Machine learning in NLP
Additional information about Assignment 3

UNIVERSITY OF GOTHENBURG

Språk-
BANKEN

Richard Johansson

October 16, 2014

assignment 3: overview

- implement the perceptron algorithm for structured objects
- use this to build a dependency parser and a named entity tagger

refresher: typical use of scikit-learn classifiers (training)

- get the training data (e.g. read it from a file)
  - we then have a list of **inputs** and another list of corresponding **outputs**
- **extract features** from each input object
  - e.g. a dict with attributes and their values
- convert the features to vectors using one of the **vectorizers**
  - `vec.fit(X)` to create the feature→dimension mapping
  - `Xe = vec.transform(X)` to convert the training set
  - (`fit_transform` carries out both steps)
- **initialize** the classifier object
- **train** the classifier
  - call `classifier.fit(Xe, Y)`
- (optional: wrap the vectorizer and the classifier into a `Pipeline`)

refresher: typical use of scikit-learn classifiers (new data)

- read the data
- **extract features** from each input object
- convert the features to vectors using the previous **vectorizer**
  - again `Xe = transform(X)`
- for each input, **predict** the output using the classifier
  - `guess = classifier.predict(x)`
- output or evaluate the result

so how does this relate to parsing and tagging?

- when you build the parser or tagger in this assignment, the code will have a quite similar structure
- . . . but the scikit-learn components are replaced with specialized components
  - e.g. a `ParseVectorizer` tailored for sentences/parses, as opposed to the general-purpose `DictVectorizer`
- we now consider the parser

training the parser

- read the training treebank
  - use the function `read_dependency_treebank`
  - we then have a list of inputs (sentences) and another list of corresponding outputs (parse trees)
- convert the sentences to vectors using the `ParseVectorizer`
  - `vec.fit(X, Y)` to create the feature→dimension mapping
  - `Xe, Ye = vec.transform(X, Y)` to convert the training set
  - this step includes feature extraction
- **initialize** the "classifier" (the parse predictor)
  - we give it a **problem definition** object that contains parsing-specific functionality
- **train** the parse predictor
  - call `parser.fit(Xe, Ye)`

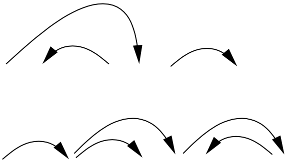
after reading the training set

X

<D> Lisa walks home

<D> Put it in the car

Y



the actual Python objects

X

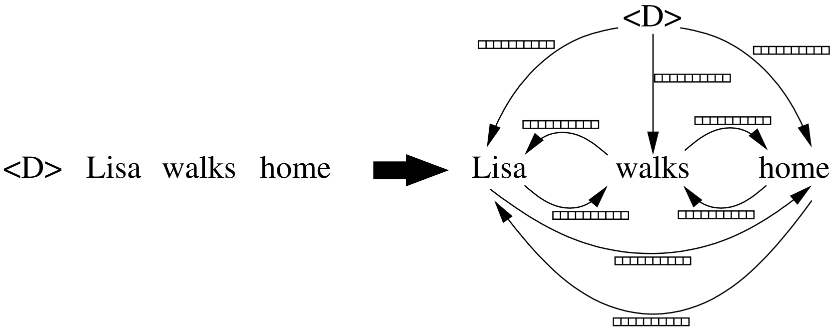
```
[ [ ('<TOP>', '<TOP>'), ('Lisa', 'NNP'),
    ('walks', 'VBZ'), ('home', 'RB') ],
  [ ('<TOP>', '<TOP>'), ('Put', 'VB'),
    ('it', 'PRP'), ('in', 'IN'),
    ('the', 'DT'), ('car', 'NN') ] ]
```

Y

```
[ [ -1, 2, 0, 2 ],

  [ -1, 0, 1, 1, 5, 3 ] ]
```

what happens in transform in the ParseVectorizer?



