Lecture 3

Designing Features For Structured Inputs

Part 1: Feature-based representations

Machine Learning for Structured Data

Vlad Niculae · LTL, UvA · `https://vene.ro/mlsd`

Designing Features For Structured Inputs

1. **Feature-based representations**
2. Sequences
3. Graphs
4. Trees
5. Grids

Features

Last time, we saw that for any ML model we must **encode** the inputs x into some sort of numeric vector.

$$\boldsymbol{h}(x) = [h_1(x), \ldots, h_d(x)] \in \mathbb{R}^d$$

Example: x is a penguin ($\mathcal{X}$ is a set of penguins. Computers don't know how to process penguins unless we're explicit.)

$h_1(x)$ is its bill length (in mm) $h_2(x)$ is its bill width (in mm)

In this case (and many simple ML cases), features are fixed, direct *measurements*.
We just have a dataset, we can't go mess with the penguins directly :(

But other times we have a rich x with plenty of extra information.

Representing structured objects

How to manually design $\boldsymbol{h}(x)$ if x is

- a text document
- an image
- a chunk of DNA
- a molecule
- a conversation tree on Reddit?

Lecture 3

Designing Features For Structured Inputs

Part 2: Sequences

Machine Learning for Structured Data
Vlad Niculae · LTL, UvA · `https://vene.ro/mlsd`

Designing Features For Structured Inputs

1. Feature-based representations
2. **Sequences**
3. Graphs
4. Trees
5. Grids

Encoding text documents

	text
x_1	"this book is good!"
x_2	"fairly long book"
x_3	"the book isn't good."
	…

To represent text in a computer-friendly way, some things must happen:

Encoding text documents

	text	tokenized
x_1	"this book is good!"	[`this`, `book`, `is`, `good`, `!`]
x_2	"fairly long book"	[`fairly`, `long`, `book`]
x_3	"the book isn't good."	[`the`, `book`, `is`, `n't`, `good`, `.`]
	...	

To represent text in a computer-friendly way, some things must happen:

1. **Tokenize:** split a string into a sequence of "tokens".
   (Roughly, think "words": but words are hard to define.)
   (Not easy! In some languages this is much harder than others!)

Encoding text documents

	text	tokenized
x_1	"this book is good!"	[this, book, is, good, !]
x_2	"fairly long book"	[fairly, long, book]
x_3	"the book isn't good."	[the, book, is, n't, good, .]
	...	

To represent text in a computer-friendly way, some things must happen:

1. **Tokenize:** split a string into a sequence of "tokens".
   (Roughly, think "words": but words are hard to define.)
   (Not easy! In some languages this is much harder than others!)
2. **Build vocabulary:** the set built from all tokens that appear in the training data.

!	.	book	fairly	...	the	this
0	1	2	3		8	9

Encoding text documents

	text	tokenized	encoded
x_1	"this book is good!"	[`this`, `book`, `is`, `good`, `!`]	[9, 2, 5, 4, 0]
x_2	"fairly long book"	[`fairly`, `long`, `book`]	[3, 6, 2]
x_3	"the book isn't good."	[`the`, `book`, `is`, `n't`, `good`, `.`]	[8, 2, 5, 7, 4, 1]
	…		

To represent text in a computer-friendly way, some things must happen:

1. **Tokenize:** split a string into a sequence of "tokens".
   (Roughly, think "words": but words are hard to define.)
   (Not easy! In some languages this is much harder than others!)
2. **Build vocabulary:** the set built from all tokens that appear in the training data.

!	.	book	fairly	…	the	this
0	1	2	3		8	9

3. **Numerically encode:** replace each token with its index in the vocabulary.

Encoding text documents

	text	tokenized	encoded
x_1	"this book is good!"	[`this`, `book`, `is`, `good`, `!`]	[9, 2, 5, 4, 0]
x_2	"fairly long book"	[`fairly`, `long`, `book`]	[3, 6, 2]
x_3	"the book isn't good."	[`the`, `book`, `is`, `n't`, `good`, `.`]	[8, 2, 5, 7, 4, 1]
	...		

