Advanced Human Language Technologies

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Outline

NERC using machine learning

Assignment

The main program parses all XML files in the folder given as argument and recognizes and classifies drug names, calling a sequence-tagging machine learning algorithm.

```
$ python3 ./ml-NER.py data/devel result.out
$ more result.out
DDI-DrugBank.d278.s0|0-9|Enoxaparin|drug
DDI-DrugBank.d278.s0|93-108|pharmacokinetics|group
DDI-DrugBank.d278.s0|113-124|eptifibatide|drug
DDI-MedLine.d88.s0|15-30|chlordiazepoxide|drug
DDI-MedLine.d88.s0|33-43|amphetamine|drug
DDI-MedLine.d88.s0|49-55|cocaine|drug
DDI-MedLine.d88.s1|82-95|benzodiazepine|drug
...
```

Outline

Sequence tagging: the B-I-O approach

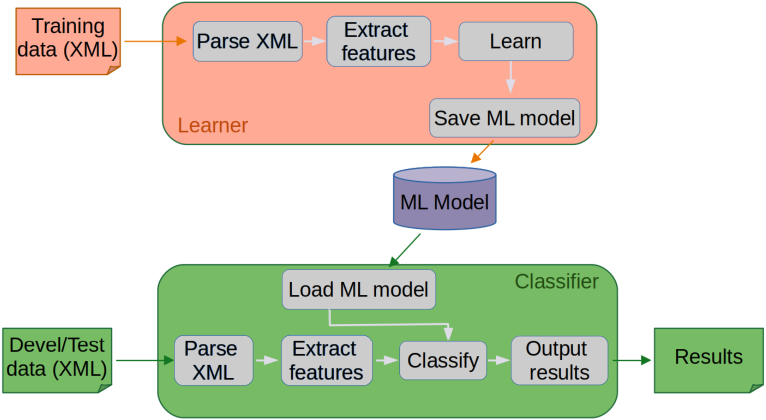
- We want to detect *subsequences* in a sentence (e.g. drug names).
- To approach this as a ML classification problem, we will classify each token.
- The classes predicted by the classifier must allow the later reconstruction of the target subsequences.
- B-I-O schema: mark each token as **B**egin of a subsequence, **I**nside a subsequence, or **O**utside any subsequence.
- If we not only want to recognize the subsequences, but also classify them, we use more informative B-I-O classes:

Ascorbic	**acid**	**,**	**aspirin**	**,**	**and**	**the**	**common**	**cold**	**.**
B-drug	I-drug	O	B-brand	O	O	O	O	O	O

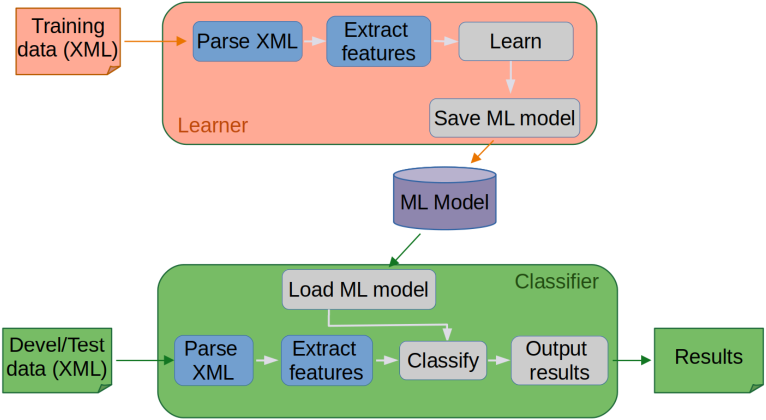
- Different variations of this schema exist: BIO, BIOS, BIOES (aka BILOU)

