MULTI-LABEL CLASSIFICATION I

Jesse Read

10 October 2016

1 Introduction

The goal of this lab is to perform a classification task with multi-labelled data. The lab will follow a typical multi-label classification pipeline. We use data from the Internet Movie Database[1] (IMDb).

The material is supplied in the `code/` folder, including some skeleton code which should be completed during the tasks of this lab. The data for the tasks is provided in the subfolder `data/`. The end goal is to investigate using this data to build a model that can recommend (i.e., predict) labels for new film entries.

1.1 Requirements

PYTHON (recommended `3.5`) is used, along with NUMPY, MATPLOTLIB, SCIKITLEARN, PANDAS, and NETWORKX libraries.

2 The Data

Throughout this lab we will use the data from the IMDb website[2]. The `plot.list` and `genres.list` has been obtained (October 2016). From these two files, `imdb.csv` has been created, containing a vector representation genres (i.e., labels), and a raw text summary. Thus, the lines correspond to a text description of the plot of a film and the corresponding label vector, indicating relevance of 28 genres. The raw text summary can be converted to a bag of words representation by running `preproc.py`, which has produces a new file. For the tasks of the lab we consider the initial 50,000 films listed; corresponding to the file produced by this script: `imdb_bow_part.csv`, which is also included.

Note that in some of the figures in the following section, tasks are illustrated using the Music dataset, where pieces of music are associated with emotions. This dataset is also made available in the `data` folder.

3 The Tasks

The task is to build an appropriate multi-label model for the data, such that given new plot summaries, labels can be recommended. In a real-world application, such a model could be used to speed up and homogenize manual label assignment, or even carry out automatic labelling.

You will find that the file `lab1.py` already contains code to load the data into memory and split it into `X` and `Y` matrices, and also a hold-out set `X_test` and `Y_test` which we will use for evaluation. The

[1] `http://imdb.com/`

[2] Data obtained from `ftp://ftp.fu-berlin.de/pub/misc/movies/database/`

following tasks make use of this file, except the final task for which the file `lab1_extra.py` should be used. Some tools and multi-label classifiers are included in the `molearn/` folder. It will not be necessary to modify these, but their classes and functions will be called upon from the other files.

To get an idea of which multi-label learning algorithm is most appropriate for the data, we will take a look at some important statistics.

Task 1

Calculate the following:

- *Label cardinality*: average number of labels per example
- *Label density*: the proportion of relevant labels to non-relevant labels in the data
- The number of *unique label vectors*
- The *relevance frequency* of each label (i.e., the proportion of relevant examples to each label)

Using an array of the relative frequencies, we can get an idea of the overall distribution, which lets us know how susceptible we would be to class imbalance if we build binary models.

Task 2

With the vector of relative frequency of each label, calculated in the previous task, plot them in decreasing order. You should observe the distribution of label frequencies. In Figure 1b it is shown for the Music dataset.

The number of labels is the biggest factor of complexity in the binary relevance methods, but it is the number of *distinct labelsets* that has the biggest influence over the performance of the label powerset method (because it treats label vectors as single values). Therefore we are also interested in the distribution of 'classes' (label *combinations*), when the label vectors are treated in this way. Such a plot is shown for the Music dataset in Figure 1a.

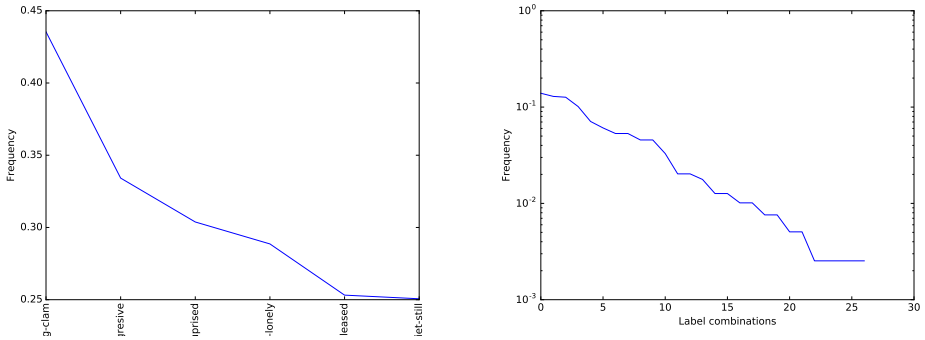
Task 3

Make also a similar plot (like in Task 1) but for the relative frequency of each label *combination* (each unique label vector).