To represent text in a computer-friendly way, some things must happen:

1. **Tokenize:** split a string into a sequence of "tokens".
   (Roughly, think "words": but words are hard to define.)
   (Not easy! In some languages this is much harder than others!)
2. **Build vocabulary:** the set built from all tokens that appear in the training data.

!	.	book	fairly	...	the	this
0	1	2	3		8	9

3. **Numerically encode:** replace each token with its index in the vocabulary.

We are not done. Text is **sequential**, and sequences have different lengths.

How to design useful features?

Bag of words

Simple but powerful idea: for each vocabulary item, a feature that counts it:

$$h_i(x) = \text{number of occurrences of word } v_i \text{ in } x.$$

This leads to:

		!	.	book	fairly	good	is	long	nt	the	this
	text	h_1	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9
x_1	"this book is good!"	1	0	1	0	1	1	0	0	0	1
x_2	"fairly long book"	0	0	1	1	0	0	1	0	0	0
x_3	"the book isn't good."	0	1	1	0	1	1	0	1	1	0

…

Variants: zero-one, normalized frequencies.

Bag of words

Simple but powerful idea: for each vocabulary item, a feature that counts it:

$$h_i(x) = \text{number of occurrences of word } v_i \text{ in } x.$$

This leads to:

		!	.	book	fairly	good	is	long	nt	the	this
	text	h_1	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9
x_1	"this book is good!"	1	0	1	0	1	1	0	0	0	1
x_2	"fairly long book"	0	0	1	1	0	0	1	0	0	0
x_3	"the book isn't good."	0	1	1	0	1	1	0	1	1	0

...

Variants: zero-one, normalized frequencies.

Order is lost: $\boldsymbol{h}$("doesn't word order matter") = $\boldsymbol{h}$("word order doesn't matter")

Getting some structure back

Sequential order = a fundamental *structure* of language.

n-grams: treat *n consecutive tokens* as a single one.

Bigram tokenization:
"the book isn't good." $\rightarrow$ [`the_book`, `book_is`, `is_n't`, `n't_good`, `good_.`]

This captures some local order.

Can even combine: 1-gram $\cup$ 2-gram $\cup \ldots \cup$ *n*-gram: [1]

But, it comes at a cost: how many features are needed?

[1] Ensure combination is reversible or else we won't be able to distinguish features. For instance, here, _ must not appear in any unigram.

Don't forget about informed hand-crafted features:

length:
$h(x)$ = number of words in x
$h(x)$ = number of characters in x
$h(x)$ = number of sentences in x

lexicon counts:
$h(x)$ = number of times a word from some given, fixed set appears.
(e.g., positive lexicon = {"good", "great", "best", …})
comp. soc. science lexicons: hedges, first vs second vs third person pronouns, etc

measures of complexity:
$h(x)$ = avg. n. characters per word
$h(x)$ = avg. n. words per sentence (for longer docs)

Computational biology

Comp. bio applies computational analysis to understand biological systems.

“The central dogma:” DNA makes RNA makes proteins.

DNA:

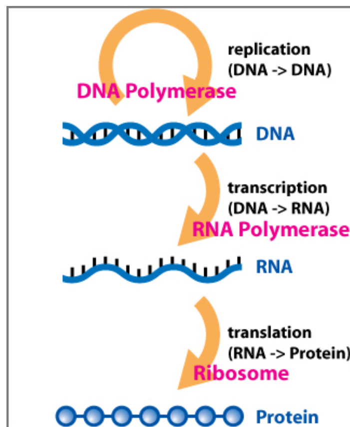
- genetic information: the “blueprint” for an organism.
- composed adenine, cytosine, guanine, thymine
- strands of DNA are sequences: `GATATGCACTTAA`...