Outline

General Structure

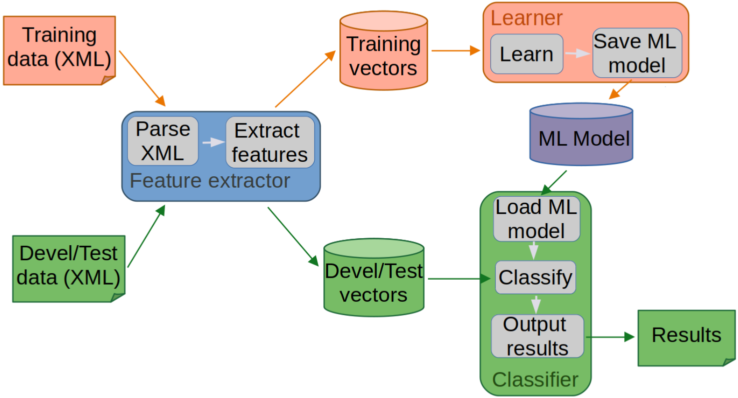


General Structure



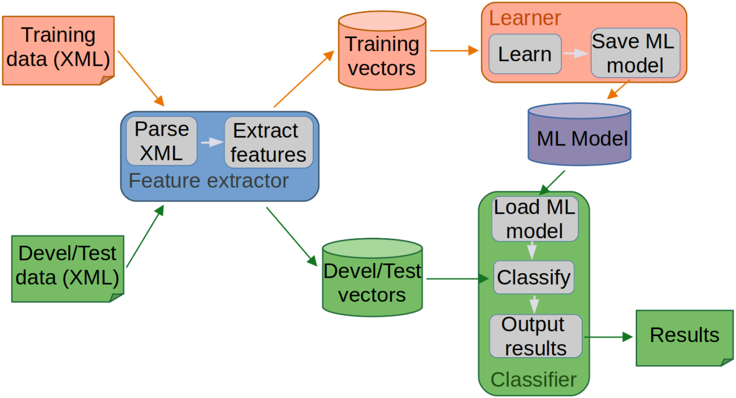
Extracting features is a costly operation, which we do not want to repeat for every possible experiment or algorithm parametrization.

General Structure



Feature extraction process is performed once, out of learning or predicting processes.

General Structure



Feature extraction process is performed once, out of learning or predicting processes.
Thus, we need to write not a single program, but three different components: feature extractor, learner, and classifier.

Outline

Outline

Feature Extractor

The feature extractor:

- Independent program, separated from learner and classifier
- Receives as argument the directory with the XML files to encode.
- Prints the feature vectors to `stdout`

```
$ python3 ./extractor_features.py data/devel devel.feat
$ more devel.feat
DDI-DrugBank.d658.s0 When 0 3 O form=When formlower=when suf3=hen
suf4=When isTitle BoS formNext=administered
formlowerNext=administered suf3Next=red suf4Next=ered
DDI-DrugBank.d658.s0 administered 5 16 O form=administered
formlower=administered suf3=red suf4=ered formPrev=When
formlowerPrev=when suf3Prev=hen suf4Prev=When isTitlePrev
formNext=concurrently formlowerNext=concurrently suf3Next=tly
suf4Next=ntly
...
```

Feature Extractor

```
# create analyzer. We don't need the parser now (it's faster without)
nlp = spacy.load("en_core_web_trf", disable=["parser"])
# parse XML file, obtaining a DOM tree
tree = parse(datafile)  # process each sentence in the file
sentences = tree.getElementsByTagName("sentence")
for s in sentences :
  sid = s.attributes["id"].value   # get sentence id
  spans = []
  stext = s.attributes["text"].value   # get sentence text
  entities = s.getElementsByTagName("entity") # get gold standard
 entities
  for e in entities :
     # for discontinuous entities, we only get the first span
     # (will not work, but there are few of them)
     (start,end) = e.attributes["charOffset"].value.split(";")[0].split(
 "-")
     typ =  e.attributes["type"].value
     spans.append((int(start),int(end),typ))
  tokens = nlp(stext)    # convert the sentence to a list of tokens
  features = extract_sentence_features(tokens)  # extract features
  # print features in format expected by CRF/SVM/MEM trainers
  for i,tk in enumerate(tokens) :
     # see if the token is part of an entity
     tks,tke = tk.idx, tk.idx+len(tk.text)
     # get gold standard tag for this token
     tag = get_label(tks, tke, spans)
     # print feature vector for this token
     print (sid, tk.text, tks, tke-1, tag, "\t".join(features[i]), sep='
 \t', file=outf)
  # blank line to separate sentences
  print(file=outf)
```

Feature extraction for NLP

- In most ML applications, the feature space is finite and known (e.g. credit scoring, medical diagnose prediction, churn prevention, fraud detection, etc).
- Also, most of the used features are numerical or categorial (income, age, sex, colestherol level, number of receipts returned, etc.)
- Thus, in these ML applications, feature vectors are usually *exhaustive* lists of pairs feature-value.