We next take a look at the relationships among the labels. First, marginal dependence among the labels (ignoring the input space). We can make a simple analysis of linear dependence with a co-occurrence matrix.

Task 4

Create a co-occurrence matrix using *the top 10 most common labels* and visualise it as a heatmap using the function `make_heatmap` provided in `utils.py`. Value $C_{j,k}$ should contain the proportion of times the j-th and k-th labels occur together; 0 if they never occur together, and 1 if they always occur together.



(a) (b)

Figure 1: The horizontal axis shows the labels (Figure 1b) and label vectors (Figure 1b) and the vertical axis shows the corresponding proportion in the training set.

Task 5

Investigate the resulting graph for the IMDb data by using the co-occurrence matrix for the *top 10 labels*, it should help us visualise the relationship among labels.

The following code shows how to initialize and train a multi-label classifier (namely, the basic binary relevance method), using the code provided in the `molearn` folder:

```
from molearn.classifiers.BR import BR
h = BR() # note: logistic regression by default as base classifier
h.fit(X,Y)
Y_pred = h.predict(X_test)
#y_prob = h.predict_proba(X_test) # posterior probability / confidence outputs
```

A multi-label classifier $\mathbf{h}$ returns a vector $\hat{\mathbf{y}} = [y_1, \ldots, y_L]$ for each instance. Recall that, under binary relevance, $y_j = h_j(\mathbf{x}) + \epsilon_j$ for each $j = 1, \ldots, L$. We can measure conditional dependence by searching for dependence among $\epsilon_1, \ldots, \epsilon_L$, in a similar way to we did with the labels above.

Task 6

Finish the code in the `get_errors` function provided in the `utils.py` file, so that it returns a vector of dimension $N \times L$ containing the errors of each prediction. Then, use again the code from the previous Task 4 to compute and plot the same results, but on the *errors* of each classifier, i.e., the distribution of errors among labels, thus producing the *conditional* correlation matrix.

So far we have looked at errors for each label individually. However, to evaluate multi-label classifiers we are interested in an overall score for each multi-labelled example. For this we can use Hamming loss, exact match, or Jaccard index (among many other possibilities).

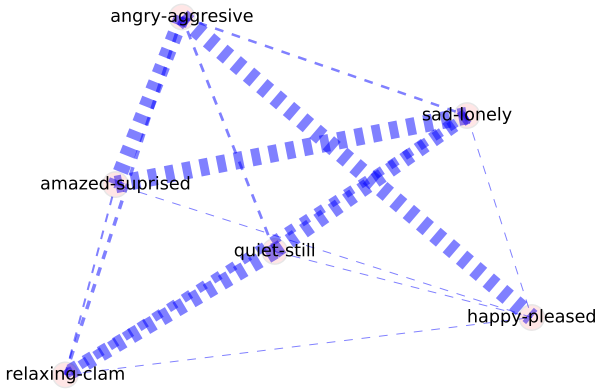


Figure 2: Label co-occurrence in the Music data, represented as a graph.

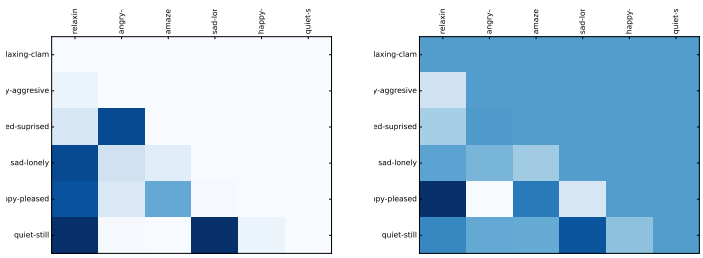


Figure 3: Estimations of marginal dependence (left) and conditional dependence (right) as heatmaps.

