CPSC-340 Machine Learning and Data Mining 2009

Classifier Competition

1 Competition rules

The competition is to create a text classifier that determines whether a datum is about a topic (`+1` label) or not about a topic (`-1` label).

1. We reserve the right to change the rules under reasonable circumstances. Changes will be posted to the newsgroup.
2. You may work singly or in pairs. No more than two people to a team, please.
3. You must have an average of at least 50% on both midterms to participate in the competition.
4. The competition closes on November 30th (2009) at 5pm.
5. Competition files are available on the web at `http://www.cs.ubc.ca/~nando/340-2009/project.php`
6. There are three train and three test data sets. **experimental train** and **experimental test** are the data provided for you to work with. **validation1 train/test** is a validation set that you may test your classifier against no more than once a week (see below). **validation2 train/test** is a final validation set that will be used to determine the competition winner.
7. Each of the data sets is in the form of wikipedia articles – the positive data are articles on related (though not necessarily the same) topics. Negative data are other wikipedia articles. The topics of the positive data in `experimental`, `validation1` and `validation2` are different, so make sure your classifier isn't biased toward the topic of **experimental**. The articles in the three negative sets are also different.
8. The winner of the competition is the individual or pair that achieves the lowest error on the average of **validation1** and **validation2**. The winning team will be awarded 25% to the final course mark. The first runner up team (silver) will be awarded 10% and the second one (bronze) 5%. There will be one award for sportsmanship of 5% and another one originality of 5%. The TAs and instructors will decide which groups should receive the last two awards.
9. Error is defined as the number of misclassifications on a data set divided by the total size of the data set.
10. You can test your classifier against the **validation1** data set by sending your classifier to Eric (ebrochu@cs.ubc.ca), in the form of a *single Python file*.
    (a) The file must have the `testClassifier` function at the bottom, like `exampleclassifier.py`, unmodified, except for filling in the name of your classifier.
    (b) Every Thursday, all the classifiers received by midnight the previous night will be tested and their score against **validation1** will be posted to the newsgroup.

(c) You are only allowed to submit your classifier once per week, so make sure it runs. If we can't get your classifier to run, it will not be scored.

(d) If your classifier takes an inordinate amount of time to train or test (more than a minute or two), or uses an excessive amount of memory, it will be terminated.

(e) The test machine is a recent (2009) iMac running OSX and with the latest version of Enthought and NLTK installed. Modules not included in Python, Enthought or NLTK probably won't work.

(f) Do not change or submit the `competition.py` file. We'll be using our own version.

11. After the results for a classifier are published, the group or individual responsible for it, must type one paragraph in the newsgroup explaining their approach.

12. You can use NLTK for tokenizing, stemming and similarly basic text processing only. You cannot use NLTK's classifiers (even as part of your own classifier), though you're welcome to test your classifier against them.

13. Your classifier must fill out the `sname` parameter with your name(s), and the `cname` parameter with the name of your classifier (anything reasonable you want, just make it short and unique). The `about` parameter is a way to identify the version of your classifier. It's a good idea to change it each time you submit so you can make sure the right version is being tested.

14. The fact that your classifier passes unit tests and runs on the test data on your computer is no guarantee it will work on our computer on the validation sets, and *we won't be helping you debug your code!* You are *strongly* encouraged to submit well before the deadline to make sure it works.

2 Mini-Tutorial: Classes, Exceptions and Unit Tests

Once again, the goal of this mini-tutorial is to give you just enough information to get started. You'll need to read the relevant sections of the `python.org` tutorial to master these tools.

Classes are an important part of Python. A typical Python class contains *data attributes* and *methods*. Let's take a look at a simple example.

```
class Foo(object):
    """
    This foo class doesn't do much. It inherits from 'object', the generic base
    type for classes.
    """

    def __init__(self, value=None):
        """ constructor """
        self.value = value  # by convention, an instance of a class refers to
                            # itself with 'self'.  the default of value is None.

    def printValue(self):   # self should be the first argument for most
                            # methods.  it allows the object to refer to itself.
        if self.value is not None:
            print 'Foo value =', self.value
        else:
            print 'Foo has no value defined.'
```

You create an instance of a class by calling it with the `()` notation. You can access its attributes with the `.` notation.

```
>>> from classtut import Foo
>>> a = Foo('test')
>>> b = Foo()
>>> a.printValue()
Foo value = test
>>> a.printValue()
Foo has no value defined.
>>> a.value
'test'
>>> b.value
None
```

Note that in Python, all attributes are public (more or less), though by convention (*ie* not enforced by the language, but you'll be a very unpopular programmer if you don't obey it), attributes preceded by an underscore are intended only for class internal use.

Inheritance in Python is straightforward. A subclass has access to it's superclass' attributes. Attributes can be overridden (in C++ terms, all methods are virtual), but are still available with the `super()` function.

```
class Bar(Foo):
    """ this class inherits from Foo """

    def __init__(self, value=None, nprint=2):
        self.nprint = nprint
        super(Bar, self).__init__(value)    # we overloaded Foo's constructor,
                                            # but we can still call it
```

```
    def printValue(self):
        """
        overriding Foo's method, but using its data attribute
        """
        if self.value is not None:
            val = ' - '.join(str(self.value) for _ in xrange(self.nprint))
            print 'Bar value =', val
        else:
            print 'Bar has no value defined.'
```

Then from the interactive session:

```
>>> from classtut import Bar
>>> a = Bar('test')
>>> b = Bar(0, 3)
>>> a.printValue()
Bar value = test - test
>>> b.printValue()
Bar value = 0 - 0 - 0
>>> super(Bar, b).printValue()
Foo value = 0
```

Python uses a `try ... except` syntax for exception handling, with two optional keywords: `else` (execute this block if there was no exception) and `finally` (always execute this block when leaving the `try` block). For example:

```
>>> n = 0
>>> while True:
...     d = raw_input('denominator? ')
...     try:
...         x = 10./float(d)
...     except ZeroDivisionError:
...         print "you can't divide by zero!"
...     else:
...         break
...     finally:
...         n += 1
...         print '\tthat was attempt', n
denominator? 0
you can't divide by zero!
    that was attempt 1
denominator? 2
    that was attempt 2
>>> x
5.0
```

Exceptions are classes, which inherit from the `Exception` base class. You can use the `raise` keyword to raise built-in exceptions, or create your own. There is an example in the competition code.

Finally, *unit tests* are a way to help a developer test for errors. Python has a useful `unittest` module which provides a framework for easily adding and running tests on a module. For details, see the provided `unittest_classifier.py`. As you develop your classifier, you can add tests to make sure it does what you expect. When you run the unittest script, you should see a report:

```
>>> execfile('unittest_classifier.py')
testBaseClassifier (__main__.TestSuppliedClassifiers) ... ok
testExampleClassifier (__main__.TestSuppliedClassifiers) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
testChallenger (__main__.TestChallenger) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

You are not required to use unit testing or hand in your unit test file. This is purely an aid to the development task. Passing unit tests is an excellent way of helping to make sure your classifier will run on the validation sets (though by itself, it does not guarantee your code will run!).