Feature extraction for NLP

- In most ML applications, the feature space is finite and known (e.g. credit scoring, medical diagnose prediction, churn prevention, fraud detection, etc).
- Also, most of the used features are numerical or categorial (income, age, sex, colestherol level, number of receipts returned, etc.)
- Thus, in these ML applications, feature vectors are usually *exhaustive* lists of pairs feature-value.

BUT...

Feature extraction for NLP

- In most ML applications, the feature space is finite and known (e.g. credit scoring, medical diagnose prediction, churn prevention, fraud detection, etc).
- Also, most of the used features are numerical or categorial (income, age, sex, colestherol level, number of receipts returned, etc.)
- Thus, in these ML applications, feature vectors are usually *exhaustive* lists of pairs feature-value.

BUT...

- In most NLP applications, features are related to appearing words, suffixes, prefixes, lemmas, etc. Thus, the feature space is huge.
- Moreover, features are usually binary-valued (a word appears or not, a suffix appears or not, etc).
- Thus, in NLP applications, feature vectors are usually *intensive* lists of strings (i.e. listing the names for features with value `true`, and ommiting all the rest), and are stored as *sparse vectors*.

Sparse vector representation: CSR matrix

- NLP applications usually have feature spaces of hundreds of thousands of dimensions, which can not be stored in a table.
- However, vectors are usually very sparse: Each example (each token in NERC case) has only a reduced number of non-zero columns
- CSR matrix representation consists of storing the coordinates and the value for non-zero elements in the matrix.
- The matrix elements are indexed by their (row,column) coordinates.

$$\begin{pmatrix} 0 & 0 & 3 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 6 \end{pmatrix} \Longrightarrow \begin{pmatrix} \mathit{row} & \mathit{col} & \mathit{val} \\ 0 & 2 & 3 \\ 1 & 0 & 2 \\ 3 & 2 & 1 \\ 3 & 4 & 6 \end{pmatrix}$$

- Each row in the matrix corresponds to a training example (a sentence token, for NERC).
- Each column corresponds to a boolean feature (a distinct string among all those instantiated by the training data).
- A feature index is needed to keep the correspondence between feature names and their column number.

Feature Extractor Functions - Extract features

```
def extract_features(s) :
  '''
  Task:
    Given a tokenized sentence, return a feature vector for each token

  Input:
    s: A tokenized sentence (list of triples (word, offsetFrom, offsetTo) )

  Output:
    A list of feature vectors, one per token.
    Features are binary and vectors are in sparse representation (i.e. only
     active features are listed)

  Example:
    >>> extract_features([("Ascorbic",0,7), ("acid",9,12), (",",13,13),
          ("aspirin",15,21), (",",22,22), ("and",24,26), ("the",28,30),
          ("common",32,37), ("cold",39,42), (".",43,43)])
    [ [ "form=Ascorbic",  "suf4=rbic", "next=acid", "prev=_BoS_", "
     capitalized"  ],
      [ "form=acid",  "suf4=acid", "next=,", "prev=Ascorbic" ],
      [ "form=,",  "suf4=,", "next=aspirin", "prev=acid", "punct" ],
      [ "form=aspirin",  "suf4=irin", "next=,", "prev=," ],
      ...
    ]
  '''
```

Outline

Learner

The B-I-O approach consists of a classifier that assigns a label to each token, so any ML algorithm may be used

- Local classifiers: Each token is classified independently.
  - Maximum Entropy Classifiers (MEM)
  - Support Vector Machines (SVM)
  - ...

  May lead to inconsistencies (e.g `III` sequences without a `B` at the beggining)
- Global Classifiers: The whole sequence is taken into account for a global decision. Features are factorized to get an affordable cost.
  - Maximum Entropy Markov Models (MEMM)
  - Conditional Random Fields (CRF)
  - ...

Learner: MEM (a.k.a. Logistic Regression)

- Install and import `scikit-learn`
  `$ pip install scikit-learn`
- Use provided `train.py` to learn a MEM model.
  `$ python3 run.py train MEM`

  You may modifiy learner parameters (loss function, thresholds, learning rates, etc).
- Check performance on development data.
  `$ python3 run.py predict MEM`
- `scikit-learn` implementation of `LogisticRegression` (MEM), admits sparse matrix representations such as CSR (Compressed Sparse Row) matrix.

Learner: SVM

- Install and import `scikit-learn`
  `$ pip install scikit-learn`
- Use provided `train.py` to learn a SVM model.
  `$ python3 run.py train SVM`
  You may modifiy learner parameters (loss function, thresholds, learning rates, etc).
- Check performance on development data.
  `$ python3 run.py predict SVM`
- `scikit-learn` implementation of `SVC` (SVM), admits sparse matrix representations such as CSR (Compressed Sparse Row) matrix.

