Lab 2: Naïve Bayes Text Classification

edward.a.hunter@gmail.com

1. Introduction

In this lab, we develop simple code to train and test Naïve Bayes classifiers on text categorization problems as described in lecture. You are provided a Python project that contains a supervised learning template file and utilities to download and manage the test datasets. The template file provides all the routine Python plumbing for importing required components and parsing command line options. We have clearly identified how to customize the to make your algorithms using the toolkit scikit-learn, that provides pre-made functions for this and other learning tasks. Following code development, we make training and testing runs against our test corpora and show common ways to evaluate supervised learning performance. You will see that using tools like scikit can save enormous time and allow researchers to focus on substantive questions without requiring extensive background in programming machine learning algorithms.

Note: In the sections to follow, examples are given that show work using a Linux bash command shell in Mac OS X. If you use a different OS than this, the precise appearance and command line instructions for listing files and the like may be slightly different. Please see the instructor or TA if you need any assistance determining the comparable commands in your OS. The Python code and commands for running our programs are identical.

2. Downloading the Lab Repository and Test Data

The class project code is kept in an online repository accessed by the **git** program. If you haven't downloaded the project code yet, open a command line window and from your home directory issue the **git clone** command with the URL of the repository as the argument. Here is the exact git clone command (first line) along with the output (your output may differ slightly)

```
(css)Edwards-MacBook-Pro:code edward$ git clone https://github.com/edwardhunter/soc290.git
Cloning into 'soc290'...
remote: Reusing existing pack: 12, done.
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 21 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (21/21), done.
```

Now, when you list the files in the current directory you will see a new directory present called /soc290. Switch to this new directory and list the files there and confirm you have the following files

```
(css)$ ls -oaF
total 240
drwxr-xr-x  12 edward    408 Mar 24 10:25 ./
drwxr-xr-x  50 edward   1700 Mar 24 10:25 ../
drwxr-xr-x  13 edward    442 Mar 24 10:25 .git/
-rw-r--r--   1 edward     56 Mar 24 10:25 .gitignore
-rw-r--r--   1 edward     56 Mar 24 10:25 README.md
-rw-r--r--   1 edward   3243 Mar 24 10:25 common.py
-rw-r--r--   1 edward  27771 Mar 24 10:25 cr_utils.py
-rw-r--r--   1 edward  13329 Mar 24 10:25 data_utils.py
-rw-r--r--   1 edward  35147 Mar 24 10:25 license.txt
-rw-r--r--   1 edward   6582 Mar 24 10:25 reuters_parser.py
-rw-r--r--   1 edward  12912 Mar 24 10:25 template_supervised.py
-rw-r--r--   1 edward    978 Mar 24 10:25 template_unsupervised.py
```

If you prefer, you can always use a file browser or code editor to inspect these directories instead of using the command line. Notice that you have a set of Python source files with the extension **.py**, an open source license and a readme used to describe the project contents.

We will add code and run our experiments in this directory to try out various learning methods, beginning with Naïve Bayes. First off, lets download our experimental datasets. We have included a Python program called **data_utils.py** to make this easy for you. Remember that Python programs can be run from the command line by running the Python interpreter program with the name of the program to run as the argument. First lets run the help option to see what options are available

```
(css)Edwards-MacBook-Pro:soc290 edward$ python data_utils.py --help
Usage: data_utils.py [options]

Download and construct training and testing data.

Options:
  -h, --help            show this help message and exit
  -d DATA_HOME, --directory=DATA_HOME
                        Home directory for data file (default: ./data).
```

We see that there are no mandatory arguments, and only one option to specify a directory for the data. The default is **./data**. The initial period is syntax for the current directory we are in, **/soc290**, so running this command with default options will create a new directory, **/soc290/data** where the data files will be created. We will use this default so go ahead and run data_utils.py (note that this will take a few minutes)

```
(css)$ python data_utils.py
./data
Downloading dataset 20 Newsgroups.
No handlers could be found for logger "sklearn.datasets.twenty_newsgroups"
Downloading dataset from http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.tar.gz (8.2
MB)
Decompressing ./data/reuters_home/reuters21578.tar.gz
Reading file: ./data/reuters_home/reut2-000.sgm
Reading file: ./data/reuters_home/reut2-001.sgm
Reading file: ./data/reuters_home/reut2-002.sgm
        ... ETC ...
Reading file: ./data/reuters_home/reut2-020.sgm
Reading file: ./data/reuters_home/reut2-021.sgm
```

Now if we list files in /soc290 we see

```
(css)$ ls -oaF
total 120
drwxr-xr-x  13 edward    442 Feb 17 14:24 ./
drwxr-xr-x  49 edward   1666 Feb 17 14:20 ../
drwxr-xr-x  13 edward    442 Feb 17 14:20 .git/
-rw-r--r--   1 edward     45 Feb 17 14:20 .gitignore
-rw-r--r--   1 edward     56 Feb 17 14:20 README.md
-rw-r--r--   1 edward   2361 Feb 17 14:20 common.py
-rw-r--r--   1 edward   2787 Feb 17 14:24 common.pyc
drwxr-xr-x   9 edward    306 Feb 17 14:26 data/
-rw-r--r--   1 edward  11251 Feb 17 14:20 data_utils.py
-rw-r--r--   1 edward   5840 Feb 17 14:20 reuters_parser.py
-rw-r--r--   1 edward   6812 Feb 17 14:24 reuters_parser.pyc
-rw-r--r--   1 edward  10823 Feb 17 14:20 template_supervised.py
-rw-r--r--   1 edward    259 Feb 17 14:20 template_unsupervised.py
```

So we now have the new directory /soc290/data/. Also notice that some files with the extension **.pyc** have appeared. These are complied byte code files created by Python as input to the Python virtual machine as an intermediate step as the source files (with extension .py) are processed. You don't have to be concerned with manipulating these. If we list the files in our new data directory we see our experimental data files

```
(css)$ ls -oaF data
total 87528
drwxr-xr-x   9 edward       306 Feb 15 16:40 ./
drwxr-xr-x  25 edward       850 Feb 17 13:57 ../
-rw-r--r--   1 edward  15665314 Feb 15 16:39 20news-bydate.pkz
-rw-r--r--   1 edward   9570194 Feb 15 16:39 20news.pkz
-rw-r--r--   1 edward   1798693 Feb 15 16:39 20news4.pkz
-rw-r--r--   1 edward   9237481 Feb 15 16:40 20news5.pkz
-rw-r--r--   1 edward     22142 Feb 15 16:40 reuters21578-10-idx.pkz
-rw-r--r--   1 edward   1963256 Feb 15 16:40 reuters21578-10.pkz
-rw-r--r--   1 edward   6538814 Feb 15 16:40 reuters21578.pkz
```

You'll notice various files for the 20 Newsgroups data, the Reuters newswire data and others. When we run our experiments we'll specify which dataset to use. The **.pkz** extension tells us these are "pickled" Python object files. That is, they are on-disk representations of Python data easy for our programs to read in and use and are portable across computer chip architectures. The templates we have constructed take care of all of

that for you. Note that you may see additional data files than those listed above as we continue to add experimental data after these labs are written.

