Automatic detection of learning needs using Machine Learning methods

Timothée Lottaz
Under the supervision of Prof. Pierre Dillenbourg
Optional semester project (8 credits)
CHILI Lab - EPFL
Lausanne, Switzerland
name.surname@epfl.ch

***Abstract*—Online educational resources have been developing a lot in the past years and it has become easier to follow distance courses & acquire knowledge whilst sitting in front of a computer. This project aimed at easing the detection of current and upcoming sources of online knowledge by building an automatic tool for course webpage classification, using Machine Learning techniques. The tool was provided with a manually constructed training set of labeled webpages and was then improved with a semi-automatic collection algorithm. Despite the difficulty of the task and the small size of the training set, the tool achieved a final accuracy of 0.726 for differentiating between online knowledge and other types of webpages. Finally, potential applications of the tool are presented in the context of online educational resources detection.**

I. Introduction

A. Motivation

The recent development of online educational material (such as videos, recorded lectures, *MOOCS*) has made *Distance Learning* possible and widely spread. The growth of online learning material is fast, and new platforms are emerging frequently. The technological shift that is happening in education is an interesting topic that leads to several questions: it would be for instance interesting to try to to analyse the *Skill Gap* between the educational resources available online and the skills that are needed in the professional world. Knowing which courses exist on the web is not a trivial task, and this projects aims at creating an automatic tool to help listing these online resources, and possibly detect new ones in the future.

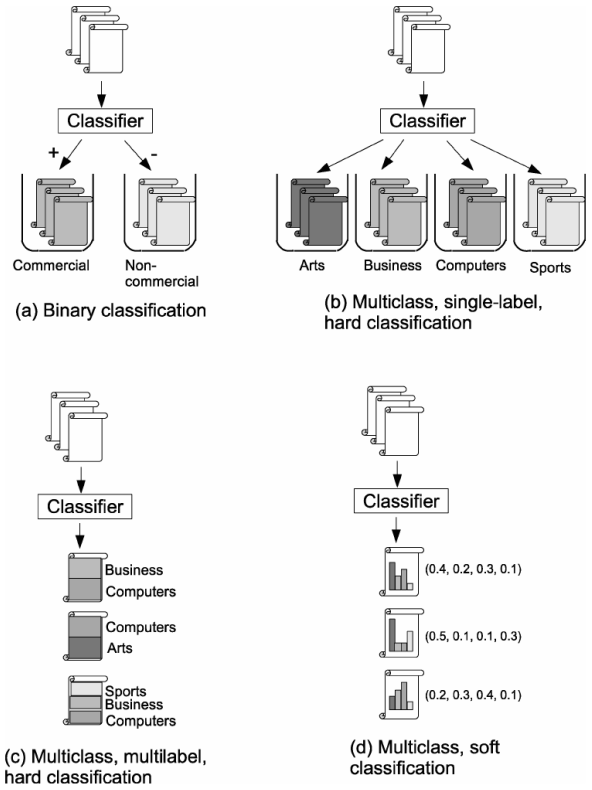

Fig. 1: Types of classification

B. Choice of the tool

Automatic Webpage Classification is a technique that aims at automatically analysing the content of an online page, and assigning it to one or multiple labels (binary / multiclass classification). This assignation can be hard or soft (with weights). This technique is for example used to create *Web Directories*, a listing of webpages with automatically associated categories, such as "Article", "Blog", "Video", etc. Different types of Webpage Classification are discussed in *Web Page Classification: Features and Algorithms* [1], and are shown in Figure 1.

I decided to get inspiration from this technique to build a classifier of my own. The goal is to take as input a webpage and give as output a hard, binary result, telling if the page contains an "online resource" or not (this definition of "online resource" is made more precise in section II).

Then, as we can see in section VI, this classifier can be used to list the available online courses or course platforms.

C. Code requirements

For this project I used python with some of the most famous Data Science libraries:

- `scikit-learn` for Machine Learning tasks, because it is fast to get a working prototype and compare different techniques. It is not the library with the best performances, but it did not matter because of the very small training size.

- `Pandas` to represent data with its useful DataFrames.
- `Networkx` to manipulate graphs and represent the links between webpages during crawling.
- `BeautifulSoup` for webpage parsing.

The complete list of requirements lies in `requirements.txt`. The `README.md` file explains the details of how to install the project.

