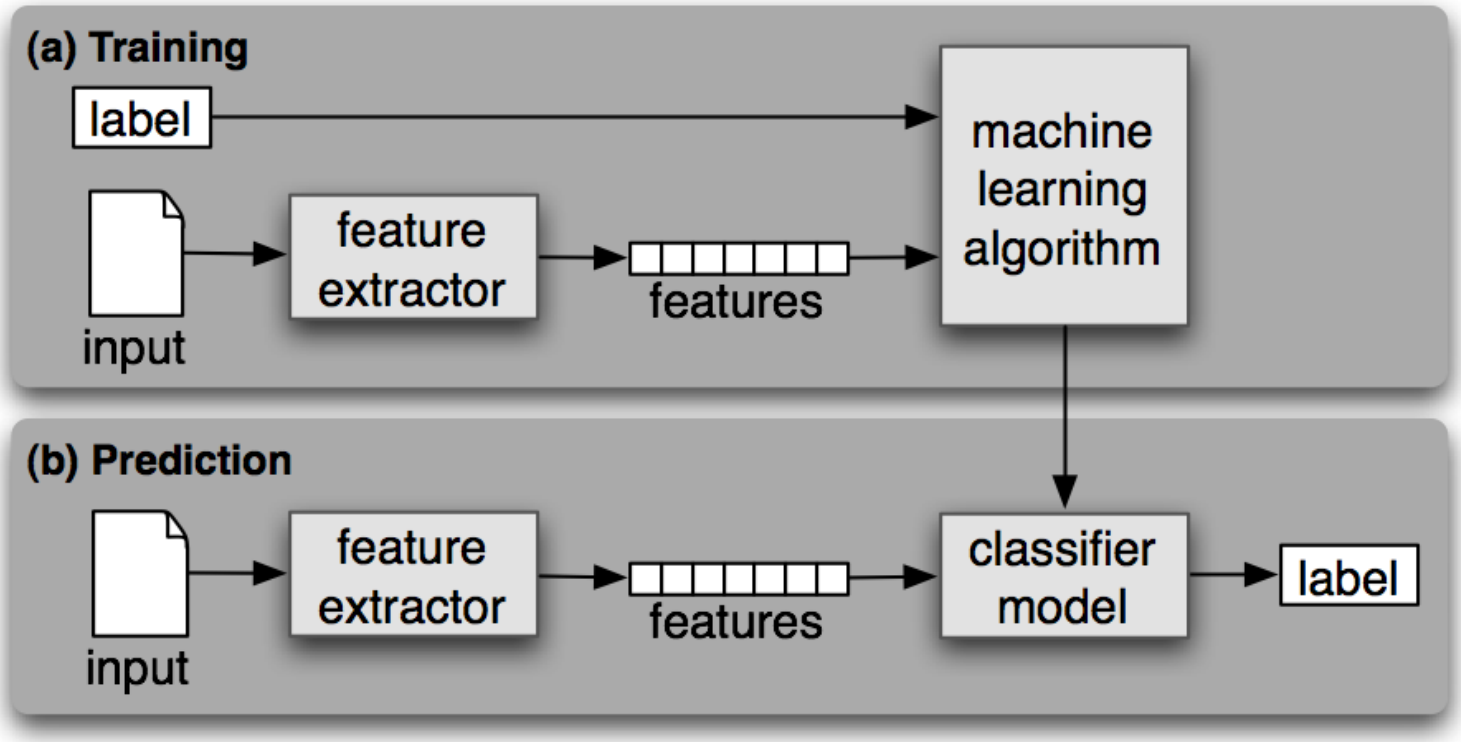
Introduction to Semantic Role Labelling

Behrang QasemiZadeh
zadeh@phil.hhu.de
Computational Linguistics Department, HHU – DRAFT

October 2018–January 2019

Recap: Semantic Role Labeling (SRL) as a Classification Task



[1]

[1] Source: `https://www.nltk.org/book/ch06.html`; (Bird et al., 2009, Chap. 6)

Recap: SRL as a Classification Task

We discussed:

Identification Problem: Finding candidates (constituents) which can potentially be semantic role fillers for a target verb;