RNA:

- regulatory role: catalyze reactions, control stuff.
- e.g., mRNA triggers protein synthesis

Protein:

- molecules that do the work in an organism
- e.g., antibodies, enzymes, transport, cell structure



`https://cm.jefferson.edu/learn/dna-and-rna/`

Encoding biological data: DNA

DNA sequences: treat as text, with “words” `A`, `C`, `G`, `T`.

Encoding biological data: DNA

DNA sequences: treat as text, with "words" `A`, `C`, `G`, `T`.

domain terminology: n-grams are called k-mers.

Encoding biological data: DNA

DNA sequences: treat as text, with “words” `A`, `C`, `G`, `T`.

domain terminology: n-grams are called k-mers.

Compared to English language: much much fewer possible words.

Encoding biological data: DNA

DNA sequences: treat as text, with "words" `A`, `C`, `G`, `T`.

domain terminology: n-grams are called k-mers.

Compared to English language: much much fewer possible words.

example: extract from the sequence below 1-mers and 6-mers.

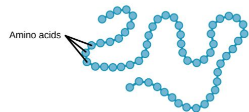
`aagacgcatcg`

Encoding bio data: Proteins

- **primary** structure: a sequence of aminoacids:

  `Gly - Ile - Val - Glu - ...` [a]

  We can use sequence encodings that we know.



[a] Abbreviations: `https://www.genome.jp/kegg/catalog/codes1.html`

[b] Figure modified from OpenStax Biology, CC BY 4.0.

Encoding bio data: Proteins

- **primary** structure: a sequence of aminoacids:

  `Gly - Ile - Val - Glu - ...` [a]

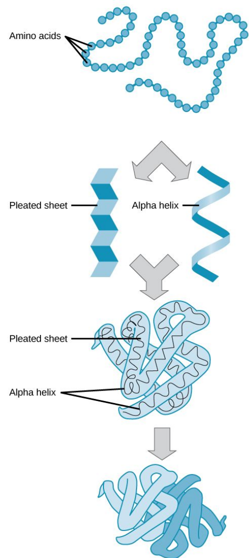
  We can use sequence encodings that we know.

- **higher-order** (secondary, tertiary, etc) structure:
  Folding due to interactions between (chunks of) aminoacids.
  We can encode as a graph: edges for interactions. [b]

---

[a] Abbreviations: `https://www.genome.jp/kegg/catalog/codes1.html`

[b] Figure modified from OpenStax Biology, CC BY 4.0.



Lecture 3

Designing Features For Structured Inputs

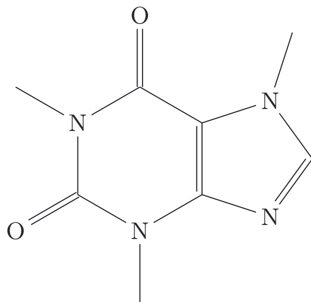
Part 3: Graphs

Machine Learning for Structured Data
Vlad Niculae · LTL, UvA · `https://vene.ro/mlsd`

Designing Features For Structured Inputs

1. Feature-based representations
2. Sequences
3. **Graphs**
4. Trees
5. Grids

Molecules

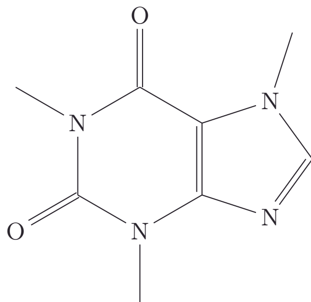


Molecules are graphs:

- atoms are nodes
- bonds are edges

What is the generalization of bigrams? trigrams?

Molecules



Molecules are graphs:

- atoms are nodes
- bonds are edges

What is the generalization of bigrams? trigrams?

hand-crafted “descriptor” features from domain knowledge:

- number of total atoms / bonds
- number of hetero atoms (not H,C)
- relative positive charge
  (highest charge / $\sum$ positive charges)

. . .

