COSC 2671 Social Media and Network Analytics

Lab Week 2

Machine Learning Revision

Learning outcomes:
- Revise using machine learning to solve problems
- Learn/revise how this can be done within Python
- Continue to familiarise with a Python IDE and/or (interactive) terminal

Requirements:
- A PC with Internet connection and Python installed (preference is version 3.X, but 2.7 is also okay, but unfortunately, we don’t have resources to provide support if there are version incompatibility issues)

Resources:
- This lab worksheet (available on Canvas)
- Associated code in lab02Code.zip (available on Canvas)

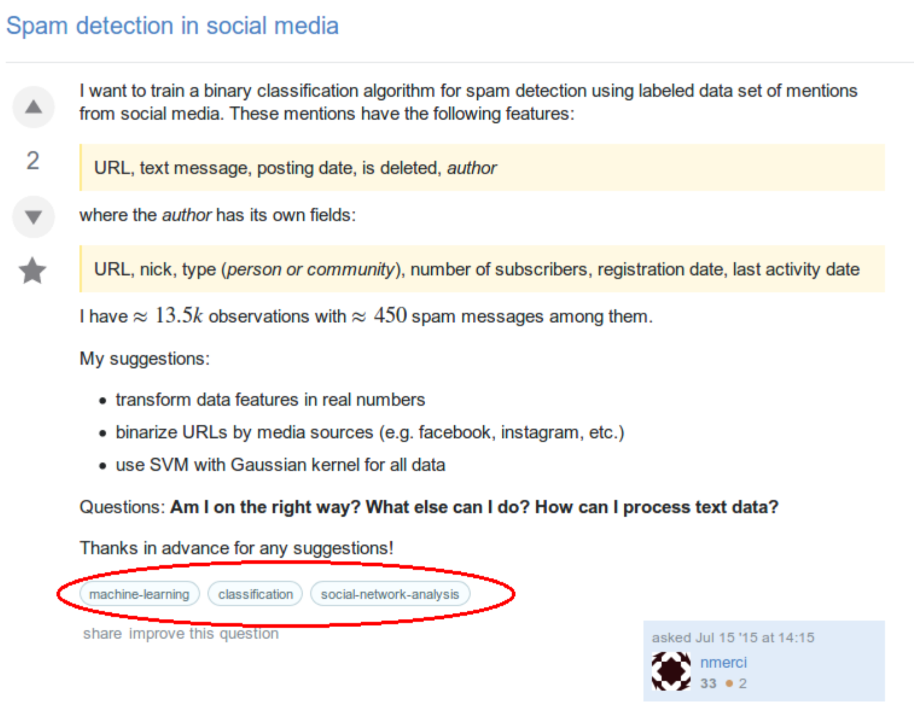
Python Packages Required:
- scikit-learn (python package name is ‘sklearn’)
- nltk
- numpy

Introduction

In this week’s class, we revise how to apply some machine learning algorithms. This week, we look at community question and answer sites. Stack Exchange is a good example of a community Q&A site, and if you had programming (and other) questions and searched on the web for answers, it is likely you will be very familiar with it.

One of the important pieces of information used to index, summarise and provide relevant answers are the tags assigned to the questions. See Figure 1 below for an example of a question posts and tags associated with it (in red circle).

Sometimes it is difficult to come up with a good set of tags to describe a question, hence tag recommendation is an important task for Stack Exchange. This also extends to other social media that asks users to tag their posts. In this lab, we will build a simple classifier that can learn to suggest tags for questions.


Spam detection in social media

I want to train a binary classification algorithm for spam detection using labeled data set of mentions from social media. These mentions have the following features:

URL, text message, posting date, is deleted, *author*

where the *author* has its own fields:

URL, nick, type (*person or community*), number of subscribers, registration date, last activity date

I have $\approx 13.5k$ observations with ≈ 450 spam messages among them.