Feature extraction: feature-based representation of candidates

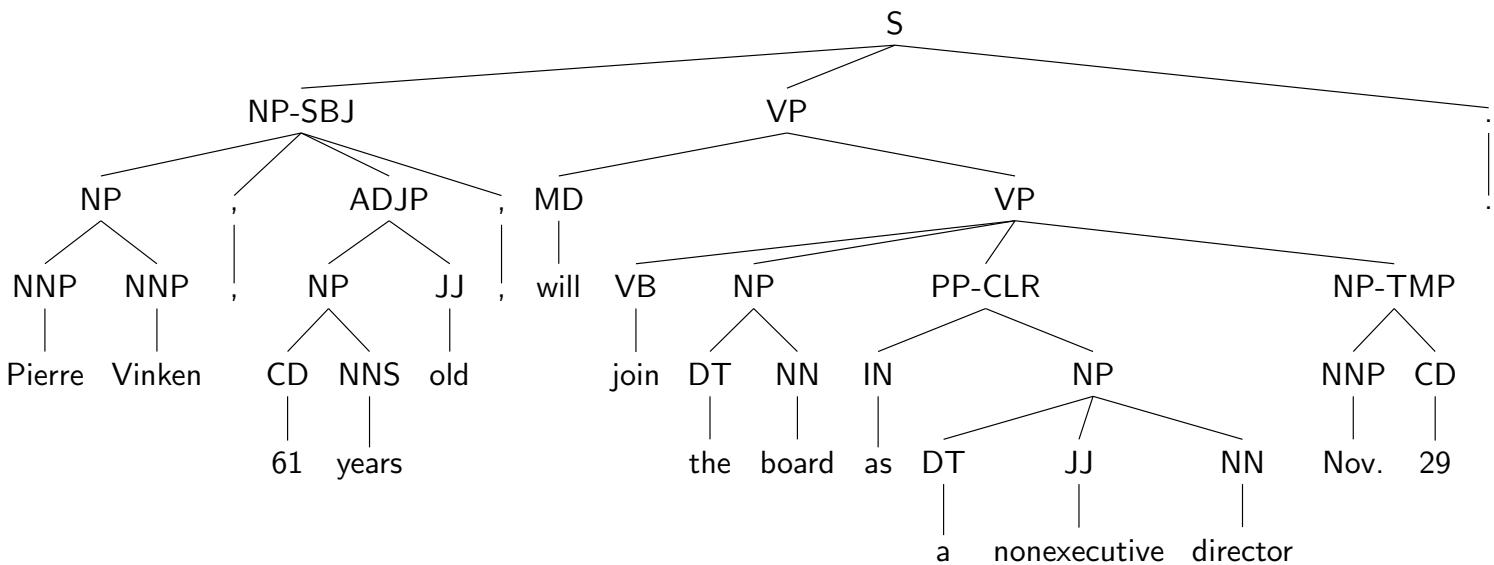
Additionally, we learned that for training purposes, we must assign class labels to candidates and their feature-based representations.

Recap: SRL as a Classification Task (contd.)

For example, for the verb join in:

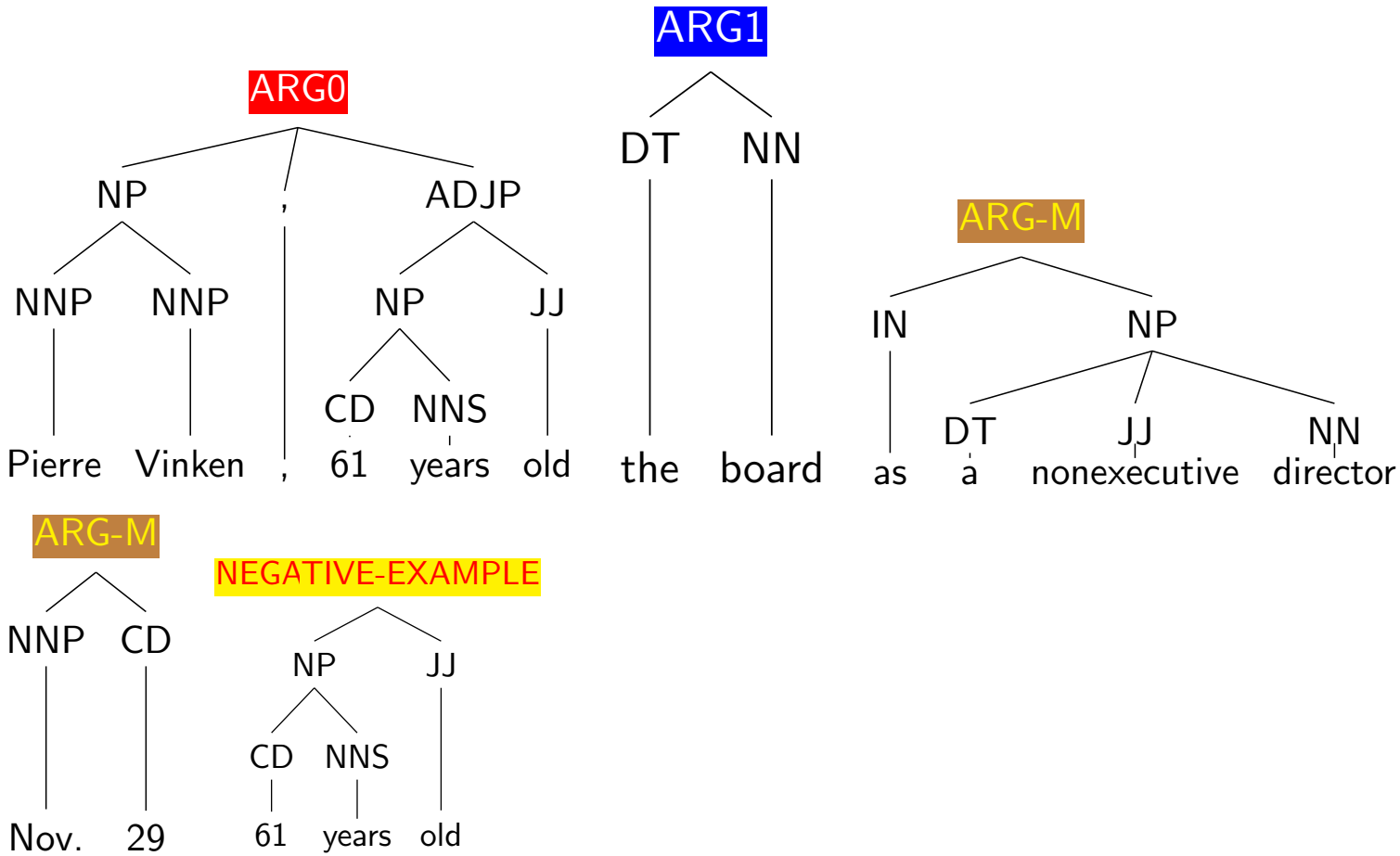
[ARG0 Pierre Vinken , 61 years old ,] [ARGM-MOD will] [rel join] [ARG1 the board] [ARGM-PRD as a nonexecutive director] [ARGM-TMP Nov. 29] .

and its constituent parse tree:



Recap: SRL as a Classification Task (contd.)

we can get candidates using a method like Xue and Palmer (2004) and label them:



Recap: SRL as a Classification Task (contd.)

