Lab3

February 24, 2022

1 Lab 3

We'll use this lab as an experiment of using a single file where you fill in codeblocks where necessary. They will be available as .py and .ipynb. Using the latter, or Jupyter Notebook, is highly recommended, as it provides substantially better feedback.

Provide your outputs in a simple report, along with textual answers.

The idea behind this format is to clarify what sort of output is required, as all answers run on tests based in the `tests.py` file.

[ ]:
```
import sklearn
import nltk
import random
import pandas as pd
import re
# feel free to import from modules of sklearn and nltk later
# e.g., from sklearn.model_selection import train_test_split
from sklearn.model_selection import train_test_split
```

1.1 Exercise 1 - Gender detection of names

In NLTK you'll find the corpus `corpus.names`. A set of 5000 male and 3000 female names. 1) Select a ratio of train/test data (based on experiences from previous labs perhaps?) 2) Build a feature extractor function 3) Build two classifiers: - Decision tree - Naïve bayes

Finally, write code to evaluate the classifiers. Explain your results, and what do you think would change if you altered your feature extractor?

[ ]:
```
class GenderDataset:
    def __init__(self):
        self.names = nltk.corpus.names
        self.data = None
        self.build()

    def make_labels(self, gender):
        """
        this function is to help you get started
        based on the passed gender, as you can fetch from the file ids,
        we return a tuple of (name, gender) for each name
```

```
        use this in `build` below, or do your own thing completely :)
        """
        return [(n, gender) for n in self.names.words(gender + ".txt")]

    def build(self):
        """ TODO
        combine the data in "male" and "female" into one
        remember to randomize the order
        """
        pass

    def split(self, ratio):
        return train_test_split(self.data, test_size=ratio)
```

[ ]:
```
class Classifier:
    def __init__(self, classifier):
        self.classifier = classifier
        self.model = None

    def train(self, data):
        # TODO: train classifier and store model
        pass

    def test(self, data):
        # TODO: return accuracy for the model on input data
        pass

    def train_and_evaluate(self, train, test):
        self.train(train)
        return self.test(test)

    def show_features(self):
        # OPTIONAL
        pass


class FeatureExtractor:
    def __init__(self, data):
        self.data = data
        self.features = []

        self.build()

    @staticmethod
    def text_to_features(name):
        # TODO: create a dict of features from a name
```

```
        return {
            "name": name
        }

    def build(self):
        # TODO: populate your features with the above function
        pass
```

Note: you should achieve an accuracy of well above 70%!

[ ]:
```
split_ratio = 0.01  # TODO: modify
train, test = GenderDataset().split(ratio=split_ratio)

classifiers = {
    "decision_tree": Classifier(None), # TODO
    "naive_bayes": Classifier(None), # TODO
}

train_set = FeatureExtractor(train).features
test_set = FeatureExtractor(test).features

for name, classifier in classifiers.items():
    acc = classifier.train_and_evaluate(train_set, test_set)
    print("Model: {}\tAccuracy: {}".format(name, acc))
```

1.2 Exercise 2 - Spam or ham

Spam or ham is referred to a mail being spam or regular ("ham"). Follow the instructions and implement the `TODO`s

[ ]:
```
spam = pd.read_csv(
    'spam.csv',
    usecols=["v1", "v2"],
    encoding="latin-1"
).rename(columns={"v1": "label", "v2": "text"})

print(spam.label.value_counts())
spam.head()
```

[ ]:
```
""" TODO: transform label to numerical
Expected output:
0    4825
1     747
Name: label, dtype: int64

hint: you can use "apply" or "replace" for a column in pandas
"""
spam.label = spam.label # your transformation goes here
```

```
spam.label.value_counts()
```

[ ]:
```
class TextCleaner:
    def __init__(self, text):
        self.text = [] # TODO: tokenize
        self.stemmer = None # TODO: incorporate a stemmer of your choice
        self.stopwords = None  # TODO: you've done this a few times
        self.lem = None  # TODO: lemmatizer

    """
    Create small functions to replace your tokens (self.text)
    iteratively. Such as a lowercase function.
    """
    def lowercase(self):
        self.text = [w.lower() for w in self.text]

    def clean(self):
        self.lowercase()
        """
        TODO: populate with your defined cleaning functions here
        perhaps you want some conditional values to
        control which functions to use?
        """

        # finally, return it as a text
        return " ".join(self.text)
```

[ ]:
```
clean = lambda text: TextCleaner(text).clean()
spam.text = spam.text.apply(clean)
```

[ ]:
```
spam.head()
```

[ ]:
```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import confusion_matrix

split_ratio = 0.01 # TODO: modify
X_train, X_test, y_train, y_test = train_test_split(
    spam.text, spam.label, test_size=split_ratio, random_state=4310)

# TODO: vectorize with sklearn
vectorizer = None
# TODO: fit the vectorizer to your training data
X_train = None

# TODO: set up a multinomial classifier
classifier = None
if classifier:
```

```
    classifier.fit(X_train, y_train)
```