1. The Supervised Learning Template File

To make life as easy as possible while exposing you to the best open source tools, we have prepared template files for supervised and unsupervised learning programs that take care of lots of routine plumbing and housekeeping, as well as the parts of the learning code that can be made the same between methods. This minimizes the amount of Python details you have to master before running experiments. Let's take a few moments and study the one we will adapt for this and the next two labs; it's called **template_supervised.py**. Go ahead and open it in your editor and start at the top of the file.

The first thing you will see is this

```
#!/usr/bin/env python

"""
@package css
@file css/template_supervised.py
@author Edward Hunter
@author Your Name Here
@brief A template to be customized for supervised learning experiments.
"""
```

Any line or portion of a line that starts with # is a comment that is ignored by Python. In this case the first line is a comment, **#!/usr/bin/env python**, that is a Unix convention telling the operating system that this is a Python script. The text enclosed by triple quotes """ is a multiline comment. This is a docstring prelude that gives some useful information about the file, such as the package, the filename, the author and so on. It tells the Python help command what to display and also is used by automatic documentation generating programs. It is good practice to keep such information in your files and we will do so as we extend the template for our own use.

Following a copyright and license notice, the next thing you will see in the template_supervised.py file are these lines

```
# Import common modules and utilities.
from common import *

# Define method and models available.
METHOD = ''
MODELS = ()
```

The # comments are here to tell us what the intent is of the code. The line **from common import *** is a Python module import. Remember there is a file in the directory called common.py. This line of code tells us to "import" all of the code objects from that file and make them available to the logic in this file, just as if that file's contents were directly written here. The **common.py** file contains lots of things that all of our programs use, so for

convenience we collect them in a separate file and import them all at once wherever they are needed. Feel free to check out what is in **common.py**. It is mostly additional import statements, bringing in parts of the Python standard library that we use, and also the pieces of the scikit-learn package that provide many of the learning algorithms and supporting utilities like feature extractors and evaluation tools.

The next two code lines define an empty string variable **METHOD** and an empty tuple variable **MODELS**. When we customize the template we use these variables to hold the name of the method and model options it supports, respectively. We'll do this in a few minutes when we start customizing the template for Naïve Bayes.

The bulk of the template_supervised.py file are the definitions for 3 functions called **train**, **predict** and **eval**. A function definition begins with the **def** statement followed by the function name and a "signature" of parameters enclosed in parentheses and ending in a colon, like this **def train(data, dataset, model, **kwargs):**

```
def train(data, dataset, model, **kwargs):
    """
    Train and store feature extractor, dimension reducer and classifier.
    @param: data training and testing dataset dictionary.
    @param: dataset dataset name string, valid key to data.
    @param: model model name string.
    @param: fappend optional file name appendix string.
    @param: dim optional dimension integer for reduction.
    """

    # Verify input parameters.
    if not isinstance(data, dict):
        raise ValueError('Invalid data dictionary.')

    if not isinstance(dataset, str):
        raise ValueError('Invalid dataset.')

    if not isinstance(model,str) or model not in MODELS:
        raise ValueError('Invalid model type parameter.')

    # Retrieve training data.
    data_train = data['train']
    data_train_target = data['train_target']

   ... LOTS MORE CODE
```

All the following lines that are indented comprise the function logic that is executed when the function is called. The triple quoted comment right under the def statement is another docstring that serves the same purpose as the prelude at the beginning of the file, but it applies to the train function specifically, telling what it does and what its parameters and output are. Scroll down and you will see similar definitions for the predict and eval functions. Together these three make up the framework for our supervised learning methods and experiments. If it is not yet obvious, we use the train function to train a particular supervised learning model given training data, the predict function to classify new data with a trained model and the eval(uate) function to run a set of performance evaluations on a trained model and compute a report.

Another thing to notice is that in four places in template_supervised.py you will see “to-do” comments that look like this one inside the train function

```
    ##############################################################
    # Create feature extractor, classifier.
    ##############################################################
    # TODO: create clf, vectorizer.
    ##############################################################
```

These to-do comments will be replaced by a very targeted amount of code you write using scikit to quickly develop programs based on the methods we covered in lecture. We will walk you though exactly how to do this so that you quickly gain experience with scikit and the machine learning development process, without first becoming a Python expert. Of these four to-dos, only the one above is needed for labs 2 and 3; the other 3 are specific for lab 4 only. All the surrounding code in these functions is common to all the methods and takes care of housekeeping chores like saving the trained models out to disk or reading in experiment data for training or testing, printing evaluation reports and the like. Studying this code is a good way to familiarize yourself with how the Python language works, and you can use these files in the future as models for your own work. If you have any questions about what the different pieces do as you study them, browse the excellent Python (http://www.python.org) and scikit (http://scikit-learn.org) documentation and feel free to ask the instructor or TA.