And in turn, perform feature extraction for each candidate and sort and represent them in a tabular structure such as (one row per candidate, class labels and features listed as columns)

	Features				
Class/Role Label	Category	HeadWord	R-Position	SubCat	...
ARG0	NP	Vinken	Left	{N,N,P,N}	...
ARG1	NP	board	Right	{N,N,P,N}	...
...	...	...	...	...	...

Recap: SRL as a Classification Task (contd.)

The shape of this table is decided by two things:

a) The way you model your task (e.g. feature extraction process – what you define as feature adds (or removes) columns to (from) the table in the previous slide);

b) Your training data (alters the number of rows, the content of the table's cells, and depending on feature extraction the number of columns)

In short, (a) and (b), together, decide the number of rows and columns of your tabular representation (or the *structure* of it).

Recap: SRL as a Classification Task (contd.)

For example, one can choose to have two different sub-tasks for a) identification and b) for label assignment/classification; however, use the same feature extraction process for both:

	Features				
Class/Role Label	Category	HeadWord	R-Position	SubCat	...
ARG0	NP	Vinken	Left	{N,N,P,N}	...
ARG1	NP	board	Right	{N,N,P,N}	...

	Features				
Class Label	Category	HeadWord	R-Position	SubCat	...
Positive	NP	Vinken	Left	{N,N,P,N}	...
Positive	NP	board	Right	{N,N,P,N}	...
NEGATIVE	NP	years	Right	{N,N,P,N}	...

Please pay attention to the rows and the class labels.

Recap: SRL as a Classification Task (contd.)

Alternatively, one may combine the two sub-tasks in one by coming up the following representation:

	Features				
Class/Role Label	Category	HeadWord	R-Position	SubCat	...
ARG0	NP	Vinken	Left	{N,N,P,N}	...
ARG1	NP	board	Right	{N,N,P,N}	...
NEGATIVE	NP	years	Right	{N,N,P,N}	...

(simply use negative samples as an additional class when training the semantic labeler)

Please look at Class/Role label and their representations shown earlier in our training data?

Is it clear that the number of rows of this table depends on your training data? and that the number of columns depends on features used to represent them?

Obviously, these are not the only possible designs/situations!

Into Machine Learning

+ Once the feature extraction is done, any supervised learning algorithm can be used for building a prediction model.
+ There are numerous choices: probabilistic generative models, discriminative learning techniques using vector space mathematics, example-based (aka memory-based) learning, deep neural networks, information-theoretic models, decision trees and association rules, ...
+ There is only one trick: Each class of these learning methods, expect your extracted features in a specific format, which is decided mostly by their underlying mathematical framework.

Into Machine Learning (contd.)

There is only one trick: Each class of these learning methods, expect your extracted features in a specific format.

Unfortunately, a majority of learning algorithms cannot directly work on symbolic feature representations such as:

ARG0[NP Vinken Left {N,N,P,N} . . .]

for a candidate constituent (here of type ARG0 but of unknown type during prediction).

Instead, many algorithms expect numerical representations like

ARG0[0 1 0 0 1 0 0 0 0 0 1 . . .]

Into Machine Learning (contd.)

or,

ARG0[.23 .34e-7 0 -3.12 .26 0.93 0.78 0.787 . . .]

or even a matrix/tensor such as:

ARG0[

0.61	0.5	0.4	0.93	0.25	0.87	0.45	0.76	0.3	0.37	0.78	0.32
0.46	0.07	0.64	0.71	0.34	0.79	0.93	0.28	0.09	0.9	0.97	0.07
0.27	0.38	0.8	0.32	0.08	0.73	0.23	0.68	0.45	0.25	0.03	0.76
0.78	0.29	0.58	0.18	0.42	0.44	0.07	0.67	0.21	0.97	0.06	0.21

]

Into Machine Learning (contd.)

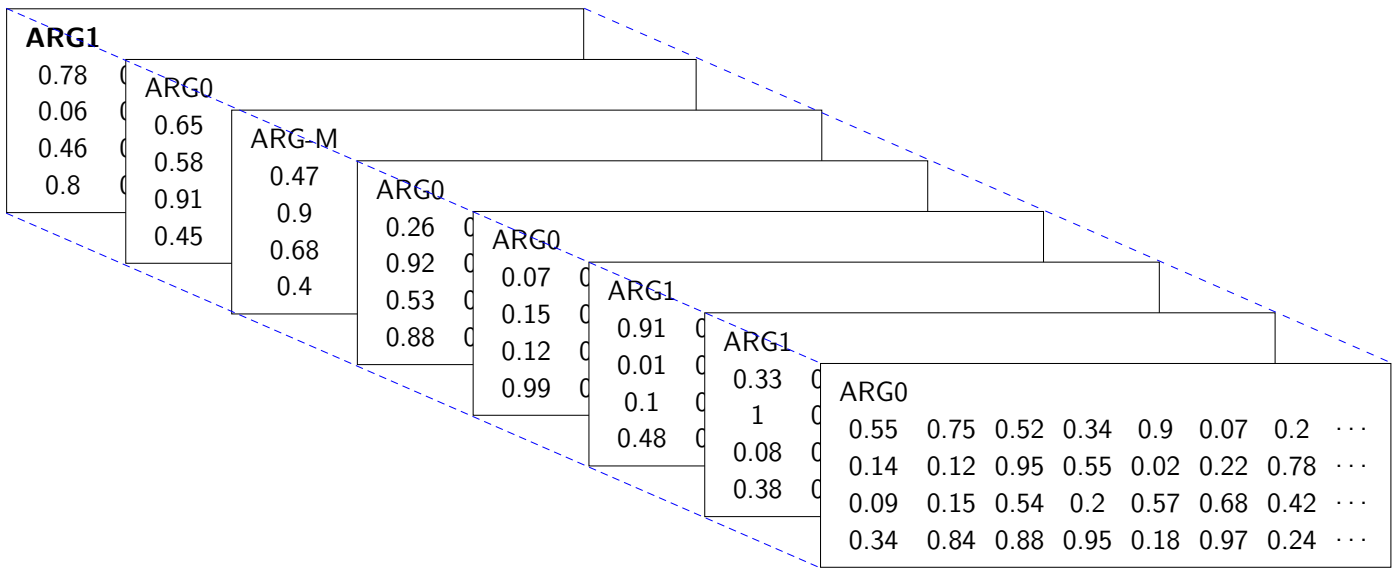
These numerical records merged in matrix-like structures.