My suggestions:

- transform data features in real numbers
- binarize URLs by media sources (e.g. facebook, instagram, etc.)
- use SVM with Gaussian kernel for all data

Questions: **Am I on the right way? What else can I do? How can I process text data?**

Thanks in advance for any suggestions!

machine-learning classification social-network-analysis

Figure 1: An example question post and associted tags in Stack Exchange.

Download data

Stack Exchange has an API, but for this lab we want to focus on the machine learning part. Hence, we are using the latest dump of the Data Science Stack Exchange (https://datascience.stackexchange.com/) which was collected up to the end of 2017. This data dump is available in the *lab02Code.zip*. Inside the zip file, there are two json files, Posts.json (the actual posts) and Tags.json (list of tags and the number of posts they appear in).

Initial Data Pre-processing

Unzip *lab02Code.zip* into your working directory.

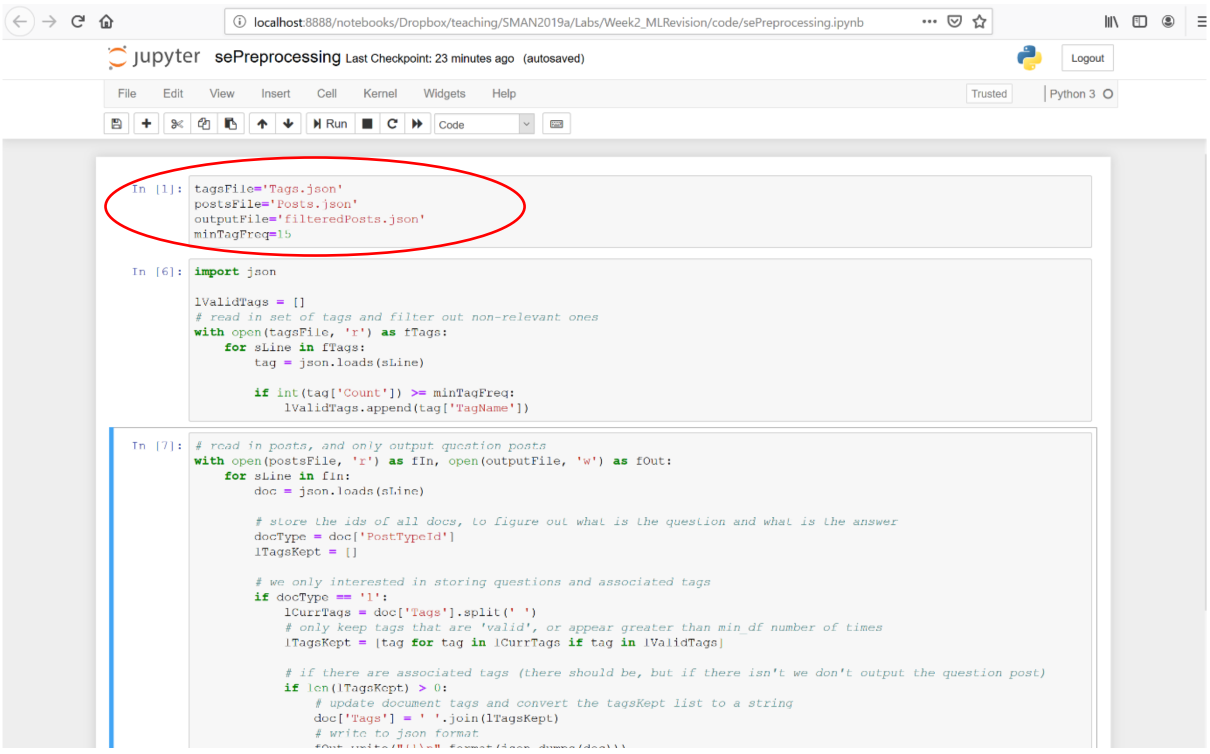
Open Posts.json. This is a very standard json file but observe two things. Look for PostTypeId and Tags. PostTypeId specify the type of post, where:

- PostTypeId = 1 is a question
- PostTypeId = 2 is an answer
- Others are not important for this lab

Tags are the list of tags associated with it. Note that only questions have tags associated with them.