Finally, toward the end of the file you will notice a section of code that begins like this

```
if __name__ == '__main__':

    # Load training/testing utilities.
    from data_utils import load_data, DATASETS

    # Parse command line arguments and options.
    usage = 'usage: %prog [options] model dataset'
    usage += ('\n\tmodel = %s\n\tdataset = %s') % (MODELS, DATASETS)
    description = 'Train and evaluate supervised classifiers.'
    p = optparse.OptionParser(usage=usage, description=description)
    p.add_option('-f','--fappend', action='store', dest='fappend',
                 help='File name appendix string.')
    p.add_option('-d','--dim', action='store', dest='dim', type='int',
                 help='Reduced feature dimension integer.')
    p.add_option('-c', '--confusion', action='store', dest='confusion',
                 help='Save confusion image. Options: linear, log')
    p.add_option('-o', '--overwrite', action='store_true', dest='overwrite',
                 help='Overwrite existing files.')
    p.add_option('--df_min', action='store',type='float', dest='df_min',
                 help='Minimum document frequency proportion (default=None).')
    p.add_option('--df_max', action='store', type='float', dest='df_max',
                 help='Maximum document frequency proportion (default=1.0).')
    p.set_defaults(fappend=None, dim=None, confusion=None, overwrite=False)

    ##############################################################
    # Add method specific options.
    ##############################################################
    # TODO
    ##############################################################

    (opts, args) = p.parse_args()
    if len(args) < 2:
        p.print_usage()
        sys.exit(1)

    model = args[0]
    dataset = args[1]
```

This is called the program **entry point**. When a Python program is run using the python command, the interpreter automatically defines a global variable called **__name__** and assigns it the string value '**__main__**'. The program is executed line by line, but all that is done up until now is import some external code, define some global variables, and define three functions; there is no actual running of anything. This entry point says, in essence, if I am running as a Python program then take these steps following the if clause. The distinction here is that this file template_supervised.py, instead of being run directly, might be imported into another file so the train, predict and eval functions could be used in this other program file. In that case, the **__name__** variable would be the name of the calling program file and so this entry point code would not be run. This allows the program file to be run as a stand-alone program or imported into other programs to use and build on the functions defined here.

The first thing the entry point logic does is import the **load_data** function and the **DATASETS** variable that holds the names of all the available data from **data_utils.py**. This tells our program what data is available to it. Then an **OptionParser** object is created and

used to extract all of the command line arguments that are specified when the program is run. This parser also provides help text that describes the parameters for you. For example, when I run the file template_supervised.py with the **--help** option we see

```
(css)$ python template_supervised.py --help
Usage: template_supervised.py [options] model dataset
       model = ()
       dataset = ('20news', '20news4', '20news5', 'reuters21578-10', 'senate')

Train and evaluate supervised classifiers.

Options:
  -h, --help            show this help message and exit
  -f FAPPEND, --fappend=FAPPEND
                        File name appendix string.
  -d DIM, --dim=DIM     Reduced feature dimension integer.
  -c CONFUSION, --confusion=CONFUSION
                        Save confusion image. Options: linear, log
  -o, --overwrite       Overwrite existing files.
  --df_min=DF_MIN       Minimum document frequency proportion (default=None).
  --df_max=DF_MAX       Maximum document frequency proportion (default=1.0).
```

The model and dataset arguments are mandatory (in that order). The help text already knows about the datasets we have made available through load_utils.py. The model parameter options are empty because they are not yet defined. In addition, we have options for appending file names, dimension reduction, saving a confusion matrix image, overwriting existing model files, and filtering features based on document frequency.

Finally at the end of the file we have

```
    # Load data.
    data = load_data(dataset)

    # Create classifier, feature extractor and dim reducer names.
    (cfname, vfname, dfname, _, _) = \
        get_fnames(METHOD, model, dataset, dim, fappend)

    # If we are specified to overwrite, or if required files missing, train and
    # store classifier components.
    cfpath = os.path.join(MODEL_HOME,cfname)
    vfpath = os.path.join(MODEL_HOME,vfname)
    model_files_present = os.path.isfile(cfpath) and os.path.isfile(vfpath)
    if dfname:
        dfpath = os.path.join(MODEL_HOME,dfname)
        dim_files_present = os.path.isfile(dfpath)
    else:
        dim_files_present = True
    if overwrite or not model_files_present or not dim_files_present:
        train(data, dataset, model, dim=dim, fappend=fappend,
              df_min=df_min, df_max=df_max)

    # Evaluate classifier.
    eval(data, dataset, model, dim=dim, fappend=fappend, confusion=confusion)
```

This is the logic that uses the functions we defined above to train and evaluate our learning models. First the specified dataset is loaded. Then we determine if the specified model is present on disk and assign a Boolean to **model_files_present**.

If the files are not present, or if an overwrite option is specified, then we first call the train() function to train the model. Training can be an expensive process so once models are trained we skip this step unless told to overwrite and retrain. Then the eval() function is called to test the trained model and generate a report. The predict function is used by eval() to carry out this performance evaluation. After you have trained and validated a model this way, you could then use it in real world classification tasks by importing the predict function from this file into another program that was collecting and classifying your new, unseen data. So these templates and the programs we create from them can be used directly in real research tasks.

2. Create the Naïve Bayes Program

Make a copy of the template_supervised.py file in the same /soc290 directory and call it **naive_bayes.py**. Edit the beginning of the file to modify the prelude docstring and define the global variables this way (for brevity below, we omitted the copyright and license text, but leave that in your files).

```
#!/usr/bin/env python

"""
@package css
@file css/naive_bayes.py
@author Edward Hunter
@author: Your Name Here
@brief Naive Bayes supervised learning and evaluation methods.
"""

[COPYRIGHT AND LICENSE OMITTED]

# Import common modules and utilities.
from common import *

# Define method and models available.
METHOD = 'Naive_Bayes'
MODELS = ('Bernoulli','Multinomial','TFIDF')
```

Now the docstring carries the brief description for the Naïve Bayes program and the file is updated to the correct name. Add your name as an @author. Then define the METHOD variable as the string 'Naive_Bayes' and the MODELS variable to be a tuple of strings ('Bernoulli', 'Multinomial', 'TFIDF') defining the three bag of words frequency weighting options we will make available in this program.

Customizing the train() Function for Naïve Bayes

The first lines in the train() function look like this

```
    # Verify input parameters.
    if not isinstance(data, dict):
        raise ValueError('Invalid data dictionary.')

    if not isinstance(dataset, str):
        raise ValueError('Invalid dataset.')

    if not isinstance(model,str) or model not in MODELS:
        raise ValueError('Invalid model type parameter.')

    # Retrieve training data.
    data_train = data['train']
    data_train_target = data['train_target']

    # Retrieve options.
    dim = kwargs.get('dim', None)
    df_min = kwargs.get('df_min',1)
    df_max = kwargs.get('df_max',1.0)

    # Create dimension reducer.
    fselector = None
    if dim:
        fselector = SelectKBest(chi2, k=dim)
```

In this code we take the following setup steps:

1. Verify input parameters for validity.
2. Extract the training data and training labels from the data parameter.
3. Extract dimension reduction and frequency filtering options.
4. If the dimension reduction option is specified, create a feature selector object.