For a number of training records which are represented as an array of numbers, they form a matrix (vector spaces, contingency tables, or if it easier to conceptualize, an abstract 2-d coordinate system) such as:

ARG1	0.78	0.38	0.27	0.59	0.79	0.88	0.49	$\cdots$
ARG0	0.06	0.94	0.81	0.87	0.69	0.76	0.35	$\cdots$
ARG-M	0.46	0.05	0.78	0.67	0.62	0.58	0.25	$\cdots$
ARG0	0.8	0.11	0.56	0.24	0.42	0.94	0.92	$\cdots$
ARG1	0.3	0.21	0.16	0.24	0.42	0.24	0.02	$\cdots$

Into Machine Learning (contd.)

But, n-dimensional arrays (tensor, coordinate systems, etc.) are also common. For example, if each of our records are represented by a matrix, then their aggregation is a tensor like:



Into Machine Learning (contd.)

As we saw, most of our features have as their value a symbol, e.g.: for *path feature* we have values such as "VP↓NP", "NP↑VP↓NP", ...; similarly, the assigned values to *governing category* are "NP", "VP", ... These, somehow, must be transformed to numbers!

There is no unique way for converting these "literal values" to numeric ones (but there are some known practices, and caveats around them).

Mapping from "literal features" to numerical representations must be decided with respect to the chosen learning algorithm.

In its simplest form, (colloquially put it), our non-numeric feature values can be converted to sequences of 0s and 1s by assuming our features being Boolean-valued (e.g., suitable for linear SVM).

Into Machine Learning (contd.)

Steps for converting raw-feature representations to high-dimensional sparse numerical vectors:

- Collect all the feature types and their associated values (feature value pairs f_v) in your human-readable feature records; (if you like, form a string such as is- or has- + feature name + feature-value!);
- Count distinct feature-value pairs in your data (e.g., let's assume we arrive to the total number of n);
- Assign each feature-value pair f_v to **a unique index number** i in which $1 \leq i \leq n$ (in computer codes we often call this a feature map);

Into Machine Learning (contd.)

- For each training record in your training set, follow the steps below:
  + instantiate a *vector* $\vec{v}$ of size (dimension) n; all components of $\vec{v}$ are set to 0 (keep track of the associated class label to the record and thus $\vec{v}$);
  + for each f_v of index x appeared in the training record, increment the value the v_x component of $\vec{v}$.
- If the training set contains m records, then the result from the above procedure is a $m \times n$ matrix;
- Pass this matrix alongside your class labels to a classifier such as Support Vector Machines to build a model.

Numerical feature vector: Example

Let's review the conversion process through a simple example. Let's say we have only 3 records, each represented by features *Category*, *HeadWord*, *R-Position*, and *SubCat*, shown blow:

	Features			
Class/Role Label	Category	HeadWord	R-Position	SubCat
ARG0	NP	Vinken	Left	{N,N,P,N}
ARG1	PP	board	Right	{N,N,P,N}
ARG0	NP	company	Left	{N,N}

Table 1: Raw Feature Representation

* Collect all the feature types and their associated values (feature value pairs f_v) in your human-readable feature records.

Numerical feature vector: Example (contd.)

	Features			
Class/Role Label	Category	HeadWord	R-Position	SubCat
ARG0	NP	Vinken	Left	{N,N,P,N}
ARG1	PP	board	Right	{N,N,P,N}
ARG0	NP	company	Left	{N,N}

Result: is-HeadWord-Vinken, is-HeadWord-board, is-HeadWord-company, is-Category-NP, is-Category-NP, is-Category-PP, is-R-Position-Left, is-R-Position-Right, is-SubCat-{N,N,P,N} is-SubCat-{N,N,P,N}, is-SubCat-{N,N}, ...

Numerical feature vector: Example (contd.)

* Count distinct feature-value pairs in your data (e.g., let's assume we arrive to the total number of n);
* Assign each feature-value pair f_v to **a unique index number** i in which $1 \leq i \leq n$ (in computer codes we often call this a feature map);

Distinct Element Count: 9

f_v to unique index i (feature map):

is-Category-NP	is-Category-PP	is-HeadWord-Vinken	is-HeadWord-board
1	2	3	4

is-R-Position-Left	is-R-Position-Right	is-SubCat-{N,N,P,N}	is-HeadWord-company	is-SubCat-{N,N}
5	6	7	8	9

Table 2: Feature Map: Keep track of indices

Numerical feature vector: Example (contd.)