D. Code structure

The code is available here: *https://github.com/timozattol/online-skills*. I used different Jupyter Notebooks to build iteratively each part of the project, and I reported in separates `.py` files the functions that I decided to keep, in order to be able to import them directly in the next notebooks. The `.py` files are documented. Here is the list of the files with their purpose:

- `01_pages_preprocessing.ipynb`: download a webpage, extract visible text, extract content of html tags.
- `02_simple_classifier.ipynb`: first classifier (Random Forest), asynchronous page download
- `03_feature_model_comparison.ipynb`: use structural features, compare different classification models.
- `04_nlp_classification.ipynb`: use Natural Language Processing features for classification
- `05_crawler.ipynb`: web crawler that constructs a graph of the webpages with the classifier prediction values.
- `06_graph_viz.ipynb`: graph exploration & visualisation
- `07_user_submissions.ipynb`: full week experiment of web crawling & user verdict to augment training set.
- `08_applications.ipynb`: examples of classifier applications.
- `crawler.py`: final implementation of the crawler
- `features.py`: functions used for building structural & NLP features
- `helpers.py`: file manipulation functions & other helpers
- `webpages/`: folder containing the urls of the webpages (training set), their cached version, and the results of the week experiment.
- `saved/`: folder containing the saved web graphs and trained models

II. Training set construction

Even though it was not the most exciting part, training set construction was at the core of this project and it took a lot of effort to be build, during multiple iterations. Human collection is very subjective and prone to bias as explained in [2], so a clear definition of an "online resource" and procedure for collection were defined in order to sample as uniformly as possible.

A. Definition of an "online resource"

An educational online resource can take multiple forms. Some are very formal (MOOCS, courses, programs, degrees, ...) and some are quite informal (Youtube videos, drawings, notes, ...). It seems like an almost impossible task for a computer to guess all kinds of online resources, so I created a list of criterias for a webpage to be a positive example that would make the task as feasible as possible:

- The webpage must represent a **formal course** about a specific subject (Geography, Management, Programming, ...)
- The webpage must be **the homepage** of the course (it for instance contains an "Enrol now" button, describe what the course is about, the teacher, the gained skills, but does not contain the course material)
- The course must be **online**. Pages describing a physical university course are not considered as positive.
- Any other webpage is a negative example, even from the same website (a list of courses such as "all courses about C++" are considered as negative examples, even if it usually links to many positive examples).

This definition is convenient, because one can imagine some common properties for all the positive examples, such as structural properties (presence of a button, of an introduction video, of a photo of the teacher, ...) or content properties (usage of the words "enrol", "skills", "teacher", ...).

Also, these pages usually contain a description of the obtained skills, and it could be interesting for the *Skill Gap* analysis mentioned earlier in section I-A.

Note: after setting this definition, we can already feel that differentiating between a course page and a list of course is going to be "hard", and differentiating between a course page and an article is going to be "easy" (or at least easier).

B. Collection procedure

Usually, *Webpage Classification* is trained on big datasets containing millions of labeled examples. Unfortunately after some research, no labeled dataset exists in this context because a "course page" is something quite specific and no-one took time of collecting samples. So I had to construct a small dataset by hand, which made a big difference in the Machine Learning pipeline since it implied working with a very small training set later on.

As mentioned earlier, the difficulty is to avoid bias due to human collection. The negative samples must represent as well as possible the entire web except the course pages. I thus tried to diversify as well as possible the types of webpages by collecting articles, videos, blogs, shopping sites, image platforms, discussion platforms, etc.

For positive examples, I first wanted to focus on big platforms (`coursera`, `edX`, ...), and download all their course

lists as it would quickly yield a lot of examples and a decent sized training set. But after some thought, the classifier would be biased towards those platforms, and it would be impossible to discover new small platforms. Also, since it is easy to download their course list in the first place, the classifier would be useless. I thus decided to take an approach where big and small platforms are considered the same.

Since finding a new platform is a time-consuming task, and is limited by the tendency of search platform to give the "top answers", I have to take multiple examples from the same platform. I thus took 6 examples in each positive platforms (for example 6 different courses of coursera). To balance, I took 6 examples in each negative website (for example 6 articles from the cnn.com website). I made sure that the 6 examples are about different subjects.

Also, as we want to be able to solve the "hard" problem of differentiating between a course page and any other page of a course platform, I added 6 examples of "other page" for each course platform (for example the terms and services page of coursera).

It took multiple days, and multiple iterations (I increased the set twice), but it is an increasingly hard task as popular platforms stand out and smaller platform get more and more difficult to find. You can find in Table I the final list of all positive and negative domains.

Note: each positive domain is also represented by negative examples, as explained earlier. Also, in the actual training set, samples from these domain appear, for example `https://www.coursera.org/learn/matlab/` for the `coursera.org` domain.

Note: some famous platforms such as `edx.org` are not listed because their content is generated after page load, and the script downloads the same empty page for each sample.

