Homework 4: Classification with Scikit-Learn

Dr. Benjamin Roth
Computerlinguistische Anwendungen

Due: Friday May 11, 2018, 16:00

In this homework you will learn something about the Python library 'scikit-learn' or 'sklearn', a well known and very useful toolbox for research areas like data analysis and machine learning.

Exercise 1: CountVectorizer [4 points]

Complete the function `trigram_quadragram_vectorizer(texts)` that takes a list of text strings, and returns a `CountVectorizer` that considers all trigrams and quadragrams that occur in at least 3 of the given texts. Use the CountVectorizer defaults for preprocessing of text (tokenization, lower-casing etc.). You can test your function with:
`python3 -m unittest -v hw04_sklearn_paraphrases/test_small_functions.py`

Exercise 2: Dict Vectorizer

In this exercise we will do some small experiments with sklearn's `DictVectorizer`. To complete this exercise you will need the documentation: `http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.DictVectorizer.html`

Exercise 2.1: Dict Vectorizer - Part 1 [4 points]

In Machine Learning in the context of NLP a common task is to efficiently transform given dictionaries of word counts to feature matrices, where each row stands for a dictionary (e.g. a sentence) and each column for a word. Each position in the matrix denotes the number of occurences of a word in a sentence. So given a large number of dictionaries, one must:

- Derive the set of all words that occur in any sentence.
- Iterate over all dictionaries, count the word occurrences and fill them into the matrix.

This can easily be done with sklearn's `DictVectorizer`! In the file `sklearn_experiments.py` write a function `make_matrix1` that performs these two things on a given list. The function should return a `scipy.sparse` matrix.
Call the function on `list_of_dicts_1` and examine the result.
Note: Usually the sklearn DictVectorizer works with *sparse matrices* which is indispensable when working with large data. But to examine these toy matrices you might want to convert the returned sparse matrix. You can also use our `print_sparse_matrix` function to print it.

Exercise 2.2: Dict Vectorizer - Part 2 [4 points]

In applications, usually the training data is transformed to such a matrix. But it is important to understand that if new sentences come in to be classified, they must be transformed to a matrix with the same number of columns as the training matrix! The bag of words features are defined by the training data only!
Write a function `make_matrix2(list_of_dicts_1, list_of_dicts_2)` that uses sklearn's DictVectorizer to do the following:

- Consider `list_of_dicts_1` to be your 'training data' that defines the known words.
- Transform `list_of_dicts_2` to a feature matrix with respect to the words seen in `list_of_dicts_1`. (Count only words that have been seen in `list_of_dicts_1`).

The function should return a `scipy.sparse` matrix. Call the function on `list_of_dicts_1` and `list_of_dicts_2` and examine the result. The matrix should have the same shape as the one from Ex 2.1.

To check if your code for exercise 2 works correctly, call the unittest:

```
python3 -m unittest -v hw04_sklearn_paraphrases/test_sklearn_experiments.py
```

Exercise 3: Paraphrase Detection

In this exercise we will use the tools provided by sklearn (including the DictVectorizer) to again approach the paraphrase dectection task that you already know from last homework.

Exercise 3.1: From files to feature matrices [4 points]

In the file `paraphrases_scikit.py` complete the function `paraphrases_to_dataset`. This function is analogical to the function from last exercise and should do the following things:

- Given a filename, all lines in the file should be read and converted to a `features`-dictionary just like in the last homework. (Code is already there).

- If no DictVectorizer is given, the function should create a new one and fit it with the feature Dictionaries created before.
- The DictVectorizer should now be used to create a sparse feature matrix from the feature dictionaries created before.
- The function returns the feature matrix, the extracted labels, and the vectorizer.

Exercise 3.2: Obtaining our matrices [4 points]

Complete the function `readData`. This function should use `paraphrases_to_dataset` to create the following things:

- Training matrix `train_X`, training labels `train_Y` and a `vectorizer` based on the training data.
- Development matrix `dev_X` and development labels `dev_Y` based on the previously constructed `vectorizer`.
- Same for testing: `test_X` and `test_Y`

Exercise 3.3: Classifying [4 points]

With sklearn one can create a classifier by a single line of code. In this example, we'll try different parameter settings for two types of classifiers: logistic regression (=MaxEnt) and Support Vector Machines.

The classifiers work on the paraphrase detection task. The only thing missing is to pass the training matrix and labels to the classifier. Search `http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html` for a way to do that!

To check if your code for exercise 3 works correctly, call the unittest:
`python3 -m unittest -v hw04_sklearn_paraphrases/test_paraphrases_scikit.py`

To see if everything works right and to get some actual results, move into the `src` folder and call:

```
python3 -m hw04_sklearn_paraphrases.paraphrases_scikit -t
  data/paraphrases/train.txt -d data/paraphrases/dev.txt -e
  data/paraphrases/test.txt
```

Update (4.5.2018):
In order for the script to work, you need to move the definition of `intersection_size(i,k)` before the statement `if __name__ == "__main__":`
You should receive something like this:

```
Classifier: LinearSVC(C=0.1) - Development Accuracy: 0.7190
Classifier: LinearSVC(C=1.0) - Development Accuracy: 0.7190
Classifier: LogisticRegression(C=0.01, penalty="l2") - Development Accuracy: 0.7378
Classifier: LogisticRegression(C=0.1, penalty="l2") - Development Accuracy: 0.7255
```

```
Classifier: LogisticRegression(C=1.0, penalty="l2") - Development Accuracy: 0.7173
Classifier: LogisticRegression(C=0.01, penalty="l1") - Development Accuracy: 0.7405
Classifier: LogisticRegression(C=0.1, penalty="l1") - Development Accuracy: 0.7431
Classifier: LogisticRegression(C=1.0, penalty="l1") - Development Accuracy: 0.7088
Best classifier: LogisticRegression(C=0.1, penalty="l1") - Test Accuracy: 0.8687
```