For each training record in your training set

* instantiate a $\vec{v}$ of size n (here $n = 9$);
* for each f_v of index x in the training record, increment v_x.

$$\vec{v_{Arg0}} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Given our raw feature representation (Table 1) feature map (Table 2):

`index`(is-Category-NP)=1;

`index`(is-HeadWord-Vinken)=3;

`index`(is-R-Position-Left)=5;

`index`(is-SubCat-{N,N,P,N})=7

$$\vec{v^1_{Arg0}} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Numerical feature vector: Example (contd.)

is-Category-NP	is-Category-PP	is-HeadWord-Vinken	is-HeadWord-board
1	2	3	4

is-R-Position-Left	is-R-Position-Right	is-SubCat-{N,N,P,N}	is-HeadWord-company	is-SubCat-{N,N}
5	6	7	8	9

$$\vec{v^1_{Arg0}} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$
$$\vec{v^2_{Arg1}} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$
$$\vec{v^3_{Arg0}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Numerical feature vector: Example (contd.)

* If the training set contains m records, then the result from the above procedure is a $m \times n$ matrix;
* Pass this matrix alongside your class labels to a classifier such as Support Vector Machines to build a model.

with some simplification and assumptions:

$$\begin{pmatrix} Arg0 \\ Arg1 \\ Arg0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

We have 3 vectors of dimension$= 9$, i.e., a 3×9 matrix, which can be passed to a SVM for learning a model.

Numerical feature vector: Example (contd.)

For predicting, we use the same feature extraction/conversion (including the feature map): The resulting vectors are passed to classifier. Given an unlabeled input, the classifier assign a class label to input such as ARG0, ARG1, etc. (i.e., labels used during training).

For instance, as its input our classifier gets

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

and as its output it gives us a label such as:

ARG0

Numerical feature vectors

The simplified one-hot/Boolean vectors are not sufficient for a) representing features of continues value, and b) for processing in several classification frameworks. For instance:

- You may define a feature as the *normalized length of path* which can take a value between 0 and 1; and/or,
- The frequency of head-word and target verb collocations; ...

In this case, although we can use the same procedure (feature map) for inserting them in our numeric representation (and/or use some method for converting continues values to discrete ones) extra caution is required, the least of which is to make sure that they are scaled properly.

Numerical feature vectors (contd.)

Our Boolean representation may also be redundant (although not necessarily) for probabilistic methods such as

in the so-called lattice backoff method of Gildea and Jurafsky (2002), in which we must calculate probabilities of class labels given a feature-vector/observations.

Naive Bayesian Classifier

Let's say we want to use a naive Bayesian classifier, for which we need to compute joint probabilities.

Even if we assume that all our feature are discrete, and that they are modelled as discussed in the past few slides, we still need to do some "processes" on our representation to compute probabilities.

We could do these "processes" on demand, i.e., every-time we want to compute a probability, but that would be inefficient and time consuming (this is mostly what is done under the label training):

> *We process our feature vectors to a contingency table (e.g., keep track the sum of values in the rows and columns and the whole table); compute and cache some of probabilities, etc.*

Naive Bayesian Classifier (contd.)

We compute probabilities directly on our feature representation Table 1. But, this does not mean that we cannot use the Boolean-valued table representation.

Naive Bayesian Classifier (contd.)

Given a set of observations/symptoms/effects/features $X = [x_1, \ldots, x_n]$ (i.e., our feature vectors) for a candidate constitute, what is the probability of it being of semantic role s_i? (for all s_i in our training set, e.g., ARG0, ARG1, etc.)

$p(s_i|X) = \frac{P(X|s_i) \times p(s_i)}{P(X)}$ (i.e., the Bayes' theorem).

$p(X)$ is a constant value and can be removed from computations:

$p(s_i|X) = p(X|s_i) \times P(s_i)$.

which in turn (using joint probabilities, chain rule and conditional independence):

$p(s_i|x_1, \ldots, x_n) = p(s_i) \prod_{i=1}^{n} p(x_i|s_i)$

Naive Bayesian Classifier (contd.)

$p(s_i|x_1, \ldots, x_n) = p(s_i) \prod_{i=1}^{n} p(x_i|s_i)$

$p(x_i|s_i)$ and $P(s_i)$ can be computed easily from the contingency/probability table, right?!

Why conditional independence assumption? When we add more features, the total number of the representations for the joint distribution explodes, to the extent that it becomes computationally intractable to compute.

To solve this problem, we use conditional independence, i.e., by assuming that observing one feature (e.g., *is-R-Position-Left*) has nothing to do with observing another one (e.g., observing *is-HeadWord-company*).

Naive Bayesian Classifier (contd.)

To classify a candidate, therefore we compute

$$P(s_i|X) = p(s_i)\prod_{i=1}^{n} p(x_i|s_i)$$

for all class labels s_i in our data C (e.g., $C = \{ARG0, ARG1, \ldots\}$) and we choose the s_i with the maximum probability, which is often written as:

$s = \arg\max_{s_i \in \{ARG0, ARG1, \ldots\}} P(s_i|X)$, or equivalently as
$s = \arg\max_{s_i \in \{ARG0, ARG1, \ldots\}} p(s_i)\prod_{i=1}^{n} p(x_i|s_i)$

Test Naive Classifier

Question: Given feature Table 1 as training data, what is the probability of a candidate with the following feature vector being of class ARG0 (sorry if it is too simple for you)?!

