Mining Unstructured Data

Outline

Assignment

Write a python program that parses all XML files in the folder given as argument and recognizes and classifies drug names. The program must use 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

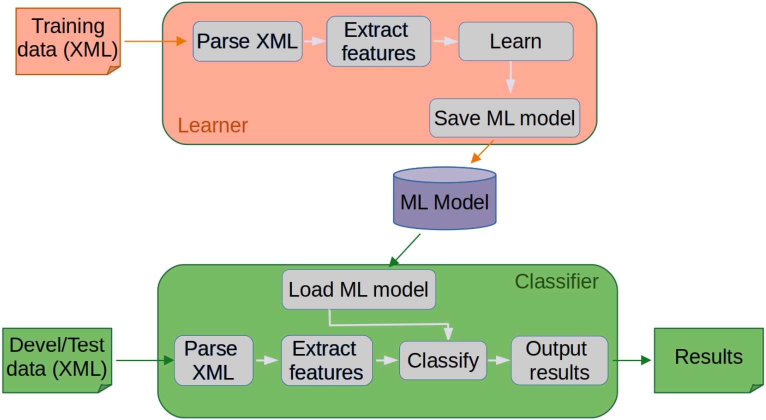
- We want to detect *subsequences* in a sentence (e.g. drug names).
- To approach this as a ML classification problem, we will classify each token.
- The classes predicted by the classifier must allow the later reconstruction of the target subsequences.
- B-I-O schema: mark each token as **B**egin of 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

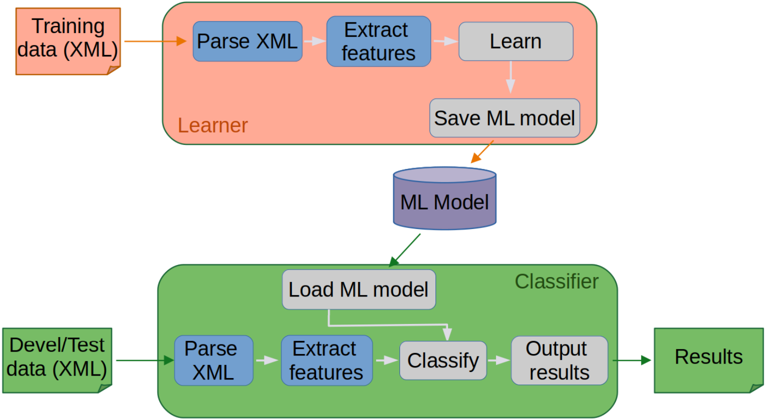
- Different variations of this schema exist: BIO, BIOS, BIOES (aka BILOU)

Outline

General Structure

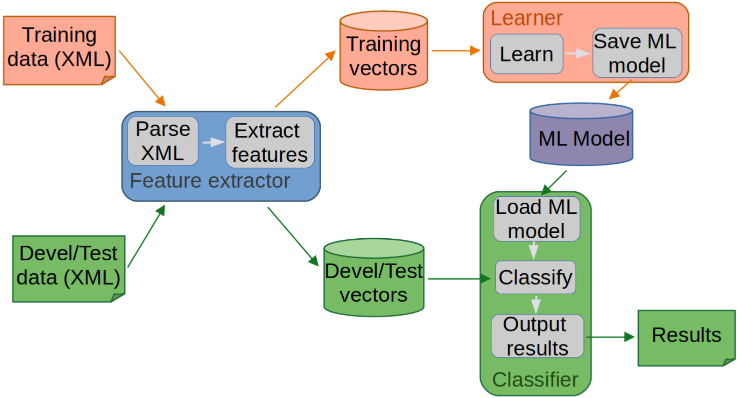


General Structure



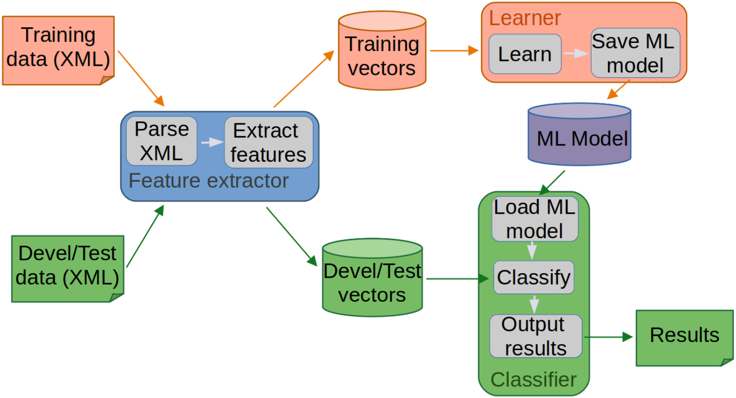
Extracting features is a costly operation, which we do not want to repeat for every possible experiment or algorithm parametrization.

