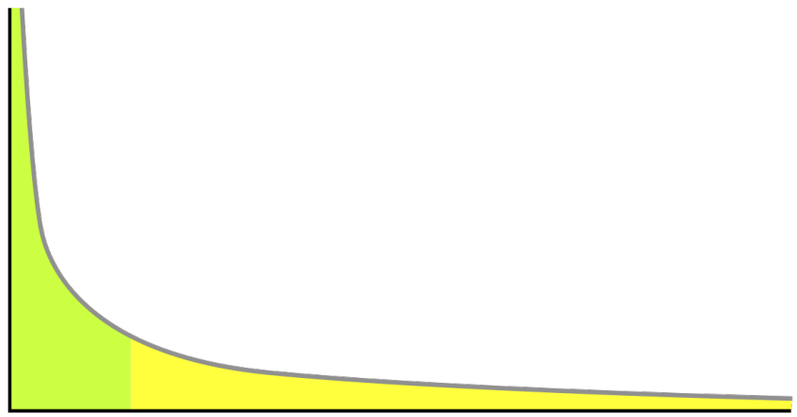
This is impractical ➡️ we can *smooth* frequencies by increasing all counts by *1*.

Speeding up the process

- Computing counts and frequencies on the fly is time-consuming.

- Hence, **to save time**, we need to *precompute* counts over the corpus, int he same manner as frequencies are precomputed in the Google Ngram project: http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

- Moreover, **to save space**, we are only going to store counts for sequences that appear more than once (we cut *the "long tail"*).



"long tail"

Precomputing the counts

- We will only use precomputed counts during the game.
- Counts can be downloaded from github in the form of a dictionary: *corpus-counts.txt*

```
# precompute counts
sequence_counts = dict()
for i in range(corpus_size - history_window+1):
    sequence = tuple(corpus[i:i+history_window+1])
    sequence_counts[sequence] = sequence_counts.setdefault(sequence, 0) + 1
print(len(sequence_counts))

# filter counts
top_frequencies = dict()
threshold = 1
for sequence,frequency in sequence_counts.items():
    if frequency > threshold:
        top_frequencies[sequence] = frequency

file1 = open('corpus-counts.txt','w')
file1.write(str(top_frequencies))
file1.close()
```

Computing probabilities

```
corpus_size = 1959580

history_window = 2
target_words = set(("of", "to", "in", "it", "if", "is", "by", "he", "on", "we", "as", "be", "up", "at"))

file2 = open('corpus-counts.txt','r')
dictionary = eval(file2.read())
print(len(dictionary))


def getSequenceProbability(sequence, corpus_size, history_window, dictionary):
    if type(sequence) is not list:
        sequence = re.split(" ", sequence)

    product = 1
    for i in range(len(sequence) - history_window):
        subsequence = tuple(sequence[i:i+history_window+1])
        if subsequence in dictionary:
            count = dictionary[subsequence]
        else:
            count = 1
        product *= count/corpus_size
    return product
```

```
>>> history_window = 2

>>> sequence = "I want to go to London"
>>> sequence_probability = getSequenceProbability(sequence, corpus, corpus_size, history_window)
>>> print(sequence_probability)

2.209791888110178e-16
```


Predicting sequences

```
def generatePotentialSequences(target_words, sequence, potential_sequences):
    if "__" in sequence:
        index = sequence.index(“__”) # in the game, the trigger for a missed word is different!

        updated_potential_sequencies = []
        for potential_sequence in potential_sequences:
            for target_word in target_words:
                potential_sequence[index] = target_word
                updated_potential_sequencies.append(deepcopy(potential_sequence))
        sequence[index] = "filled"
        potential_sequences = generatePotentialSequences(target_words, sequence, updated_potential_sequencies)
    return potential_sequences


def predictWords(target_words, sequence, corpus_size, history_window, dictionary):
    sequence = re.split(" ", sequence)
    if "__" in sequence:
        potential_sequences = generatePotentialSequences(target_words, sequence, [sequence])

        current_highest_probability = 0
        current_best_sequence = []
        for potential_sequence in potential_sequences:
            probability = getSequenceProbability(potential_sequence, corpus_size, history_window, dictionary)
            if probability > current_highest_probability:
                current_highest_probability = probability
                current_best_sequence = potential_sequence
        return current_best_sequence
    else:
        return sequence
```

Predicting sequences

```
def generatePotentialSequences(target_words, sequence, potential_sequences):
    if "__" in sequence:
        index = sequence.index("__") # in the game, the trigger for a missed word is different!

        updated_potential_sequencies = []
        for potential_sequence in potential_sequences:
            for target_word in target_words:
                potential_sequence[index] = target_word
                updated_potential_sequencies.append(deepcopy(potential_sequence))
        sequence[index] = "filled"
        potential_sequences = generatePotentialSequences(target_words, sequence, updated_potential_sequencies)
    return potential_sequences


def predictWords(target_words, sequence, corpus_size, history_window, dictionary):
    sequence = re.split(" ", sequence)
    if "__" in sequence:
        potential_sequences = generatePotentialSequences(target_words, sequence, [sequence])

        current_highest_probability = 0
        current_best_sequence = []
        for potential_sequence in potential_sequences:
            probability = getSequenceProbability(potential_sequence, corpus_size, history_window, dictionary)
            if probability > current_highest_probability:
                current_highest_probability = probability
                current_best_sequence = potential_sequence
        return current_best_sequence
    else:
        return sequence
```

```
>>> sequence = "I want to __ __ London"
>>> full_sequence = predictWords(target_words, sequence, corpus_size, history_window, dictionary)
>>> print(full_sequence)
I want to be in London
```