Lecture 3

Designing Features For Structured Inputs

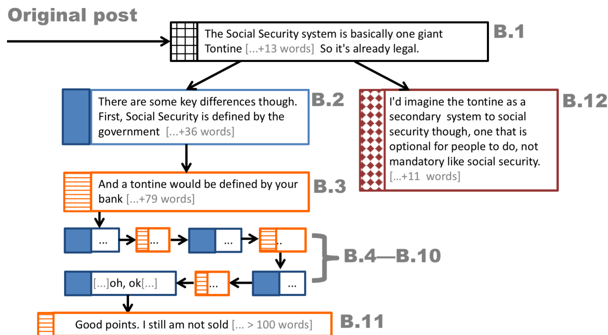
Part 4: Trees

Machine Learning for Structured Data
Vlad Niculae · LTL, UvA · `https://vene.ro/mlsd`

Designing Features For Structured Inputs

1. Feature-based representations
2. Sequences
3. Graphs
4. **Trees**
5. Grids

Tree-structured data: internet conversations



(figure cropped from Tan et al "Winning Arguments: Interaction Dynamics and Persuasion Strategies in Good-faith Online Discussions." Reproduced with authors' permission.)

structure is **within message** as well as **between messages.**

prompt-response pairwise features: why/because, so/though, . . .

descriptors:

- number of replies
- tree height (deepest path)
- . . .

Practical tricks for discrete features

B.1: "it's already legal" `[it's_already, already_legal]`

B.2: "some key differences though" `[some_key, key_differences, differences_though]`

ordered pairwise interactions:
`[(it's_already, some_key), (it's_already, key_differences), ...]`

Practical tricks for discrete features

B.1: "it's already legal" `[it's_already, already_legal]`

B.2: "some key differences though" `[some_key, key_differences, differences_though]`

ordered pairwise interactions:
`[(it's_already, some_key), (it's_already, key_differences), ...]`

Practical tricks for discrete features

B.1: “it’s already legal” `[it’s_already, already_legal, $]`

B.2: “some key differences though” `[some_key, key_differences, differences_though, $]`

ordered pairwise interactions:

```
[(it’s_already, some_key), (it’s_already, key_differences), ...]
```

Easily include unary features by adding a placeholder token.

```
    + [(it’s_already, $), ...] + [($, some_key), ...]
```

Practical tricks for discrete features

B.1: “it’s already legal” `[it’s_already, already_legal, $]`

B.2: “some key differences though” `[some_key, key_differences, differences_though, $]`

ordered pairwise interactions:
`[(it’s_already, some_key), (it’s_already, key_differences), ...]`

Easily include unary features by adding a placeholder token.
`+ [(it’s_already, $), ...] + [($, some_key), ...]`

Possibly a huge number of features
- most very rare
- even building the feature vocabulary is expensive
- can use a hashing trick (Count-Min) to prune rare features
- or encode features directly via hashing to save memory.

Lecture 3

Designing Features For Structured Inputs

Part 5: Grids

Machine Learning for Structured Data

Vlad Niculae · LTL, UvA · `https://vene.ro/mlsd`

Designing Features For Structured Inputs

1. Feature-based representations
2. Sequences
3. Graphs
4. Trees
5. Grids

Images

Images are 3d tensors $x \in \mathbb{R}^{W \times H \times C}$

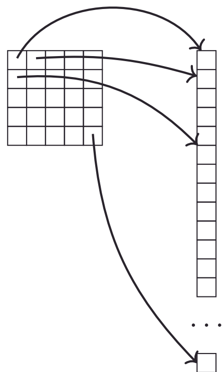
If all images have the same size, we could in theory use raw pixel features:

$h_{ij0}(x)$ = the percentage of red in pixel (i,j),

$h_{ij1}(x)$ = the percentage of green in pixel (i,j)...

$h_{ij2}(x)$ = the percentage of blue in pixel (i,j)...

What isn't great in this representation?

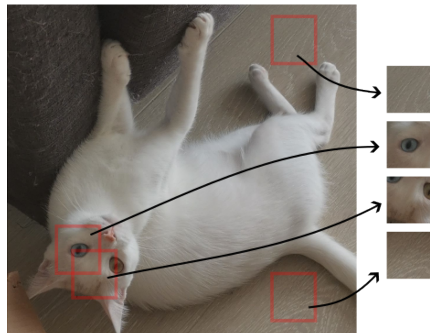


(not showing the 3rd dimension)

Image patch extraction

Patches = blocks of neighboring pixels.