Category	*HeadWord*	*R-Position*	*SubCat*
NP	*Vinken*	*Left*	$\{N,N\}$

Test Naive Classifier (contd.)

We use the general formula:

$$P(s_i|X) = p(s_i)\prod_{i=1}^{n} p(x_i|s_i)$$

Since we look for the probability of the class ARG0

$$P(ARG0|X) = p(ARG0)\prod_{i=1}^{n} p(x_i|ARG0)$$

and we have

$X = [Category = NP, HeadWord = Vinken, R - Position = Left, SubCat = \{N, N\}]$

indeed, $p(ARG0) = \frac{2}{3}$ and that $p(Category = NP|ARG0) = 1$,
$p(HeadWord = Vinken|ARG0) = \frac{1}{2}$,
$p(R - Position = Left|ARG0) = 1$, and
$p(SubCat = \{N, N\}|ARG0) = \frac{1}{2}$, which gives us

$p(ARG0|X) = \frac{2}{3} * 1 * \frac{1}{2} * 1 * \frac{1}{2} = \frac{1}{6} \approx 0.16$

Smoothing/Interpolation/Back-off; General Idea

In real applications, frequently we must compute class probabilities from large feature vectors $X = \{x_1 \dots x_n\}$, in which for some $x_j = v$ we have $p(x_j = v|s_i) = 0$. This $p(x_j = v|s_i) = 0$ cancels the effect of all other $p(x_i|s_i)$s (remember we have a product in our formula).

A usual cause for this problem are feature values (or their combinations) that are not seen in the training set.

Smoothing/Interpolation/Back-off; General Idea (contd.)

For example, given Table 1 as training data, what is the probability of the following record being ARG0 or ARG1:

Category	*HeadWord*	*R-Position*	*SubCat*
NP	*John*	*Left*	$\{N,N\}$

The result is 0 for both classes (but isn't more like ARG0)?!

We would need to compute class probabilities when HeadWord=John: The probability of $p(HeadWord = John|Arg_0)$, $p(HeadWord = John|Arg_1)$ . . . aren't they all zero?!

Given our formula $P(s_i|X) = p(s_i)\prod_{i=1}^{n} p(x_i|s_i)$, we arrive to $p(s_i|X) = 0$ since 0 multiplied to other probabilities yields 0.

Smoothing/Interpolation/Back-off; General Idea (contd.)

To avoid the problem caused by $p = 0$, we use smoothing/interpolation/back-off methods.

For instance, using additive smoothing, instead of $p(HeadWord = John|Arg_0) = \frac{0}{3}$, we can use $p(HeadWord = John|Arg_0) = \frac{1}{3+3}$ (one additional 1 per number of distinct feature values in denominator)

Smoothing/interpolation (aka Jelinek-Mercer smoothing)/back-off (Katz smoothing), which are named in our textbook, are solutions for problems such as sketched above (unknown or zero probability).

Other smoothing techniques used in NLP tasks: Good-Turing estimate, Stupid back-off, Witten-Bell smoothing, Kneser-Ney smoothing, Discounting smoothing, ...

Naive classification method wrap-up

These very simple ideas can lead you to really complex systems.

The basic steps, building feature tables, forming vectors, or computing probabilities are simple essential steps that are required in any sophisticated system.

Word Embedding and Dimension Reduction

In modern neural networks, the sparse feature vector is often replaced by (or appended to) a set of "word/symbol embeddings".

Essentially, word embedding is another name for feature vectors.

In word embeddings (aka. word vectors), the vectors/embeddings represent words. These vectors often assumed to have a low dimension, e.g., $50 < n < 1500$.

Word vectors are essentially the same as feature vectors: Features simply capture a correlation between word co-occurrences (sometimes a little more than word co-occurance/collocations).

Word Embedding and Dimension Reduction (contd.)

Word vectors and the correlations can be learned from large corpora using a neural network (e.g., word2vec), or using a combination of other (perhaps simpler) statistical techniques:

* Let's assume we have a corpus which contains n *distinct* words;
* Assign each word w to a unique index $1 \leq i \leq n$;
* To each word w_i assign an empty n-dimensional vector; $\vec{w}_i$;
* Scan the input corpus, if w_i co-occurs with w_j (e.g., within the same line or chunk of text) then $\vec{w}_i[index(w_j)] + +$ and $\vec{w}_j[index(w_i)] + +$ ($++$ means increment by 1), which means:
  + Find the assigned index to w_i and w_j (i.e., $index(w_i)$ and $index(w_j)$, e.g., $index(w_j) = x$ and $index(w_i) = y$).

Word Embedding and Dimension Reduction (contd.)

+ Assert the co-occurrence of these words simply through changing the component x and y of $\vec{w}_i$ and $\vec{w}_j$, i.e., $\vec{w}_i[x] = \vec{w}_i[x] + 1$ and $\vec{w}_j[y] = \vec{w}_j[y] + 1$

* Out of this process you will have a co-occurrence matrix, which you can use for building word embeddings.

Essentially, one can observe the co-occurrence matrix as a set of feature vectors discussed earlier. What is listed below (dimension reduction and weighting) can (and in some cases must) be applied to feature vectors, too.

Word Embedding and Dimension Reduction (contd.)

Once you have a co-occurrence matrix, you can use a dimensionality reduction method to reduce the size of the raw n-dimensional word vectors, e.g., from $n = 10,000,000$ to $m = 1000$.