For our data pre-processing, we want to only keep question posts, i.e., PostTypeId = 1, as these are the only ones that have tags. We also want to only train on and classify for popular tags, i.e, those that occur in a number of posts. We have constructed a notebook for that; examine *sePreprocessing.ipynb*. Given the Posts.json and Tags.json, this file will filter out non-question post and update their associated tags to those that are occur in a number of posts. Its output is in the json format.

Run all the lines in the notebook. Note that the first ‘line’ [1] has a number of commands that specify the different file inputs. If you decide to use this notebook for later, make sure you change these variables to reflect your own files.


localhost:8888/notebooks/Dropbox/teaching/SMAN2019a/Labs/Week2_MLRevision/code/sePreprocessing.ipynb
jupyter sePreprocessing Last Checkpoint: 23 minutes ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Code

```
In [1]: tagsFile='Tags.json'
        postsFile='Posts.json'
        outputFile='filteredPosts.json'
        minTagFreq=15

In [6]: import json

        lValidTags = []
        # read in set of tags and filter out non-relevant ones
        with open(tagsFile, 'r') as fTags:
            for sLine in fTags:
                tag = json.loads(sLine)

                if int(tag['Count']) >= minTagFreq:
                    lValidTags.append(tag['TagName'])

In [7]: # read in posts, and only output question posts
        with open(postsFile, 'r') as fIn, open(outputFile, 'w') as fOut:
            for sLine in fIn:
                doc = json.loads(sLine)

                # store the ids of all docs, to figure out what is the question and what is the answer
                docType = doc['PostTypeId']
                lTagsKept = []

                # we only interested in storing questions and associated tags
                if docType == '1':
                    lCurrTags = doc['Tags'].split(' ')
                    # only keep tags that are 'valid', or appear greater than min_df number of times
                    lTagsKept = [tag for tag in lCurrTags if tag in lValidTags]

                    # if there are associated tags (there should be, but if there isn't we don't output the question post)
                    if len(lTagsKept) > 0:
                        # update document tags and convert the tagsKept list to a string
                        doc['Tags'] = ' '.join(lTagsKept)
                        # write to json format
```

The vaiables have the following meaning:

- tagsFile is to specify the Tags.json input file
- postsFile is to specify the Posts.json input file
- outputFile is to specify the filtered tags and posts output file
- minTagFreq is to specify what is the minimum number of posts a tag must appear before it is considered a tag we will classify for. This assigned value is 15.

After running the script, examine the output file (filteredPosts.json) to see that only question posts have been stored, and only tags that appear in at least minTagFreq number of posts remain.

Note this may be slightly different to what you are used to, as we store the output so we can reuse it multiple times without having to do the filtering again and again.

Build a tag classifier

Now we have the data ready, we can proceed with building a classifier to suggest tags. We will be using the words in the posts and its title as the features, and the tags will be used as (classification) labels.

There is a few steps we need to do to build our classifier:

1. For each post, extract the words from the posts and title.
2. Construct a “document-word” frequency matrix
3. Apply weighting scheme to the matrix
4. Extract our labels
5. Using the document-word matrix and document labels (the tags), train our chosen classifier

Essentially what we are doing is to learn a model that associates the words in each post and its title (the features) with the corresponding tags with that post. When we are testing or using it to suggest tags for a new question, it will suggest tags based on the set of words in the question.

When processing text, it is standard practice (and required for many ML libraries) to put the document, associated words and their frequency into a document-word matrix, which is also what we do here.