Learner: CRF

- Install and import `pycrfsuite`
  `$ pip install python-crfsuite`
- Use provided `train.py` to learn a CRF model.
  `$ python3 run.py train CRF`
  You may modifiy learner parameters (loss function, thresholds, learning rates, etc).
- Check performance on development data.
  `$ python3 run.py predict CRF`
- `python-crfsuite` admits sparse vectors of boolean features represented as lists of names of the columns (i.e. features) that are true for each example.

Learner: Others

- You are welcome to try and compare any other ML algorithm of your choice available in scikit-learn. The chosen algorithm must support sparse vectors of binary features (e.g. scipy `csr_matrix`). DO NOT use neural network approaches, we'll do that later in the course.
- Create a new class similar to `MEM.py`, `SVM.py` or `CRF.py` with:
  - A constructor that allows either loading an exisiting model or initializing one for training
  - A `train` method that receives a dataset and trains a model
  - A `predict` method that receives a classification example and returns its class.
  - Adapt `train.py` and `predict.py` to use the new class depending on the provided file extension
- Use `train.py` to learn your new XYZ model.

```
$ python3 train.py train.feat model.xyz
```

Outline

Classifier - All options

You can apply the learned models to new data:

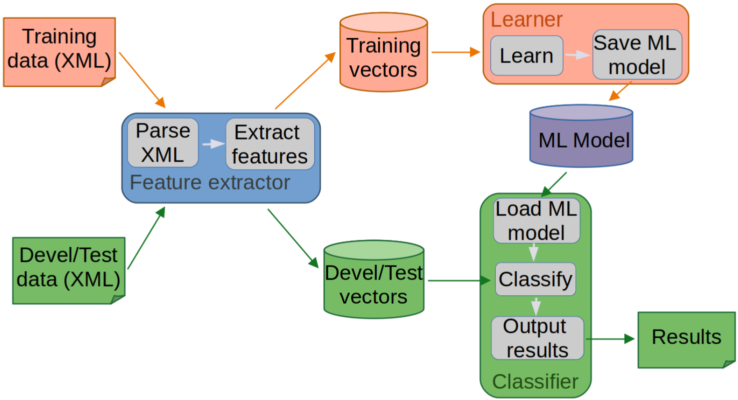
```
$ python3 run.py extract test
$ python3 run.py predict MEM test
$ python3 run.py predict SVM test
$ python3 run.py predict CRF test
```

Once a best learner is selected experimenting with different algorithms and parameters, the best model can be applied to test data to evaluate generalization ability of the model.

Outline

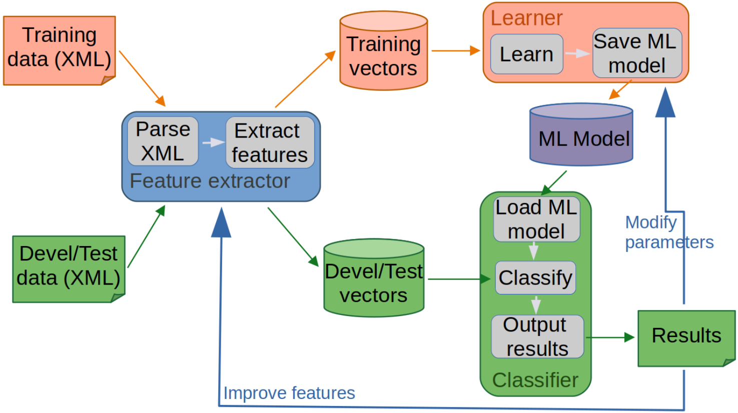
Build a good ML-based drug NERC

Strategy to follow:



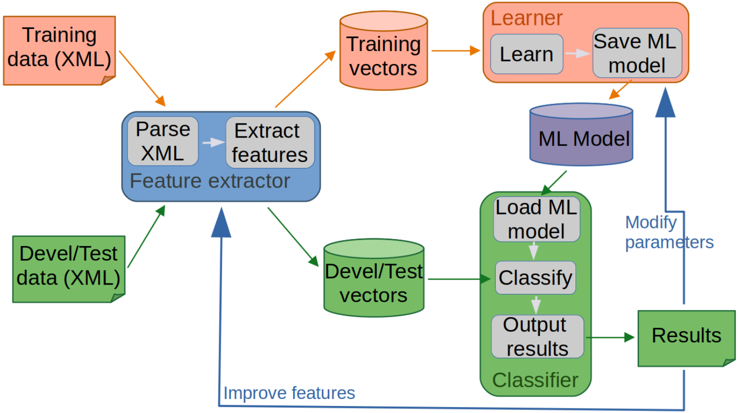
Build a good ML-based drug NERC

Strategy to follow:



Build a good ML-based drug NERC

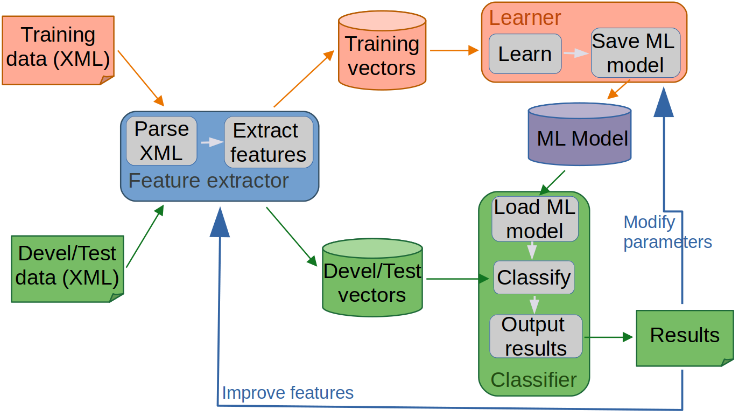
Strategy to follow:



- Repeat training – evaluation cycle on devel dataset to find out which is the best parameterization for the used algorithm.

Build a good ML-based drug NERC

Strategy to follow:



- Repeat training – evaluation cycle on devel dataset to find out which is the best parameterization for the used algorithm.
- Repeat feature extraction – training – evaluation cycle on devel dataset to find out which features are useful.

Choosing useful features

- Used models are *token* classifiers, so there is a feature vector *per token*.
- Features about a token should allow its classification, so they should encode information about *both* the token itself and its context (i.e. nearby words).
- Feature names must be *unambiguous*. E.g., a feature named `sufx=azole` may not be enough if one wants to encode also context word suffixes. In that case, different feature names are needed (e.g.: `sufx=azole` for the focus word, plus e.g. `sufx-2=azole`, `sufx-2=azole`, `sufx+1=azole`, `sufx+2=azole` for nearby words).
- Including features encoding information from external dictionaries will largely improve results.

Choosing useful features

- IMPORTANT: Feature names such as `sufx=azole` are not key-value pairs (i.e. not a `sufx` feature with value `azole`), but just a string naming a binary (true/false) feature.
  The feature name could be any (`sufxisazole`, `wordendsinazole`, ...) as long as it is active (i.e. present in the sparse vector) only when that property holds.
- IMPORTANT: Features are boolean, and only those with value `true` are listed. So, a feature not present in the list is an *existing* column with value `false`.

  Thus, it does not make sense to have e.g. a boolean feature `capitalized:true` and another feature `capitalized:false`, since the absence of the former is equivalent to the presence of the latter and viceversa.

Outline

Exercise Goals

What you should do:

- Work on your feature extractor. It is the component of the process where you have most control.
- Experiment with different algorithms and parameterizations.
- Keep track of tried features and parameter combinations, and results produced by each.

What you should **NOT** do:

- Use neural network learners. We'll do that later on the course.
- Produce an overfitted model: If performance on the test dataset is much lower than on devel dataset, you probably are overfitting your model.

Exercise Goals

Orientative results:

- Provided initial version achieves over 45% macro average F1 on devel with 2 simple feature templates (word form and size-3 suffix).
- A set of 10 feature templates is enough to get a macroaverage F1 over 65%. Used information includes (for current, previous, and next tokens)
  - word forms, original and lowercase
  - suffixes (of different lenghts)
  - capitalization pattern (all upper, title, camelcase,...)
  - presence of numbers, dashes, etc. in the token
  - existence (and class) of the token in external lists
  - ...

Deliverables

After DDI-ML task, you will need to deliver a single report on the work carried out on both (NERC-ML and DDI-ML) tasks.

So, during the development and experimentation on NERC task:

- keep track of tried/discarded features
- keep track of used algorithms and parameterizations.
- Record obtained results in the different experiments, and compile the information you'll later need to elaborate the report.