Several dimension reduction methods are available:

* Heuristics–by intuition or math: select one of many correlated features, etc.
* Distance Preserving Methods – using normed-space definitions:
  * Singular Value Decomposition (SVD) for ℓ_1 and ℓ_2 spaces;
  * Principle Component Analysis;
  * Random Projections using projection matrices of $\mathcal{N}(0,1)$ for ℓ_2 and $\mathcal{C}(0,1)$ for ℓ_1;
  * etc..

Word Embedding and Dimension Reduction (contd.)

* Hashing techniques ...
* Application-oriented Optimization Techniques: an optimization of a domain-specific function is solved, e.g., use the probability of word sequences (most neural networks):

  Word2Vec; GolVe; Directional Skip-Gram; ...

  In general, use a layer (usually hidden layer) of neural nets.

Disregarding of the employed technique, the output is a set of vectors of reduced dimension.

Obviously, instead of word-by-word matrices, one can build matrices of any type of record-by-feature and pass them to these dimension reduction methods. E.g. We can use them for reducing the dimension of feature-vectors we built for semantic role labeling.

Word Embedding and Dimension Reduction (contd.)

Why dimension reduction? many motivations can be listed ...

Avoiding the curse of dimensionality problem;

Enhancing/Reducing computation time;

Removing noise, correlated/duplicate features;

Increasing the overall discriminatory power of vectors;

...

Word Embedding and Dimension Reduction (contd.)

Some methods merge co-occurence matrix construction and dimension reduction into one process (QasemiZadeh et al., 2017):

1: Initialize a zero m-dimensional vector $\vec{w}$
2: **for each** c co-occurred with w **do**
3: $\quad \mathtt{d} \leftarrow \mathtt{abs}(\mathtt{hash}(c)\ \%\ m)$
4: $\quad \vec{w}_{\mathtt{d}} = \vec{w}_{\mathtt{d}} + 1$
return $\vec{w}$

The resulting $\vec{w}$ is a m-dimensional embedding of the word w that $\vec{w}$ represents, c is the co-occurring context word/feature with w, % is modulo operator, and `hash` is a hash function that returns an integer.

QasemiZadeh et al. (2017) state that these vectors, however, must go through a weighting process.

A quick detour to Weighting techniques

Apart from dimension reduction, weighting techniques are often used to remove noise and boost the impact of certain features according to some criteria.

The most well-known technique applied to *document-by-word* models (for text classification, information retrieval applications) is $tf - idf$ (term frequency – inverse document frequency):

a) Words that appear in all documents (e.g., 'a', 'the', 'that', 'this', etc.) are not important.

b) Words that appear frequently in some documents but infrequently across documents are important.

A quick detour to Weighting techniques (contd.)

The two ideas above are boiled down into the *tf-idf* formula:

$$\textit{tf-idf}(w_{ij}) = w_{ij} * \log(\frac{N}{D})$$

- w_{ij} is the number of occurrences of word j in the document i;
- D is the total number of documents in which w_j appears, i.e., the count of rows in which j component is greater than 0;
- N is a constant: total number of documents (number of rows).

To weight feature matrix with tf-idf means to replace the value of cells with their respective tf-idf values.

A quick detour to Weighting techniques (contd.)

Another popular weighting technique is Positive Point-wise Mutual Information (PPMI). Given a $M_{p \times n}$ model, in which v_{xy} is the yth component of the xth vector:

$$ppmi(v_{xy}) = \max(0, \log \frac{v_{xy} \times \sum_{i=1}^{p} \sum_{j=1}^{n} v_{ij}}{\sum_{i=1}^{p} v_{iy} \times \sum_{j=1}^{n} v_{xj}} \quad (1)$$

* $M_{p \times n}$ means that we have a matrix with p rows and n columns (in our convention, p is the number of records and n is the number of features).

The list of weighting methods is long: Mutual Information, Odds Ratio, Simple Probabilities, etc.

NOTE: Scaling and Weighting go hand in hand!

Embeddings as Input Features to Neural Nets

Let's get back to neural-network based methods ...

Modern neural net models expect embeddings as their input.

For instance, you can use the model from (He et al., 2018) for joint identification. To use this system:

+ Produce a predicate-specific word representation x_i for each word w_i, where i is the position of word in the input sequence;
+ To do so, word representation x_i is the concatenation of several types of features, represented by low-dimensional vectors (embedding):

Embeddings as Input Features to Neural Nets (contd.)

- predicate specific indicator embedding *ie*,
- character-level embedding *ce*,
- randomly-initialized word embedding *re*,
- pre-trained word embedding *pe*,
- randomly-initialized lemma embedding *le*,
- randomly initialized POS-tag embedding *pos*, and
- randomly-initialized embedding for dependency relation to head *de*.

Finally, x_i is the concatenation of $x_i^{ie}, x_i^{ce}, x_i^{re}, x_i^{pe}, x_i^{le}, x_i^{pos}$, and x_i^{de}.

See also resources from their earlier work here:
`https://github.com/luheng/deep_srl`.

Embeddings as Input Features to Neural Nets (contd.)