More informative than a pixel alone!

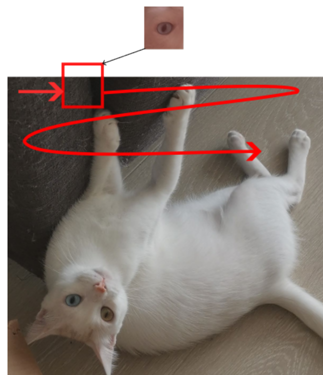


Filter matching: convolutions

Given a collection of known relevant patches (called filters) $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_d$, of some fixed dimension (e.g. 10x10 px)

$h_k(x)$ = whether filter $\boldsymbol{W}_k$ appears within the patches of image x.

Slide the filter over the image, checking at each position if it matches.



Filter matching: convolutions

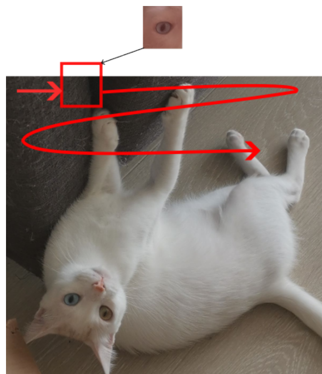
Given a collection of known relevant patches (called filters) $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_d$, of some fixed dimension (e.g. 10x10 px)

$h_k(x)$ = whether filter $\boldsymbol{W}_k$ appears within the patches of image x.

Slide the filter over the image, checking at each position if it matches.

Let $\boldsymbol{P}_{i,j}$ be the patch of x centered at pixel i, j. Then,

$$\textbf{hard matching} : m_{k,i,j} = \begin{cases} 1, & \boldsymbol{P}_{i,j} = \boldsymbol{W}_k, \\ 0, & \text{otherwise.} \end{cases}$$



Filter matching: convolutions

Given a collection of known relevant patches (called filters) $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_d$, of some fixed dimension (e.g. 10x10 px)

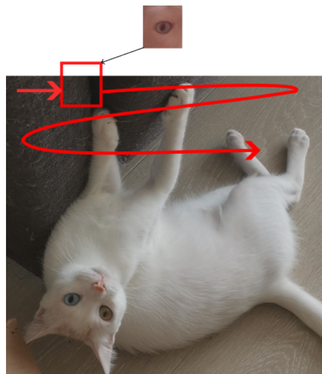
$h_k(x)$ = whether filter $\boldsymbol{W}_k$ appears within the patches of image x.

Slide the filter over the image, checking at each position if it matches.

Let $\boldsymbol{P}_{i,j}$ be the patch of x centered at pixel i, j. Then,

hard matching : $m_{k,i,j} = \begin{cases} 1, & \boldsymbol{P}_{i,j} = \boldsymbol{W}_k, \\ 0, & \text{otherwise.} \end{cases}$

soft matching: $m_{k,i,j} = \boldsymbol{P}_{i,j} \cdot \boldsymbol{W}_k$
(dot product; higher if more similar)

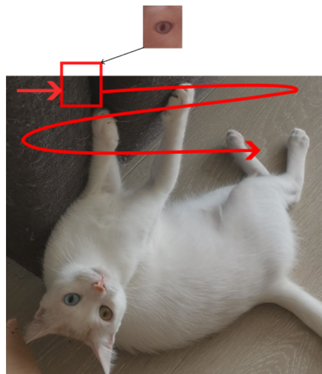


Filter matching: convolutions

Given a collection of known relevant patches (called filters) $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_d$, of some fixed dimension (e.g. 10x10 px)

$h_k(x)$ = whether filter $\boldsymbol{W}_k$ appears within the patches of image x.

Slide the filter over the image, checking at each position if it matches.