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 ./feature-extractor.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

```
# process each file in directory
for f in listdir(datadir) :
    # parse XML file, obtaining a DOM tree
    tree = parse(datadir + "/" + f)
    # process each sentence in the file
    sentences = tree.getElementsByTagName("sentence")
    for s in sentences :
        sid = s.attributes["id"].value  # get sentence id
        stext = s.attributes["text"].value  # get sentence text
        # load ground truth entities.
        gold=[]
        entities = s.getElementsByTagName("entity")
        for e in entities :
            # for discontinuous entities, we only get the first span
            offset = e.attributes["charOffset"].value
            (start,end) = offset.split(";")[0].split("-")
            gold.append((int(start), int(end), e.attributes["type"].value))

        # tokenize text
        tokens = tokenize(stext)
        # extract features for each word in the sentence
        features = extract_features(tokens)
        # print features in format suitable for the learner/classifier
        for i in range (0,len(tokens)) :
            # see if the token is part of an entity, and which part (B/I)
            tag = get_tag(tokens[i], gold)
            print (sid, tokens[i][0], tokens[i][1], tokens[i][2],
                   tag, "\t".join(features[i]), sep='\t')
        # blank line to separate sentences
        print()
```

Feature Extractor Functions - Tokenize text


```
def tokenize(s) :
'''
Task:
  Given a sentence, calls nltk.tokenize to split it in
  tokens, and adds to each token its start/end offset
  in the original sentence.

Input:
  s: string containing the text for one sentence

Output:
  Returns a list of tuples (word, offsetFrom, offsetTo)
'''
```

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*.

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=," ],
      ...
    ]
  '''
```

Feature Extractor Functions - Ground truth tag

```
def get_tag(token, gold) :
  '''
  Task:
    Given a token and a list of ground truth entites in a sentence, decide
     which is the B-I-O tag for the token

  Input:
    token: A token, i.e. one triple (word, offsetFrom, offsetTo)
    gold: A list of ground truth entities, i.e. a list of triples (
     offsetFrom, offsetTo, type)

  Output:
    The B-I-O ground truth tag for the given token ("B-drug", "I-drug", "B-
     group", "I-group", "O", ...)

  Example:
    >>> get_tag(("Ascorbic",0,7), [(0, 12, "drug"), (15, 21, "brand")])
    B-drug
    >>> get_tag(("acid",9,12), [(0, 12, "drug"), (15, 21, "brand")])
    I-drug
    >>> get_tag(("common",32,37), [(0, 12, "drug"), (15, 21, "brand")])
    O
    >>> get_tag(("aspirin",15,21), [(0, 12, "drug"), (15, 21, "brand")])
    B-brand
  '''
```

Outline

Learner - Option 1: CRF

- Install and import `pycrfsuite`
  `$ pip install python-crfsuite`
- Use provided train-crf.py to learn a model.
  `$ python3 train-crf.py model.crf < train.feat`
  You may modifiy learner parameters (loss function, thresholds, learning rates, etc).

Learner - Option 2: Naive Bayes

- Install and import `skcit-learn`

  `$ pip install sckit-learn`

- Use provided train-sklearn.py to learn a model.

  `$ python3 train-sklearn.py model.joblib vectorizer.joblib < train.clf.feat`

Learner - Option 3: Your choice

- Select a ML algorithm of your choice (DT, SVM, RF, ...) and a python library implementing it. You can use the Naive Bayes code as template to train other Sckit-learn algorithms.
- Adapt the feature file format to the needs of the selected algorithm.
- Train a classification model for the task of predicting B-I-O tags for each token.
- Create a module `XXX.py` with a constructor and a `predict` method, following the structure of `CRF.py` and `sklearn.py`. Add your new classifier to the constructor in `ML_model.py`
- DO NOT use neural network approaches, we'll do that later in the course.

Outline

Classifier - All options

You can apply the learned models to new data:

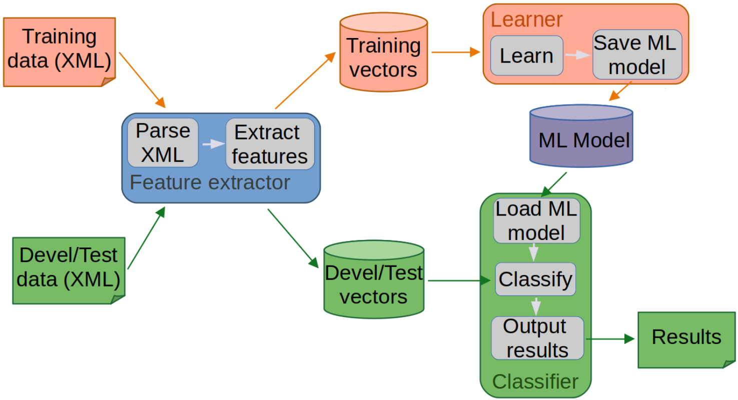
```
$ python3 extract-features.py data/devel > devel.feat
$ python3 predict.py model.mem <devel.feat >devel.out
$ python3 evaluator.py data/devel devel.out