In step 4, the code looks for the **dim** argument in the parameters passed to train and, if present, uses it to construct a scikit dimension reducer object. The **SelectKBest** object is constructed, using **chi2** as the method for selecting specified number of dimensions **k=dim**. The **dim** is parameter is an optional parameter specified at the program command line as we saw in the help output above and forwarded to train if it is present. As before, **SelectKBest** and **chi2** are scikit imports defined in common.py.

Now we customize the train() function for Naïve Bayes. To do this, we replace the first to-do comment with the following code

```
    ##############################################################
    # Create feature extractor, classifier.
    ##############################################################
    if model == 'Bernoulli':
        vectorizer = CountVectorizer(stop_words='english', binary=True)
        clf=BernoulliNB(alpha=.01)
        if fselector:
            fselector.__normalize = False

    elif model == 'Multinomial':
        vectorizer = CountVectorizer(stop_words='english')
        clf=MultinomialNB(alpha=.01)
        if fselector:
            fselector.__normalize = False

    elif model == 'TFIDF':
        vectorizer = TfidfVectorizer(stop_words='english')
        clf=MultinomialNB(alpha=.01)
        if fselector:
            fselector.__normalize = True
    ##############################################################
```

This is an if-clause with three conditions that you recognize as the three models we will support, as defined in the global **MODELS** at the top of the file. Now we will call on scikit's methods to build the various flavors of Naïve Bayes text classifiers. There are two variables that are defined differently depending which model we select: **vectorizer** and **clf**. vectorizer is the bag-of-words feature extractor that converts the text documents into feature vectors of various kinds. clf is the actual supervised model that we will train and then does the predictions for us. These are the three cases

1. Bernoulli Model: the vectorizer is a scikit **CountVectorizer** object, which returns the word frequencies as the vector over the entire vocabulary seen in the training set. The parameter **stop_words='english'** has the effect of filtering out English "stop words" that are very common and functional in nature and contain little information in classification (e.g. as, the, is, it, which, on, …). The second parameter **binary=True** tells the vectorizer to return present/absent values rather than an actual word frequency. This is the Bernoulli model discussed in class. Then a Bernoulli Naïve Bayes classifier is created with the scikit **BernoulliNB** object. The parameter **alpha=0.1** is the smoothing factor that avoids zero probability estimates and allows the Naïve Bayes model to function reasonably well when it encounters terms not seen in training.
2. Multinomial Model: Here the vectorizer is the same as before except the binary=True parameter is omitted, allowing a full word frequency count to be computed as the document representation. The classifier is an instance of the scikit object **MultinomialNB** that implements the Multinomial Naïve Bayes model discussed in class. The alpha parameter has the same function as above.
3. TFIDF Model: In this case we construct a scikit **TfidfVectorizer** using English stop word filtering to return TFIDF weighted values for each document term. In essence, this model computes weighted features for each word that combine the ratio of word frequency across the whole corpus to the proportion of documents they appear in. We will see that this kind of feature weighting can greatly improve

classifier performance. The details on TFIDF weighting will be discussed in a future lecture.

Note that each of the above cases contains an if-clase on the **fselector** variable. fselector is the scikit object discussed above that is used to reduce feature dimension if the command line option –d is specified at the command line. When we reduce the feature dimension size with the TFIDF model, it is important to normalize the resulting vectors so they have a length of 1.0. We set an internal variable fselctor.__normalize to False for the Bernoulli and Multinomial models and True for the TFIDF model

The scikit objects we create, **CountVectorizer**, **TfidfVectorizer**, **BernoulliNB**, and **MultinomialNB** are not part of the native Python language. How is it we can call these classes to construct the vectorizer and clf variables above? In common.py, the set of common objects we import at the beginning of the file contains imports from scikit for these classes so they are available to us in our template program.

The train() function is now customized for Naïve Bayes. The next to-do is only used for Support Vector Machines, so remove the following lines

```
##############################################################
# Grid search and top feature output for SVMs.
##############################################################
# TODO: SVMs only.
##############################################################
```

Next lets briefly look at the rest of the common code to gain an understanding of what is happening in train().

```
# Extract features, reducing dimension if specified.
print 'Extracting text features...'
start = time.time()
x_train = vectorizer.fit_transform(data_train)
if fselector:
    x_train = fselector.fit_transform(x_train, data_train_target)
    if fselector.__normalize:
        x_train = normalize(x_train)
print 'Extracted in %f seconds.' % (time.time() - start)
print 'Feature dimension: %i' %x_train.shape[1]
print 'Feature density: %f' % density(x_train)

# Train classifier.
print 'Training classifier...'
start = time.time()
clf.fit(x_train, data_train_target)
print 'Trained in %f seconds.' % (time.time() - start)
```

In the first block of code, we call the vectorizer function called **fit_transform**. When you pass the raw text training data contained in **data_train** and their labels **data_train_target** to this function, a set of training vectors is returned from whichever vectorizer you are using as specified above (Bernoulli, Multinomial or TFIDF). This is the code that transforms the raw text into binary, frequency or TFIDF bag-of-words vectors. If the dim option is

specified, then the above constructed fselector **SelectKBest** object is used to reduce the feature dimension as specified. Note the use of the time module to record the start time and print out the total time taken for feature extraction. The time module is part of the Python standard library and is imported by common.py like the scikit classes. We'll see this time report as we run the program.

Now we have constructed feature extractor and classifier objects according to the model choice. We have also constructed a dimension reducer, if specified, and converted the raw text to numerical form. In the second block of code above, we call the **fit** function on the classifier object, taking the numerical bag-of-words features and training labels as parameters to calculate the Naïve Bayes model parameters from training data as we discussed in class. Here the time required is also calculated and reported as we go.

The final block of code in train() writes out the vectorizer, classifier and dimension reducer objects to disk for use by the predict() function.

```
    # Write out classifier.
    cpname = os.path.join(MODEL_HOME,cfname)
    fhandle = open(cpname,'w')
    pickle.dump(clf, fhandle)
    fhandle.close()
    print 'Classifier written to file %s' % (cpname)

    # Write out feature extractor.
    vpname = os.path.join(MODEL_HOME,vfname)
    fhandle = open(vpname,'w')
    pickle.dump(vectorizer, fhandle)
    fhandle.close()
    print 'Feature extractor written to file %s' % (vpname)

    # Write out dimension reducer.
    if dim:
        dpname = os.path.join(MODEL_HOME,dfname)
        fhandle = open(dpname,'w')
        pickle.dump(fselector, fhandle)
        fhandle.close()
        print 'Feature selector written to file %s' % (dpname)
```

Examining the predict() Function

We've set up supervised_template.py so we don't have to customize anything beyond the train() function and, in the case of SVMs, add some optional parameters. However we describe the other functions to understand program behavior and how scikit is used.