Let $\boldsymbol{P}_{i,j}$ be the patch of x centered at pixel i, j. Then,

hard matching : $m_{k,i,j} = \begin{cases} 1, & \boldsymbol{P}_{i,j} = \boldsymbol{W}_k, \\ 0, & \text{otherwise.} \end{cases}$

soft matching: $m_{k,i,j} = \boldsymbol{P}_{i,j} \cdot \boldsymbol{W}_k$
(dot product; higher if more similar)

Sliding window soft matching is called “convolution” (or, more accurately, cross-correlation.)

Shape of $m_{k,i,j}$? Can we use these ms as features?

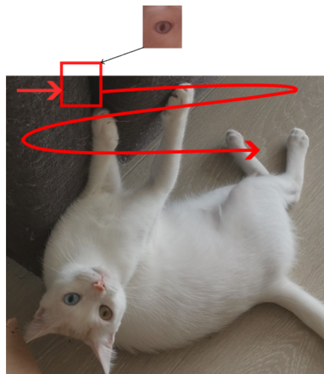
Filter matching: convolutions

Given a collection of known relevant patches (called filters) $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_d$, of some fixed dimension (e.g. 10x10 px)

$h_k(x)$ = whether filter $\boldsymbol{W}_k$ appears within the patches of image x.

Slide the filter over the image, checking at each position if it matches.

Let $\boldsymbol{P}_{i,j}$ be the patch of x centered at pixel i, j. Then,



hard matching : $m_{k,i,j} = \begin{cases} 1, & \boldsymbol{P}_{i,j} = \boldsymbol{W}_k, \\ 0, & \text{otherwise.} \end{cases}$

soft matching: $m_{k,i,j} = \boldsymbol{P}_{i,j} \cdot \boldsymbol{W}_k$
(dot product; higher if more similar)

Sliding window soft matching is called "convolution" (or, more accurately, cross-correlation.)

Shape of $m_{k,i,j}$? Can we use these ms as features?

Pooling: $h_k(x) = \max_{i,j} m_{k,i,j}$.

Aside: dot products

We're probably familiar with the dot product between vectors **of same dimension:**

$$\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d : \boldsymbol{a} \cdot \boldsymbol{b} := \sum_{i=1}^{d} a_i b_i$$

Aside: dot products

We're probably familiar with the dot product between vectors **of same dimension:**

$$\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d : \boldsymbol{a} \cdot \boldsymbol{b} := \sum_{i=1}^{d} a_i b_i$$

Sometimes it's more convinent to work with matrices and tensors: e.g., an image patch $\boldsymbol{P} \in \mathbb{R}^{w \times h \times c}$ is a tensor.

Sometimes this is not for mathematical reasons, but convenience, i.e., so we can easily point at the *red* channel as `P[:,:,0]`.

Mathematically, we can treat matrices and tensors as if they were vectors, flattened:

$$\boldsymbol{P}, \boldsymbol{F} \in \mathbb{R}^{w \times h \times c}, \quad \boldsymbol{P} \cdot \boldsymbol{F} := \sum_i \sum_j \sum_k p_{i,j,k} w_{i,j,k}$$

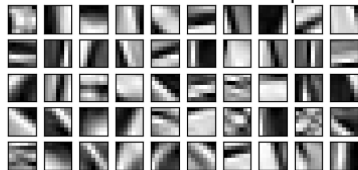
I'm not kidding, this is known as endowing the vector space $\mathbb{R}^{w \times h \times c}$ with the Frobenius inner product structure.

How to select a good collection of filters?

For very small size (eg 5x5), hand-crafted “edge detector” and “corner detector” patches are useful, but don’t say much about objects.

Larger patches: extract all patches from an entire dataset, and use some criterion to select the “interesting” ones (e.g., clustering.)
(quite costly..)

Automatic feature learning with deep networks: next time.



From scikit-learn example gallery, Image denoising using dictionary learning.

Feature design summary:

1. Feature-based representations
2. Sequences
3. Graphs
4. Trees
5. Grids