Machine learning in NLP
Introduction to the scikit-learn library

Richard Johansson

September 4, 2014
what I will say today

- go through the steps of the function tagging program
- the basics of the scikit-learn library
overview of the function tagging program

1. splits data into training, development and test parts
   ▶ using scikit-learn’s train_test_split
overview of the function tagging program

1. splits data into training, development and test parts
   - using scikit-learn’s `train_test_split`
2. goes through all phrases in the training trees and extracts examples from them
   - for each example, the features are stored in the list $X$ and the corresponding true outputs in $Y$
overview of the function tagging program

1. splits data into training, development and test parts
   ▶ using scikit-learn’s train_test_split
2. goes through all phrases in the training trees and extracts examples from them
   ▶ for each example, the features are stored in the list $X$ and the corresponding true outputs in $Y$
3. trains a classifier using the examples we have collected
   ▶ details about that later
overview of the function tagging program

1. splits data into training, development and test parts
   ▶ using scikit-learn’s `train_test_split`
2. goes through all phrases in the training trees and extracts examples from them
   ▶ for each example, the features are stored in the list $X$ and the corresponding true outputs in $Y$
3. trains a classifier using the examples we have collected
   ▶ details about that later
4. reads the phrases from the dev (or test) set and extracts features from them
   ▶ in the same way as we did for the training set
overview of the function tagging program

1. splits data into training, development and test parts
   ▶ using scikit-learn’s train_test_split
2. goes through all phrases in the training trees and extracts examples from them
   ▶ for each example, the features are stored in the list $X$ and the corresponding true outputs in $Y$
3. trains a classifier using the examples we have collected
   ▶ details about that later
4. reads the phrases from the dev (or test) set and extracts features from them
   ▶ in the same way as we did for the training set
5. applies the classifier to all the examples from the test set
overview of the function tagging program

1. splits data into training, development and test parts
   ▶ using scikit-learn’s train_test_split
2. goes through all phrases in the training trees and extracts examples from them
   ▶ for each example, the features are stored in the list $X$ and the corresponding true outputs in $Y$
3. trains a classifier using the examples we have collected
   ▶ details about that later
4. reads the phrases from the dev (or test) set and extracts features from them
   ▶ in the same way as we did for the training set
5. applies the classifier to all the examples from the test set
6. evaluates and prints the statistics
the function \texttt{train\_scikit\_classifier}

```python
def train_scikit_classifier(X, Y):
    # A DictVectorizer maps a feature dict to a sparse vector,
    # e.g. vec.transform({'label': 'NP'}) might give
    # [0, 0, ..., 0, 1, 0, ...]
    vec = DictVectorizer()

    # Convert all the feature dicts to vectors.
    # As usual, it’s more efficient to handle all at once.
    Xe = vec.fit_transform(X)

    # Initialize the learning algorithm we will use.
    classifier = Perceptron(n_iter=20)

    # Finally, we can train the classifier.
    classifier.fit(Xe, Y)

    # Return a pipeline consisting of the vectorizer followed
    # by the classifier.
    return Pipeline([('vec', vec), ('classifier', classifier)])
```
what are X and Y?

- our feature extractor has collected features in the form of attributes with values
  - e.g. { 'label':'NP', ... }
  - they are stored in the list X
- the corresponding true outputs are stored in the list Y

\[
\begin{align*}
X & = \left\{ \begin{array}{l}
\{ 'label':'NP', ... \} \\
\{ 'label':'PP', ... \} \\
\{ 'label':'S', ... \} \\
\vdots \\
\{ 'label':'PP', ... \}
\end{array} \right\} \\
Y & = \begin{bmatrix}
\text{SBJ} \\
\text{ADV} \\
\text{(empty)} \\
\vdots \\
\text{TMP}
\end{bmatrix}
\end{align*}
\]
The first step: mapping features to numerical vectors

- scikit-learn’s learning methods work with features as numbers, not strings
- they can’t directly use the feature dicts we have stored in $X$
- converting from string to numbers is the purpose of these lines:

```python
vec = DictVectorizer()
Xe = vec.fit_transform(X)
```

\[
\begin{bmatrix}
\{ 'label': 'NP', \ldots \} \\
\{ 'label': 'PP', \ldots \} \\
\{ 'label': 'S', \ldots \} \\
\ldots \\
\ldots \\
\ldots \\
\{ 'label': 'PP', \ldots \}
\end{bmatrix}
\rightarrow
\begin{bmatrix}
1 & 0 & 0 & 0 & \ldots \\
0 & 1 & 0 & 0 & \ldots \\
0 & 0 & 1 & 0 & \ldots \\
\ldots \\
\ldots \\
\ldots \\
0 & 1 & 0 & 0 & \ldots 
\end{bmatrix}
\]
types of vectorizers

▶ a DictVectorizer converts from attribute–value dicts:

\[
X = \begin{bmatrix}
{ 'label': 'NP', ... } \\
{ 'label': 'PP', ... } \\
{ 'label': 'S', ... } \\
... \\
... \\
... \\
{ 'label': 'PP', ... }
\end{bmatrix}
\]

\[
X_e = \begin{bmatrix}
1 & 0 & 0 & 0 & ... \\
0 & 1 & 0 & 0 & ... \\
0 & 0 & 1 & 0 & ... \\
... \\
... \\
... \\
0 & 1 & 0 & 0 & ...
\end{bmatrix}
\]

▶ a CountVectorizer converts from texts (after applying a tokenizer) or lists:

\[
X = \begin{bmatrix}
"this is a text"
"here is another text"
... \\
... \\
... \\
"a cat on a mat"
\end{bmatrix}
\]

\[
X_e = \begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & ... \\
0 & 1 & 0 & 1 & 1 & 1 & ... \\
... \\
... \\
... \\
0 & 0 & 2 & 0 & 0 & 0 & ...
\end{bmatrix}
\]

▶ a TfidfVectorizer is like a CountVectorizer, but also uses \text{TF*IDF}
what goes on in a DictVectorizer?