The first piece of predict() code verifies the input parameters similar to the beginning of the train() function. Next, the vectorizer, classifier and, if specified, the dimension reducer objects created and written to disk in train() are read in

```
    # Read in the classifer.
    cpname = os.path.join(MODEL_HOME,cfname)
    fhandle = open(cpname)
    clf = pickle.load(fhandle)
    fhandle.close()
    print 'Read classifer from file: %s' % cpname

    # Read in the feature extractor.
    vpname = os.path.join(MODEL_HOME,vfname)
    fhandle = open(vpname)
    vectorizer = pickle.load(fhandle)
    fhandle.close()
    print 'Read feature extractor from file: %s' % vpname

    # If requested, load the dimension reducer.
    dfname = kwargs.get('dfname', None)
    if dfname:
        dpname = os.path.join(MODEL_HOME,dfname)
        fhandle = open(dpname, 'r')
        fselector = pickle.load(fhandle)
        fhandle.close()
        print 'Feature selector read from file %s' % (dpname)
```

The real work done by predict() is done in just these few lines of code

```
    # Compute features and predict.
    x_test = vectorizer.transform(input_data)
    if dfname:
        x_test = fselector.transform(x_test)
        if fselector.__normalize:
            x_test = normalize(x_test)
    pred = clf.predict(x_test)

    # Return vector of predicted labels.
    return pred
```

Here the vectorizer object created by train() is used to transform the unseen testing data to a numerical vector **x_test**. An important thing to notice here is that there is no reference to the specific type of objects or models used for feature extraction or classification. Although they do them in different ways, the different vectorizers and classifiers all make the same set of functions available to us. This is a programming concept known as **polymorphism**. Similarly, the classifier is used to predict the category of the test data and stored in the **pred** variable, which is then returned as the output of predict().

Examining the eval() Function

Similar to the predict() function, we don't have any code to write for eval(). Due to polymorphism our template code is general and does not need any edits. Looking through the function we see the following steps. First, the input variables are verified similar to train() and eval(). Next the test data and labels are extracted from the data parameter. Filenames for the vecotizer, classifier and dimension reducer are created using **get_fnames**. Then eval() calls the predict() function we just studied, passing the test data and filenames and storing the prediction output in the variable **pred**.

Following prediction of the test data, eval() uses scikit methods to create a classification report and confusion matrix which are used to evaluate performance when we run the program.

```
    # Evaluate predictions: metrics.
    class_report = metrics.classification_report(data_test_target, pred,
                                                 target_names=data_target_names)
    conf_matrix = metrics.confusion_matrix(data_test_target ,pred)
```

As we will see when we run our experiments, the classification report gives us a table of categorical statistics that tell us how well our model is performing. The confusion matrix is an KxK table, with K the number of document categories, showing the number of correctly classified documents of each class as well as which categories misclassified documents were assigned to. In the confusion matrix, each row corresponds to the true data category, and the value at each column give the number of texts classified into each possible category. This helps us easily see which document categories are difficult for the model to discriminate between and how the errors are distributed.

The remainder of the eval() function formats and prints these statistics to the screen as the program runs and saves a report file to disk. If the optional argument **–c** or **--confusion** is specified when the program is run, then a png image of the confusion matrix is generated using a visualization library called **matplotlib**. matplotlib is imported in common.py and provides powerful set of plotting and visualization tools worthy of many lectures and exercises of its own. It has a programming interface that closely follows the popular **MATLAB** software, which you may be familiar with.

Editing the Entry Point

The function of the entry point was described above. In addition, the entry point has two to-do sections, but these are used only for SVM specific options so we can delete them. Remove the following lines

```
    ############################################################
    # Add method specific options.
    ############################################################
    # TODO: SVMs only.
    ############################################################

    ############################################################
    # Extract method specific options.
    ############################################################
    # TODO: SVMs only.
    ############################################################
```

3. Experimenting with the 20 Newsgroups Data

We are now ready to run some experiments. Now that we've constructed our naïve_bayes.py program file lets see what the command line syntax is. Ask for help for the program as before

```
(css) $ python naive_bayes.py --help
Usage: naive_bayes.py [options] model dataset
       model = ('Bernoulli', 'Multinomial', 'TFIDF')
       dataset = ('20news', '20news4', '20news5', 'reuters21578-10', 'senate')

Train and evaluate naive Bayes supervised classifiers.

Options:
  -h, --help            show this help message and exit
  -f FAPPEND, --fappend=FAPPEND
                        File name appendix string.
  -d DIM, --dim=DIM     Reduced feature dimension integer.
  -c CONFUSION, --confusion=CONFUSION
                        Save confusion image. Options: linear, log
  -o, --overwrite       Overwrite existing files.
  --df_min=DF_MIN       Minimum document frequency proportion (default=None).
  --df_max=DF_MAX       Maximum document frequency proportion (default=1.0).
```

Now we see that we have three models to select from as well as our choice of datasets. We'll start with the full 20 Newsgroups data described in lecture. Very quickly though, let's explain what happens when an error is encountered and what to do about it.

Dealing with Errors

It is almost never the case that code you write runs correctly the first time. The exception might be these labs where there is very limited coding and we tell you exactly what to do, but even here you might make a typo. For example, let's say that when you were filling out the train function we used **Countvectorizer** instead of **CountVectorizer**, forgetting to capitalize the V when constructing the feature extractor. Since Countvectorizer with a lower-case "v" is not defined anywhere this will result in an error. But how will it present when the code runs and how do we go about finding and fixing it? Let's run the program with this mistake and look at the output

```
(css)Edwards-MacBook-Pro:css edward$ python naive_bayes.py Bernoulli 20news
Loading: ./data/20news.pkz
[. . . DATA SUMMARY OMITTED . . .]
Traceback (most recent call last):
  File "naive_bayes.py", line 361, in <module>
    df_min=df_min, df_max=df_max)
  File "naive_bayes.py", line 77, in train
    vectorizer = Countvectorizer(stop_words='english', binary=True)
NameError: global name 'Countvectorizer' is not defined
```

So as the program runs we see our 20news.pkz data getting loaded, and then we see a **traceback** followed by an error message. The last line is the error message and gives us information on the type of error that was encountered. It tells us there is an undefined name 'Countvectorizer' that the interpreter cannot resolve. This is our typo of course. The traceback lines tell us, recursively, where the error is. The traceback is organized as a **call stack**. The first two lines specify that the error occurred when the entry point called the train function (line 361). The next two lines show where in the train function the actual offending code is (line 77). When we run our program the entry point calls train, and the train function encounters the undefined name error, so there only two steps in the

traceback. More complex programs will have longer tracebacks. Where in the traceback the mistake is located depends on if the error being thrown is in your own code or in a function you are using but didn't write yourself. For example, you may have passed a bad parameter to some Python built-in, standard library or scikit function. Although all the information you need to find the bug is usually there, it takes a little practice to develop good debugging skills. If you see output like this when running your program, see if you can identify and fix your errors yourself. Then, if you need assistance, just ask the instructor or the TA.