pseudocode for training the perceptron

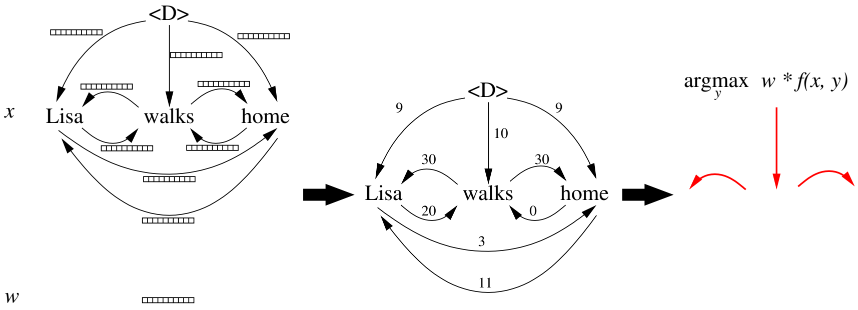
$\boldsymbol{w} = (0, \ldots, 0)$
repeat N times
  for (x, y) in $\mathcal{T}$
    $g = \arg\max_{y'} \boldsymbol{w} \cdot \boldsymbol{f}(x, y')$ ← find the highest-scoring tree
    if g is not equal to y
      $\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{f}(x, y) - \boldsymbol{f}(x, g)$ ← add the features for the good tree and subtract those for the bad tree
return w

- the two highlighted parts are specific to the problem we're considering, in this case parsing
- this is why we wrap them into a "problem definition" object
- now, let's look at the implementation of those two methods

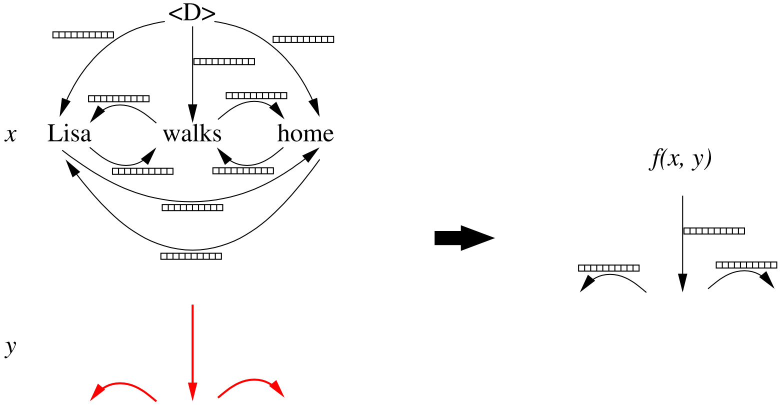
the methods in the problem definition

- `predict`: if we have a weight vector w, find the highest-scoring parse tree for the sentence x
- `get_features`: find the feature vector $f(x, y)$ for a sentence x and a tree y

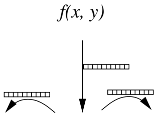
predict in MSTParsingDefinition



get_features in MSTParsingDefinition



a technical note on the feature vectors



- see the code or McDonald's paper for a description of what features we're using
- the features are stored in sparse vectors
- for practical reasons, `get_features` returns a sparse **matrix**, that is a "list" of vectors
  - use the helper function `add_sparse_rows_to_dense` to add all edge vectors to $\boldsymbol{w}$

running the parser on new data

- read the testing treebank
- convert the sentences to vectors using your previous `ParseVectorizer`
  - `Xe, Ye = vec.transform(X, Y)` to convert the training set
- for each input, predict the output using the parser you trained previously
  - `guess = parser.predict(x)`
- output or evaluate the result
  - in this assignment: compute the attachment accuracy

attachment evaluation: example

gold-standard trees

```
[ [ -1, 2, 0, 2 ],
  [ -1, 0, 1, 1, 5, 3 ] ]
```

predicted trees

```
[ [ -1, 2, 0, 2 ],
  [ -1, 0, 1, 2, 5, 3 ] ]
```

$$\text{attachment accuracy} = \frac{\text{number of correct attachments}}{\text{number of tokens}} = \frac{7}{8}$$

- remember not to count the dummy root token!