- each feature corresponds to one or more columns in the output matrix
- easy case: boolean and numerical features:
  
  \[
  X = \begin{bmatrix}
  \{'f1': False, 'f2': 7\} \\
  \{'f1': True, 'f2': 2\} \\
  \{'f1': False, 'f2': 9\}
  \end{bmatrix}
  \]

  \[
  X_e = \begin{bmatrix}
  0 & 7 \\
  1 & 2 \\
  0 & 9
  \end{bmatrix}
  \]

- for string features, we reserve one column for each possible value
  
  - that is, we convert to booleans

  \[
  X = \begin{bmatrix}
  \{'f1': 'NP', 'f2': 'in'\} \\
  \{'f1': 'NP', 'f2': 'on'\} \\
  \{'f1': 'VP', 'f2': 'in'\}
  \end{bmatrix}
  \]

  \[
  X_e = \begin{bmatrix}
  1 & 0 & 1 & 0 \\
  1 & 0 & 0 & 1 \\
  0 & 1 & 1 & 0
  \end{bmatrix}
  \]
code example (DictVectorizer)

▶ here's an example:

```python
from sklearn.feature_extraction import DictVectorizer
X = [{'f1': 'NP', 'f2': 'in', 'f3': False, 'f4': 7},
     {'f1': 'NP', 'f2': 'on', 'f3': True, 'f4': 2},
     {'f1': 'VP', 'f2': 'in', 'f3': False, 'f4': 9}]
vec = DictVectorizer()
Xe = vec.fit_transform(X)
print(Xe.toarray())

print(vec.vocabulary_)
```

▶ the result:

```
[[ 1.  0.  1.  0.  0.  7.]
 [ 1.  0.  0.  1.  1.  2.]
 [ 0.  1.  1.  0.  0.  9.]]

{’f4’: 5, ’f2=’in’: 2, ’f1=’NP’: 0, ’f1=’VP’: 1, ’f2=’on’: 3, ’f3’: 4}
```
CountVectorizer for document representation

- a CountVectorizer converts from documents
  - the document is a string or a list of tokens
- just like string features in a DictVectorizer, each word type will correspond to one column

\[
\begin{align*}
X & \\
\begin{bmatrix}
\text{"example text"} \\
\text{"another text"}
\end{bmatrix} & \rightarrow
\begin{bmatrix}
0 & 1 & 1 \\
1 & 0 & 1
\end{bmatrix} \\
\text{another} & \text{example}
\end{align*}
\]
code example (CountVectorizer)

▶ here’s an example:

```python
X = ['example text', 'another text']

vec = CountVectorizer()
Xe = vec.fit_transform(X)
print(Xe.toarray())

print(vec.vocabulary_)
```

▶ the result:

```
[[0 1 1]
 [1 0 1]]

{'text': 2, 'example': 1, 'another': 0}
```
a comment about the vectorizer methods

- **fit**: look at the data, create the mapping
- **transform**: convert the data to numbers
- **fit_transform** = **fit** + **transform**
training a classifier

▶ after mapping the features to numbers with our Vectorizers, we can train a perceptron classifier:

```python
classifier = Perceptron(n_iter=20)
classifier.fit(Xe, Y)
```

▶ other classifiers (e.g. Naive Bayes) can be trained in a similar way:

```python
classifier = MultinomialNB()
classifier.fit(Xe, Y)
```
applying the classifier to new examples

X_new = ... # extract the features for new examples

Xe_new = vectorizer.transform(X_new)

guesses = classifier.predict(Xe_new)
combining a vectorizer and a classifier into a pipeline

vectorizer = ...
Xe = vectorizer.fit_transform(X)
classifier = ...
classifier.fit(Xe, Y)

pipeline = Pipeline([('vec', vectorizer), ('cls', classifier)])

X_new = ... # extract the features for new examples
guesses = pipeline.predict(X_new)
a note on efficiency

- Python is a nice language for programmers but not always the most efficient
- In scikit-learn, many functions are implemented in faster languages (e.g. C) and use specialized math libraries
- So in many cases, it is much faster to call the library once than many times:
  ```python
  import time
  t0 = time.time()
  guesses1 = classifier.predict(X_eval)
  t1 = time.time()
  guesses2 = [classifier.predict(x) for x in X_eval]
  t2 = time.time()
  
  print(t1-t0)
  print(t2-t1)
  
  result: 0.29 sec and 45 sec
  ```
some other practical functions

- splitting the data:

```python
from sklearn.cross_validation import train_test_split
train_files, dev_files = train_test_split(td_files,
                                          train_size=0.8,
                                          random_state=0)
```

- evaluation, e.g. precision, recall, F-score:

```python
from sklearn.metrics import f1_score
print(f1_score(Y_eval, Y_out))
```

- note that we’re using our own evaluation in this assignment, since we need more details