Analyzing the Data

The 20 newsgroups data contains short text documents from online newsgroups in approximately equal distributions, split into approximately equal training and testing sets. The full 20 newgroups categories are

```
20 Newsgroups Labels
0   'alt.atheism'
1   'comp.graphics'
2   'comp.os.ms-windows.misc'
3   'comp.sys.ibm.pc.hardware'
4   'comp.sys.mac.hardware'
5   'comp.windows.x'
6   'misc.forsale'
7   'rec.autos'
8   'rec.motorcycles'
9   'rec.sport.baseball'
10  'rec.sport.hockey'
11  'sci.crypt'
12  'sci.electronics'
13  'sci.med'
14  'sci.space'
15  'soc.religion.christian'
16  'talk.politics.guns'
17  'talk.politics.mideast'
18  'talk.politics.misc'
19  'talk.religion.misc'
```

So we will try training Naïve Bayes classifier to discriminate these documents based on our three bag-of-words models. Note that some of these categories are "closer" to each other than others. For example, the documents in the "comp" newsgroup categories will likely have lots of overlapping terms with each other, but less so with documents from the "rec" categories. The number of documents in the training and test classes are in the low hundreds, roughly 250-600 for each category, which is certainly not a large amount of data considering the vastness of the vocabulary space.

We're now ready to put Naïve Bayes to the test. Let's run the Bernoulli model against the 20 newgroups data. The command line syntax and output looks like this

```
(css)$ python naive_bayes.py -c linear Bernoulli 20news
Loading: ./data/20news.pkz
Class Names:
    0               alt.atheism  train size:      480, test size:      319
    1             comp.graphics  train size:      584, test size:      389
    2   comp.os.ms-windows.misc  train size:      591, test size:      394
    3  comp.sys.ibm.pc.hardware  train size:      590, test size:      392
    4     comp.sys.mac.hardware  train size:      578, test size:      385
    5            comp.windows.x  train size:      593, test size:      395
    6              misc.forsale  train size:      585, test size:      390
    7                 rec.autos  train size:      594, test size:      396
    8           rec.motorcycles  train size:      598, test size:      398
    9        rec.sport.baseball  train size:      597, test size:      397
   10          rec.sport.hockey  train size:      600, test size:      399
   11                 sci.crypt  train size:      595, test size:      396
   12           sci.electronics  train size:      591, test size:      393
   13                   sci.med  train size:      594, test size:      396
   14                 sci.space  train size:      593, test size:      394
   15    soc.religion.christian  train size:      599, test size:      398
   16        talk.politics.guns  train size:      546, test size:      364
   17     talk.politics.mideast  train size:      564, test size:      376
   18        talk.politics.misc  train size:      465, test size:      310
   19        talk.religion.misc  train size:      377, test size:      251
                       Totals:  train size:    11314, test size:     7532
Extracting text features...
Extracted in 2.641120 seconds.
Feature dimension: 101323
Feature density: 0.000659
Training classifier...
Trained in 0.147397 seconds.
Classifier written to file ./models/Naive_Bayes_Bernoulli_20news_clf
Feature extractor written to file ./models/Naive_Bayes_Bernoulli_20news_vec
Read classifer from file: ./models/Naive_Bayes_Bernoulli_20news_clf
Read feature extractor from file: ./models/Naive_Bayes_Bernoulli_20news_vec
[CLASSIFICATION REPORT AND CONFUSION MATRIX OMITTED]
```

The output tells us what happens. We loaded the 20news.pkz data, a summary of the data is printed to the screen broken down by cateogry, then the Bernoulli feature vectors are extracted from the documents and the Bernoulli classifier is trained. Note the helpful feature dimension and density information that is reported. In this dataset, our vectors have dimension over 101,000 with only about .066% of those dimensions present over the training set. Following training, the feature extractor and classifier are written to disk. When the eval and predict functions are called, the feature extractor and classifier are read back from disk and an evaluation is run. Finally, the classification report and confusion matrix and printed to the screen (not shown above) and also saved to disk. If you look at your /soc290 directory contents, you will notice two new subdirectories have appeared: **/soc290/models** and **/soc290/reports**. The trained feature extractors and classifiers are stored in the models directory and our classification reports are stored in the reports directory. For example we see this in /soc290/reports

```
(css)$ ls -l ./reports
total 96
-rw-r--r--  1 edward  staff  41142 Feb 17 08:27 Naive_Bayes_Bernoulli_20news_confusion.png
-rw-r--r--  1 edward  staff   3681 Feb 17 08:27 Naive_Bayes_Bernoulli_20news_report.txt
```

The txt file contains the same report output that was written to the screen. In addition a confusion image with the extension png is here. This was created because we used the –c option when we ran the program. Confusion matrix images are useful when the number of categories starts to get big, because they allow us to gain a quick feel for how errors are distributed across complex categories. Open and examine both files. You will collect these report files and confusion images (for the full 20 news and Reuters data) into your lab report.