$ python3 extract-features.py data/test > test.feat
$ python3 predict-sklearn.py model.joblib vectorizer.joblib
<test.feat >test.out
$ python3 evaluator.py data/test test.out
```

Outline

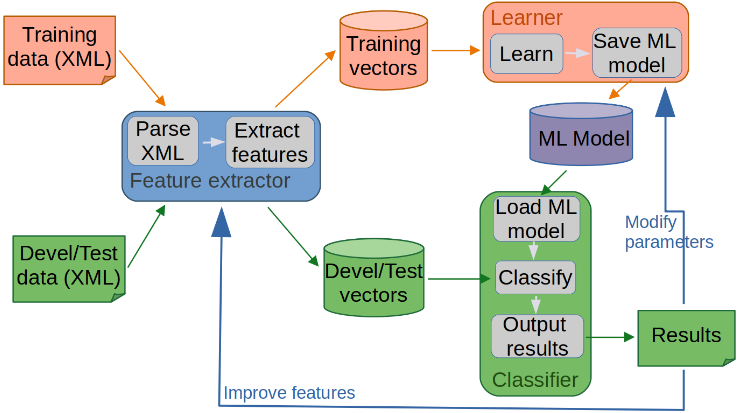
Build a good ML-based drug NERC

Strategy to follow:



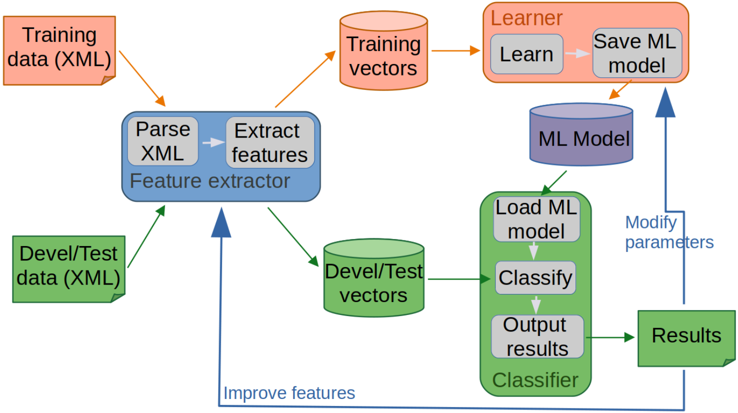
Build a good ML-based drug NERC

Strategy to follow:



Build a good ML-based drug NERC

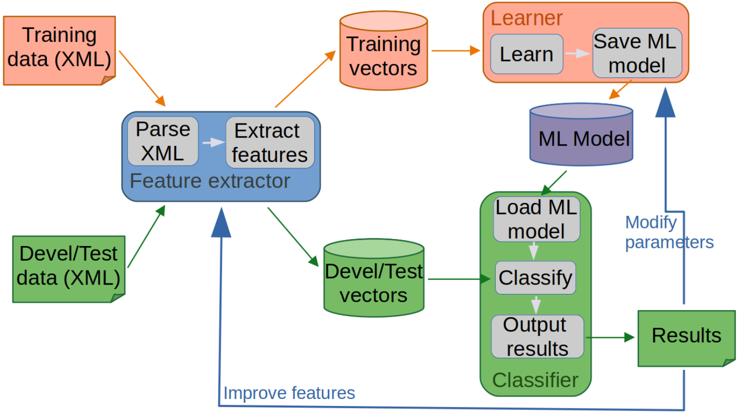
Strategy to follow:



- 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).
- As in the rule-based approach, including features encoding information from external dictionaries will largely improve results.

Choosing useful features

- REMEMBER: 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.

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 parameterizations of the chosen learner. You may try different learning algorithms if you feel up to. Note that the same feature vectors can be fed to different learners (maybe with some format adaptation).
- 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.
- Alter the provided code structure.

Exercise Goals

Orientative results:

- Provided initial version achieves 55%(CRF) and 49%(Naive Bayes) macro average F1 on devel with 2 simple feature templates.
- A set of 10 feature templates is enough to get a macroaverage F1 about 63%(CRF). Used information includes (for current, previous, and next tokens)
  - word forms, original and lowercase
  - suffixes (of different lengths)
  - capitalization pattern (all upper, title, camelcase,...)
  - presence of numbers, dashes, etc
  - ...
- Adding features to look up in external resources (provided in the lab project zipfile) rises macroaverage F1 to about 75% (CRF).

Deliverables

Write a report describing the work carried out on NERC tasks.
The report must be a **single self-contained PDF document**, under 10 pages, containing:

- *Introduction:* What is this report about. What is the goal of the presented work.

Deliverables (continued)

- *Machine learning NERC*
  - *Selected algorithm:* Which classifier/s did you select or try. Reasons of the choice. Comparison if you tried more than one.
  - *Feature extraction:* Tried/discarded/used features. Impact of different feature combinations.
  - *Code:* Include your `extract_features` function (and any other function it may call), properly formatted and commented. **Do not include any other code.**
  - *Experiments and results:* Results obtained on the **devel** and **test** datasets, for different algorithms, feature combinations, parameterizations you deem relevant.
- *Conclusions:* Final remarks and insights gained in this task.

Keep result tables in your report in the format produced by the evaluator module. Do not reorganize/summarize/reformat the tables or their content.