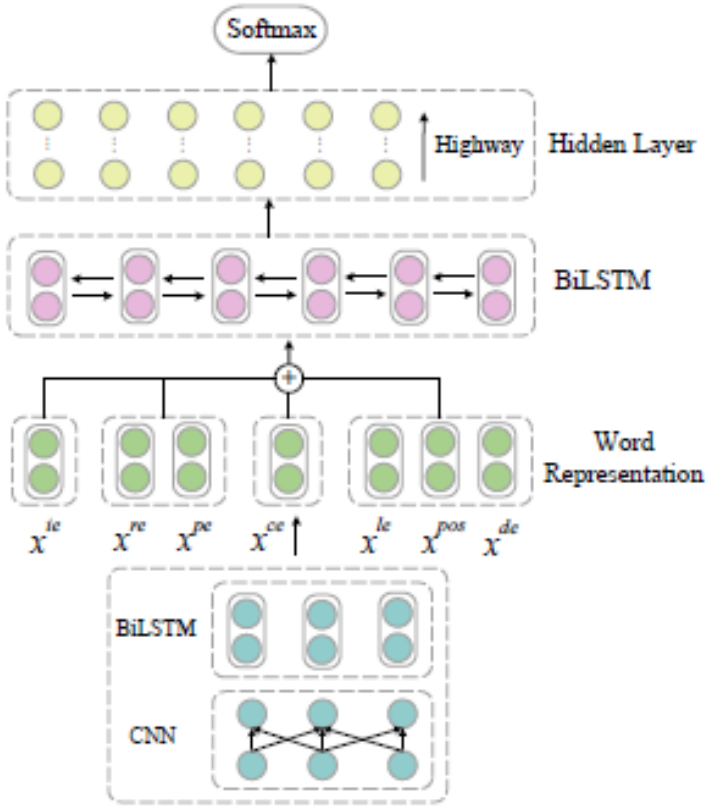


Figure 1: A state-of-the-art NN-based SRL (He et al., 2018).

Joint Inference

So far, we modelled semantic role labeling as Identification followed by a number of independent Classifications:

In the identification step: Find all the candidates for a given target verb (e.g., let's assume identification gives us C_1, C_2, and C_3).

In the classification step, we must label each of the candidates with a class label, e.g., to label C_1, C_2, and C_3 as ARG0, ARG1, ARG2, ARG-M, etc.

So far, in our small examples, we classified C_1 independently of the label for C_2, and C_3; C_2, independently of C_1, C_3 and so on.

This is not very clever for obvious reasons: these decisions are highly dependant on each other, e.g., only one of C_1, C_2, and C_3 can be ARG0.

Joint Inference (contd.)

How to model dependence of classification tasks? There are numerous ways.

As suggested by Palmer et al. (2010), one can

A) classify each candidate independently and assign them a ranked listed of class labels (e.g., based on probabilities given by our naive method);

B) once these classifications for all the candidates are done, choose among the possible answers the best one, i.e., a combination that optimizes some formula.

Note that as a result of the method above, not always the best answer/class (e.g., most probable class) for a candidate is chosen.

Joint Inference (contd.)

Palmer et al. (2010) list solutions which are proposed by different techniques/frameworks:

* Reranking (a combination of class labels which gives the overall highest probability – uses smoothinh – back-off)
* Using a Viterbi search, Hidden Markov Models (very similar to Reranking technique, use probabilities);
* Integer Linear Programming (ILP)

Dynamic Programming is the major common theme between these methods (such as used also in the *Travelling salesman problem* algorithm, or the *Bellman–Held–Karp* algorithm).

Divide the problem into sub-problems often in a recursive manner;

Solve the sub-problems;

Joint Inference (contd.)

Keep track of the sub-solutions and their combinations;

Find the overall solution that gives the most desirable result according to some criteria.

NB: In many cases, finding the solution for one sub-problem depends on finding the solution for another sub-problem.

In some methods, we simply assume that we know the solution for a sub-problem and based on this assumption we solve the remaining problems.

For example, given arguments $a_1 \ldots a_n$ for a target verb, we can assume that the class labels of a_is are input features for a_j $(i \neq j)$.

Initially, we do not know what are the class labels for $a_1 \ldots a_n$, but instead we assume that they are know.

Joint Inference (contd.)

We solve the problem based on all possible assumptions and their respective solutions and pick one that yields the best outcome according to some criteria (e.g., the overall likelihood).

E.g., the pseudocode of the Viterbi algorithms.

Evaluation

Central to Identification: Find the number of correctly identified arguments.

Central to classification: Find the number of correctly identified labels.

Using the absolute numbers as a measure may not make sense; instead we a trad-off between precision (the amount of answers in the system's output that are correct) and recall (the portion of all the expected correct answers in the system output).

Evaluation (contd.)

We report the harmonic mean of precision P and recall R (known as F-Score) as a figure of merit:

$$F = 2 \times \frac{P \times R}{P + R}$$

or, in general

$$F_\beta = (1 + \beta^2) \times \frac{P \times R}{\beta^2 \times P + R}$$

where β is a constant, e.g., 0.7 to put more emphasis on precision or recall.

Evaluation (contd.)

For identification sub-task, there are several reports of relaxing the exact match criteria.

Palmer et al. (2010) suggest that apart for argument identification and classification, the evaluation framework can be extended to account for predicate identification and classification (to assign sense category labels to predicates – word sense disambiguation).

Bibliography

Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition.

Gildea, D. and Jurafsky, D. (2002). Automatic labeling of semantic roles. *Comput. Linguist.*, 28(3):245–288.

He, S., Li, Z., Zhao, H., and Bai, H. (2018). Syntax for semantic role labeling, to be, or not to be. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2061–2071, Melbourne, Australia. Association for Computational Linguistics.

Palmer, M., Gildea, D., and Xue, N. (2010). *Semantic Role Labeling*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers.

QasemiZadeh, B., Kallmeyer, L., and Passban, P. (2017). Sketching word vectors through hashing. *CoRR*, abs/1705.04253.

Bibliography (contd.)

Xue, N. and Palmer, M. (2004). Calibrating features for semantic role labeling. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*.