We will now repeat this process for the full 20 newsgroups data using Multinomial and TFIDF models (omitting the data summaries and report outputs)

```
(css)$ python naive_bayes.py -c linear Multinomial 20news
Loading: ./data/20news.pkz
Extracting text features...
Extracted in 2.591580 seconds.
Feature dimension: 101323
Feature density: 0.000659
Training classifier...
Trained in 0.143173 seconds.
Classifier written to file ./models/Naive_Bayes_Multinomial_20news_clf
Feature extractor written to file ./models/Naive_Bayes_Multinomial_20news_vec
Read classifer from file: ./models/Naive_Bayes_Multinomial_20news_clf
Read feature extractor from file: ./models/Naive_Bayes_Multinomial_20news_vec
```

```
(css)$ python naive_bayes.py -c linear TFIDF 20news
Loading: ./data/20news.pkz
Extracting text features...
Extracted in 2.719179 seconds.
Feature dimension: 101323
Feature density: 0.000659
Training classifier...
Trained in 0.135132 seconds.
Classifier written to file ./models/Naive_Bayes_TFIDF_20news_clf
Feature extractor written to file ./models/Naive_Bayes_TFIDF_20news_vec
Read classifer from file: ./models/Naive_Bayes_TFIDF_20news_clf
Read feature extractor from file: ./models/Naive_Bayes_TFIDF_20news_vec
```

Our reports directory now has six files in it, summarizing our work so far

```
(css)$ ls -oaF reports
total 288
drwxr-xr-x   8 edward    272 Feb 17 14:15 ./
drwxr-xr-x  25 edward    850 Feb 17 13:57 ../
-rw-r--r--   1 edward  41142 Feb 17 08:27 Naive_Bayes_Bernoulli_20news_confusion.png
-rw-r--r--   1 edward   3681 Feb 17 08:27 Naive_Bayes_Bernoulli_20news_report.txt
-rw-r--r--   1 edward  42728 Feb 17 11:03 Naive_Bayes_Multinomial_20news_confusion.png
-rw-r--r--   1 edward   3681 Feb 17 11:03 Naive_Bayes_Multinomial_20news_report.txt
-rw-r--r--   1 edward  42024 Feb 17 11:03 Naive_Bayes_TFIDF_20news_confusion.png
-rw-r--r--   1 edward   3681 Feb 17 11:03 Naive_Bayes_TFIDF_20news_report.txt
```

The 20news4 and 20news5 Datasets

So far we have collected report evaluation data for three Naïve Bayes models for the full 20 newsgroups dataset. To gain a sense of how performance can vary with the variables in problem formulation, we are going to repeat these analysis on two additional datasets based of the full 20 newsgroups data.

The **20news4** dataset is just the newsgroup data but with only for four non-similar document categories included. The categories are

```
4 Newgroups Labels:
1   'comp.graphics'
7   'rec.autos'
14  'sci.space'
19  'talk.religion.misc'
```

So if we limit the focus of the classification task to fewer, less similar document categories will we see any performance differences?

The **20news5** dataset is interesting in a different way. Here we collapse document categories that are similar into 5 super-categories. This increases the training/testing data for each category and reduces the complexity of the classification problem. The super-categories for this dataset are

```
5 Newsgroups Labels:
0   computers
    1 'comp.graphics'
    2 'comp.os.ms-windows.misc'
    3 'comp.sys.ibm.pc.hardware'
    4 'comp.sys.mac.hardware'
    5 'comp.windows.x'
1   recreation
    7   'rec.autos'
    8   'rec.motorcycles'
    9   'rec.sport.baseball'
    10  'rec.sport.hockey'
2   science
    11  'sci.crypt'
    12  'sci.electronics'
    13  'sci.med'
    14  'sci.space'
3   politics
    16  'talk.politics.guns'
    17  'talk.politics.mideast'
    18  'talk.politics.misc'
4   religion
    0   'alt.atheism'
    15  'soc.religion.christian'
    19  'talk.religion.misc'
```

In both cases code in the data_utils.py program has already created these datasets for you and you only need to specify which you want to use when you run the Naïve Bayes program. Let's run all three Naïve Bayes models on both of these datasets, so that with our previous results we have a total of nine sets of results to look at. The command line syntax

for each case is just as above, and done six times for each combination of one of the models with one of these two new datasets. When done the data in /soc290/reports now looks like

```
(css)$ ls -lao reports/
total 720
drwxr-xr-x  20 edward    680 Mar  3 18:34 .
drwxr-xr-x  26 edward    884 Mar  2 14:33 ..
-rw-r--r--   1 edward  30315 Mar  3 18:02 Naive_Bayes_Bernoulli_20news4_confusion.png
-rw-r--r--   1 edward    707 Mar  3 18:02 Naive_Bayes_Bernoulli_20news4_report.txt
-rw-r--r--   1 edward  30588 Mar  3 18:02 Naive_Bayes_Bernoulli_20news5_confusion.png
-rw-r--r--   1 edward    789 Mar  3 18:02 Naive_Bayes_Bernoulli_20news5_report.txt
-rw-r--r--   1 edward  41142 Mar  3 18:02 Naive_Bayes_Bernoulli_20news_confusion.png
-rw-r--r--   1 edward   3734 Mar  3 18:02 Naive_Bayes_Bernoulli_20news_report.txt
-rw-r--r--   1 edward  30862 Mar  3 18:04 Naive_Bayes_Multinomial_20news4_confusion.png
-rw-r--r--   1 edward    709 Mar  3 18:04 Naive_Bayes_Multinomial_20news4_report.txt
-rw-r--r--   1 edward  32257 Mar  3 18:04 Naive_Bayes_Multinomial_20news5_confusion.png
-rw-r--r--   1 edward    791 Mar  3 18:04 Naive_Bayes_Multinomial_20news5_report.txt
-rw-r--r--   1 edward  42728 Mar  3 18:03 Naive_Bayes_Multinomial_20news_confusion.png
-rw-r--r--   1 edward   3736 Mar  3 18:03 Naive_Bayes_Multinomial_20news_report.txt
-rw-r--r--   1 edward  29628 Mar  3 18:05 Naive_Bayes_TFIDF_20news4_confusion.png
-rw-r--r--   1 edward    703 Mar  3 18:05 Naive_Bayes_TFIDF_20news4_report.txt
-rw-r--r--   1 edward  30724 Mar  3 18:05 Naive_Bayes_TFIDF_20news5_confusion.png
-rw-r--r--   1 edward    785 Mar  3 18:05 Naive_Bayes_TFIDF_20news5_report.txt
-rw-r--r--   1 edward  42024 Mar  3 18:05 Naive_Bayes_TFIDF_20news_confusion.png
-rw-r--r--   1 edward   3730 Mar  3 18:05 Naive_Bayes_TFIDF_20news_report.txt
```

So we have 18 files: nine report txt files, and nine confusion images, one for each model-dataset combination used used.