To improve classification performance, sometimes it is necessary to reweight the word frequencies. One approach is TF-IDF (term frequency-inverse document frequency) – which we will revise/learn about in this week’s lectures and tutorials. The intuition is that it is not the most frequent terms that are most helpful for classification, but those words that appear in only a few documents (posts), which means the presence of these words are generally good indicators of the corresponding labels. We reweigh the frequencies to make such words have larger weight and hence more importance, while those words that are appear in many document/posts are weighted down to reduce their influence. For more details, have a look at https://nlp.stanford.edu/IR-book/html/htmledition/inverse-document-frequency-1.html

After this pre-processing, we can use the constructed TF-IDF weighted document-word matrix and the document/posts labels/tags to train a classifier and evaluate its performance.

We use cross-validation to evaluate its performance. We compute the precision, recall and F1-score, averaged across all the tag labels.

Implementation

We will use the *Scikit Learn* machine learning library, with some help from *nltk*, to help us do all this.

Open the supplied *seTagClassifier.ipynb* file. Each of the above steps have been implemented. Please read the file carefully and ensure you understand how the notebook implements each step (ask your friendly lab demonstrator if you unsure). Run the code. It may take a few minutes before there is output, unfortunately with Jupyter notebooks it is sometimes not clear whether it is running, but there are apparently widgets that can show progress.

Again line [1] specifies the variables.
inputPosts specify the file location of the posts we are training and testing the classifier on.
max_df and min_df are variables that specifies filtering cutoffs for words.
We will use the file outputted from sePreprocessing.py for this.

You should see something similar to the following (might be different numbers):

```
Average precision: 0.6498361098054307
Average recall: 0.2564069702472262
Average F1: 0.36758187712496215
```

Congratulations, you have built a tag classifier!

But the performance is a little low, so lets try some things to improve it.

Exercise:

Classifier: First task is to try a different classifier. Typically this will be a step after more pre-processing, but I want you to explore how to use different classifiers. Read up the documentation on how to use NaiveBayes and k-nearest neigbour classifers. Modify seTagClassifier.ipynb to first use the NaiveBayes (in scikit learn, look up MultinomialNB), then the k-nearest neighbour classifier (KNeighborsClassifier). Has it made any difference in the performance?

Number of tags: Second task is to try to understand why the classifier got such low performance from the data perspective. One typical reason is there are too many labels (tags) to classify, particularly with the number of posts we have. Hence, first thing is to figure out how many tags are in the data. Another issue could be the number of words we have.

How many tags are there? How many posts do we have? How many words?
Hint: After the document-word frequency matrix is built, you can determine its size, which can tell us how many posts and words we have. Scikit learn uses numpy arrays to represent matrices, and the 'shape' property is tuple indicating the dimensions of an array. E.g., A.shape returns a tuple (x,y), where x and y are the dimensions of A.

The number of tags is determined by the *minTagFreq* variable of sePreprocessing.ipynb. Change it to 10 and 20, construct new training and testing data using sePreprocessing. ipynb, and rerun seTagClassifier. ipynb. Does it make a difference? Why do you think it made or not make a difference? Discuss this with your lab demonstrator if unsure about answer.

Number of unique words: Typically we only want a few hundred unique words as features for the number of documents/posts we have. Recall from Practical Data Science that two approaches to reduce the number of features (words) is to do feature selection or projection. In this case, we will use feature projection, namely PCA, and project our word space to a much smaller number of dimensions. Consider the following segment of code to perform PCA on a document-word matrix X into 500 dimension space.

```
pca = TruncatedSVD(n_components=500)
pca.fit(X)
newX = pca.transform(X)
```

As an aside, we use TruncatedSVD for our PCA because X is a sparse matrix, which is the output matrix format from the vectorising classes and methods in Scikit learn.

Note: PCA doesn’t work with MultinomialNB, as that requires X to be non-negative. PCA can produce negative values.

Using this as an example and documentation at http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html, use PCA to reduce the dimension of our document-word frequency. Try 250, 500 and 1000 dimensions. What performance did you achieve? Which dimension number achieved the best result, and why do you think this was the case?