[ ]:
```
def predict(model, vectorizer, data, all_predictions=False):
    data = None # TODO apply the transformation from the vectorizer to test␣
 ↪data
    if all_predictions:
        return model.predict_proba(data)
    else:
        return model.predict(data)

def print_examples(data, probs, label1, label2, n=10):
    percent = lambda x: "{}%".format(round(x*100, 1))

    for text, pred in list(zip(data, probs))[:n]:
        print("{}\n{}: {} / {}: {}\n{}".format(
            text,
            label1,
            percent(pred[0]),
            label2,
            percent(pred[1]),
            "-" * 100  # to print a line
        ))
```

[ ]:
```
if classifier:
    y_probas = predict(classifier, vectorizer, X_test, all_predictions=True)
    print_examples(X_test, y_probas, "ham", "spam", n)

    y_pred = predict(classifier, vectorizer, X_test)
    # TODO display a confusion matrix on the test set vs predictions
    confusion_mat = None
    print(confusion_mat)

    # show precision and recall in a confusion matrix
    tn, fp, fn, tp = confusion_mat.ravel()
    recall = tp / (tp + fn)
    precision = tp / (tp + fp)

    print("Recall={}\nPrecision={}".format(round(recall, 2), round(precision,␣
 ↪2)))
```

1.3 Exercise 3 - Word features

Word features can be very useful for performing document classification, since the words that appear in a document give a strong indication of what its semantic content is. However, many words occur very infrequently, and some of the most informative words in a document may never have occurred in our training data. One solution is to make use of a lexicon, which describes how different words relate to each other.

Your task: - Use the WordNet lexicon and augment the movie review document classifier (See NLTK book, Ch. 6, section 1.3) to use features that generalize the words that appear in a document, making it more likely that they will match words found in the training data.

Download wordnet and import

[ ]:
```
nltk.download('wordnet')
from nltk.corpus import movie_reviews
from nltk.corpus import wordnet as wn
import random
```

[ ]:
```
# TODO: implement a function that returns a synonym for "word" if available,␣
 ↪otherwise return the word itself
def word_to_syn(word):
    pass
```

[ ]:
```
"""
this is from Ch. 6, sec. 1.3, with slight modifications
note that word_to_syn(word) (from the above implementation)
is in the beginning of the following function
"""
documents = [([word_to_syn(word) for word in list(movie_reviews.
 ↪words(fileid))], category)
             for category in movie_reviews.categories()
             for fileid in movie_reviews.fileids(category)]

random.shuffle(documents)

all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
n_most_freq = 2000
word_features = list(all_words)[:n_most_freq]

def document_features(document):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features
```

[ ]:
```
featuresets = [(document_features(d), c) for (d, c) in documents]

split_ratio = 0.01 # TODO: modify
train_set, test_set = train_test_split(featuresets, test_size=split_ratio)

# TODO: select a suitable classifier
classifier = None
model = classifier.train(train_set)
```

[ ]:
```python
# TODO: return a flattened list of input words and their lemmas
def synset_expansion(words) -> list:
    pass

expanded_word_features = synset_expansion(word_features)
```

[ ]:
```python
# some assertions to test your code :-)
assert sorted(synset_expansion(["pc"])) == ["microcomputer", "pc",␣
↪"personal_computer"]
assert sorted(synset_expansion(["programming", "coder"])) == [
    'coder',
    'computer_programing',
    'computer_programmer',
    'computer_programming',
    'program',
    'programing',
    'programme',
    'programmer',
    'programming',
    'scheduling',
    'software_engineer'
]
```

[ ]:
```python
doc_featuresets = [(document_features(d), c) for (d, c) in documents]
doc_train_set, doc_test_set = train_test_split(doc_featuresets, test_size=0.1)

doc_model = model.train(doc_train_set)
doc_model.show_most_informative_features(5)
print("Accuracy: ", nltk.classify.accuracy(doc_model, doc_test_set))
```

[ ]:
```python
def lexicon_features(reviews):
    review_words = set(reviews)
    features = {}
    for word in expanded_word_features:
        if word not in word_features:
            features['synset({})'.format(word)] = (word in review_words)
        features['contains({})'.format(word)] = (word in review_words)

    return features
```

Question: do you see any issues with including the synsets? Experiment a bit with different words and verify your ideas.

[ ]:
```python
# warning: this may take some time to run
lex_featuresets = [(lexicon_features(d), c) for (d, c) in documents]
lex_train_set, lex_test_set = train_test_split(lex_featuresets, test_size=0.1)
lex_model = model.train(lex_train_set)  # the same classifier as you defined␣
↪above
```

```
lex_model.show_most_informative_features()
print("Accuracy: ", nltk.classify.accuracy(lex_model, lex_test_set))
```

1.4 Exercise 4 – Experimentation

This exercise is largely open to experiment with and testing your skills thus far! Large websites are an ideal place to look for large corpora of natural language. In this exercise, you're free to implement what you've learned on real-world data, mined from youtube (see `youtube_data`). Reuse classes defined earlier on in the exercise if you want.

The only requirement here is to **use a classifier not previously used in the exercise**

[ ]:
```

```