4. Experimenting with the Reuters Data

In this section we repeat our experiments with the Reuters dataset. The full dataset is composed of 21,578 newswire texts that have been hand categorized by Reuters personnel. However, the full dataset is problematic since many of the texts are labeled into multiple categories and some categories have very few documents. To overcome this, our data_utils.py program parses the full dataset and extracts the documents that are labeled with a single category, retaining the ten categories with the most example documents. The top categories and the number of documents in each are

```
        'earn',              # 3945
        'acq',               # 2362
        'crude',             # 408
        'trade',             # 362
        'money-fx',          # 307
        'interest',          # 285
        'money-supply',      # 161
        'ship',              # 158
        'sugar',             # 143
        'coffee'             # 116
```

So one difference we immediately see with the Reuters data is that it is “unbalanced” in the sense that there are large differences in the number of examples in the top categories.

Lets run our three Naïve Bayes models with the Reuters data and see how they do. This is done just like before, only we will use a logarithmic confusion image since there is so much range in the number of training and testing samples. The three commands are (we have omitted the output)

```
(css)$ python naive_bayes.py -c log Bernoulli reuters21578-10
(css)$ python naive_bayes.py -c log Multinomial reuters21578-10
(css)$ python naive_bayes.py -c log TFIDF reuters21578-10
```

As before additional report files show up in our /soc290/reports folder. In addition to the 18 20 newsgroups files, we have three reports and three log confusion images for the Reuters data.

```
(css)$ ls -lao reports/
total 960
drwxr-xr-x  26 edward    884 Mar  3 18:05 .
drwxr-xr-x  26 edward    884 Mar  2 14:33 ..
-rw-r--r--   1 edward  30315 Mar  3 18:02 Naive_Bayes_Bernoulli_20news4_confusion.png
-rw-r--r--   1 edward    707 Mar  3 18:02 Naive_Bayes_Bernoulli_20news4_report.txt
-rw-r--r--   1 edward  30588 Mar  3 18:02 Naive_Bayes_Bernoulli_20news5_confusion.png
-rw-r--r--   1 edward    789 Mar  3 18:02 Naive_Bayes_Bernoulli_20news5_report.txt
-rw-r--r--   1 edward  41142 Mar  3 18:02 Naive_Bayes_Bernoulli_20news_confusion.png
-rw-r--r--   1 edward   3734 Mar  3 18:02 Naive_Bayes_Bernoulli_20news_report.txt
-rw-r--r--   1 edward  34674 Mar  3 18:03 Naive_Bayes_Bernoulli_Reuters21578-10_confusion.png
-rw-r--r--   1 edward   1429 Mar  3 18:03 Naive_Bayes_Bernoulli_Reuters21578-10_report.txt
-rw-r--r--   1 edward  30862 Mar  3 18:04 Naive_Bayes_Multinomial_20news4_confusion.png
-rw-r--r--   1 edward    709 Mar  3 18:04 Naive_Bayes_Multinomial_20news4_report.txt
-rw-r--r--   1 edward  32257 Mar  3 18:04 Naive_Bayes_Multinomial_20news5_confusion.png
-rw-r--r--   1 edward    791 Mar  3 18:04 Naive_Bayes_Multinomial_20news5_report.txt
-rw-r--r--   1 edward  42728 Mar  3 18:03 Naive_Bayes_Multinomial_20news_confusion.png
-rw-r--r--   1 edward   3736 Mar  3 18:03 Naive_Bayes_Multinomial_20news_report.txt
-rw-r--r--   1 edward  35138 Mar  3 18:05 Naive_Bayes_Multinomial_reuters21578-10_confusion.png
-rw-r--r--   1 edward   1431 Mar  3 18:05 Naive_Bayes_Multinomial_reuters21578-10_report.txt
-rw-r--r--   1 edward  29628 Mar  3 18:05 Naive_Bayes_TFIDF_20news4_confusion.png
-rw-r--r--   1 edward    703 Mar  3 18:05 Naive_Bayes_TFIDF_20news4_report.txt
-rw-r--r--   1 edward  30724 Mar  3 18:05 Naive_Bayes_TFIDF_20news5_confusion.png
-rw-r--r--   1 edward    785 Mar  3 18:05 Naive_Bayes_TFIDF_20news5_report.txt
-rw-r--r--   1 edward  42024 Mar  3 18:05 Naive_Bayes_TFIDF_20news_confusion.png
-rw-r--r--   1 edward   3730 Mar  3 18:05 Naive_Bayes_TFIDF_20news_report.txt
-rw-r--r--   1 edward  34270 Mar  3 18:05 Naive_Bayes_TFIDF_reuters21578-10_confusion.png
-rw-r--r--   1 edward   1425 Mar  3 18:05 Naive_Bayes_TFIDF_reuters21578-10_report.txt
```

5. Analyzing the Data and Preparing the Lab Report

We have now collected a good set of results to examine Naïve Bayes performance on some standard datasets. To prepare for the analysis and report, you should now assemble these data into a Word document with the report output and confusion images on the same pages. The results section of the lab report should have the following 9 data pages, titled as below. We will add our analysis and a cover page to the beginning of this document in the next section.

1. Data 1: Bernoulli, 20news
2. Data 2: Bernoulli, 20news5 and 20news4

3. Data 3: Multinomial, 20news
4. Data 4: Multinomial, 20news5 and 20news4
5. Data 5: TFIDF, 20news
6. Data 6: TFIDF, 20news5 and 20news4
7. Data 7: Bernoulli, Reuters
8. Data 8: Multinomial, Reuters
9. Data 9: TFIDF, Reuters

At the beginning of the report, before the data results pages provide the following analysis.

Performance Measures

Our reports summarize classifier performance in terms of precision, recall and the F1 score. Look up these statistics on the internet (or elsewhere) and in your own words provide a description of what each of these statistics measures.

Results Analysis

Summary: For each of the datasets 20news, 20news5, 20news4 and reuters21578-10, and models Bernoulli, Multinomial, TFIDF:

1. How would you characterize the performance Naïve Bayes in these 12 tests? Justify your argument with specific references to the average precision and recall and their distributions across document categories in each case. How is performance compared to blind guessing in the best and worst cases?
2. How does the distribution of errors as shown in the confusion matrix and full classification report (i.e. precision, recall and F1 for each category) change depending on the model and dataset?

Comparisons:

1. What impact does changing the problem by structuring the 20news data in the three different ways have on performance and error distribution? Provide your explanation or suspicions about why you make your observations.
2. Describe performance differences you see between the Reuters data and the 20news data. Explain why you think this might be so.

Lab reports and submission

When you have completed your report, title and export the file as soc290_lab2_your_last_name.pdf. Instructions will be given in class for how and when to submit your report.