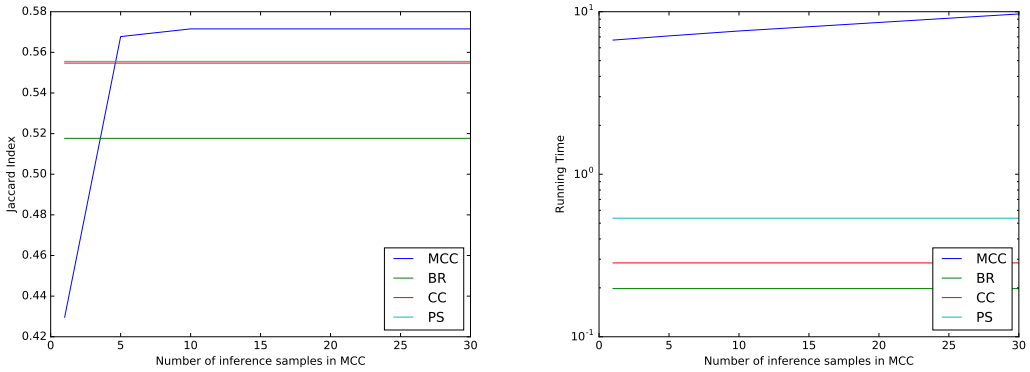


Figure 4: Demonstrating the higher accuracy (in terms of Jaccard Index) with respect to the binary relevance method, and the possible tradeoff involved in obtaining this accuracy – on the Music data.

Task 7

Instantiate and train `BR`, `CC`, and `PS(p=20)` classifiers. Evaluate each on the test data, and make a note of their accuracy under Jaccard index, exact match, and Hamming loss, and also total (train + test) running time (use the `clock()` function to time them).

You should observe that the predictive performance can be obtained that is better than the baseline binary relevance method. However, this performance advantage may often come with a penalty in terms of running time. We can always do more iterations of inference (search in the label space), add more or improved structures, and add more members to an ensemble, etc. (depending on the model). But it is important to be be conscious of how much computational trade-off is involved.

Task 8

Instantiate and train the classifier chains model with Monte-Carlo sampling for chain inference (`MCC`) and then, using a loop, make predictions with 1, 5, 10 and 30 samples (inference iterations) per instance (you can simply run prediction again, for example using `mcc.predict(X_test,M=5)` for 5 iterations, and so on ...). Make two plots, the first to contain the predictive performance (in terms of Jaccard index) vs the number of iterations, and the second to contain the running time (also vs iterations on the horizontal axis, but a logarithmic vertical axis). In both, plot the results of the three methods experimented with just previously, horizontally as a baseline (a flat line). In all cases use the hold-out data (`X_test`, `Y_test`) to test and evaluate the model. Figure 4 shows the result on the Music dataset.

What do you observe? We would expect the running time for MCC to increase linearly with the number of iterations, but the returns (in terms of predictive performance) on the computation investment, is trailing off logarithmically. Thus we see clearly the trade-off involved in multi-label classification (similar to classification in general). We are always limited by information contained in the feature representation and will not always be able to obtain a better model by investing more computational resources.

```
p(Romance|x) = 0.762
p(Comedy|x) = 0.626
p(Drama|x) = 0.603
p(Short|x) = 0.275
p(Music|x) = 0.042
p(Musical|x) = 0.032
...
p(Reality-TV|x) = 0.001
p(Sport|x) = 0.000
p(News|x) = 0.000
p(Talk-Show|x) = 0.000
p(Film-Noir|x) = 0.000
p(Game-Show|x) = 0.000
```

Figure 5: Example output from a novel plot summary.

Task 9

Open `lab1_extra.py`. Use the function `fetch_and_proc` (from `utils.py`) to load and preprocess the data from plain text into feature vectors. This function returns the dataset as matrices `X` and `Y`, the `CountVectorizer` that was instantiated and used to vectorise the text, and the `labels`. First, instantiate and train an `MCC` classifier (default parameters will do). Then be creative and write a brief plot summary for a film (in English). Use the already-instantiated `CountVectorizer` to vectorize this result, as follows:

```
xtest = tf_vectorizer.transform([my_plot_summary]).toarray()
```

where `my_plot_summary` is a string containing your plot summary. Obtain a prediction for this instance using the `predict_proba(xtest)` function to obtain posterior probability scores for each label. Investigate these outputs, making use of the names of the `labels`. You might obtain something like in Figure 5.

Pass the probabilistic output of the previous task over a threshold of 0.5 to obtain a set, e.g., $\{\texttt{Drama}, \texttt{Romance}\}$. Then experiment with different thresholds, e.g., 0.1. Observe the difference.

Hopefully the result is subjectively good enough to suggest that the model might be useable to recommend genres for new films (or even other pieces of text). This depends, of course, on what kind of text you wrote and the power of the vector space representation of the input, as well as the chosen multi-label model.