Positive domains	Negative domains
coursera.org	sbb.ch
codecademy.com	edition.cnn.com
openuniversity.edu	nba.com
online-learning.harvard.edu	youtube.com
udemy.com	vimeo.com
openclassrooms.com	dailymotion.com
conted.ox.ac.uk	pinterest.ch
open2study.com	scholar.google.com
alison.com	reddit.com
deakin.edu.au	imgur.com
online.stanford.edu	google.ch [images]
class-central.com	ebay.com
kadenze.com	wikipedia.org
futurelearn.com	amazon.com
skillshare.com	economist.com
iversity.org	independent.co.uk
generalassemb.ly	
learnopia.com	
creativelive.com	
pluralsight.com	+all positive domains
newsu.org	
trainingcenter.com	
vtc.com	
156 samples	218 samples

TABLE I: Domain name of positive & negative samples

III. Performance analysis

A. Cross validation

First, I did a cross validation on the training set with 3 folds. I found it important to perform a cross validation because of the small size of the training set: the performance would highly depend on the split. But the performances were too good to be trusted, and I figured that since each train/test set are sampled from the same "bucket", it is very probable that two examples from the same platform end up in both the train and the test set. The good results could then be explained by the fact that examples from the same platform are similar, leading to over-fitting.

B. Easy-Hard test split

I thus decided it was wise to forget about doing a cross-validation, and do a simple split. The domains used in the test set are then never used in the train set, and vice-versa (all 6 examples of a domain either go to training or to testing). Also, since the separation is always the same, different methods & models can be compared without worrying of the randomness of the split.

Also, I took this occasion to evaluate separately the "easy" and the "hard" task, by creating two negative test sets: the easy one containing the non-course domains such as `sbb.ch` and the hard one containing the non-course pages from the course platforms, such as the main page of `coursera.org`. The testing set separation is represented in Figure 2. A better result is expected for the easy test.

The results were worse with the Easy-Hard method, but hopefully more realistic.

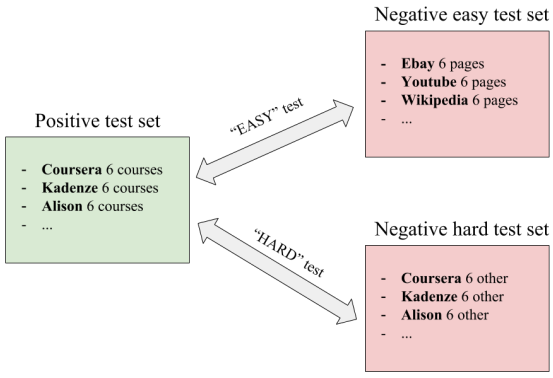

Fig. 2: Easy-hard test split

IV. Feature choice

The webpage classifier takes as input what we call *features*, that are then used to make a prediction. I tried two different approaches to translate a webpage into *features*, which are presented here.

A. Structural Features

The first idea was to use *structural features*. The motivation behind this is that a webpage is more structured than some text in a book: it contains html tags, images, buttons, links, etc. These elements can bring information about the content of the page, and one can imagine that course webpages have some structural patterns in common.

I used several features in conjunction, that simply counted the occurrence of html elements: number of links, number of videos, number of buttons, number of iframes (embedded documents), and I trained a Random Forest classifier. I expected the number of buttons and the number of videos to be the most representative, since there often is a single "Enroll" button and a single "Presentation video".

After comparing different models and applying a Grid Search over the hyperparameters, the best cross validation accuracy was reached by a K-Nearest-Neighbor classifier and was of **0.75**, with also a f1-score of 0.75, which is quite impressive for such a difficult task. Although, as discussed in III, this is probably over-fitting: the same platform often has exactly the same number of links in all their course pages.

This was proven right by our second performance metric: the Easy-Hard test. Indeed, with this metric the accuracy went as low as **0.45** for the easy test, and **0.52** for the hard test, so as good as a random guess.

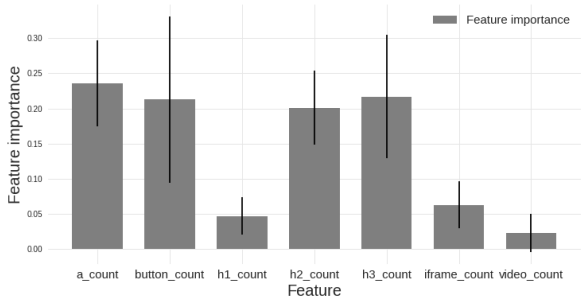

Fig. 3: Features importance

Figure 3 shows the feature importance, values that sum to 1.0 and show which features are the most used by the classifier. As we can see, `a_count` (the count of outgoing links) is one of the most relevant feature. After checking, its value is almost constant on different courses from the same platform, which is not good: our classifier is going to over-fit on the platforms in the training set. I tried to think of more complex structural features, but the same problem would also arise: all structural features are very platform specific. I could not find something valid across all course platforms, so I jumped to something more promising: Natural Language Processing features.

B. NLP Features

Analysing the textual content of a page looked more promising. My motivation was that if the structural content is very platform-specific, the text used to describe courses is probably more general: after looking at a lot of course pages during collection time, it seemed that the same vocabulary were often employed, and that some words can be found frequently such as "enroll", "program", "lesson", "pass", "skills", etc. These words occurring together would sound like a great sign of a course webpage.

First, I extracted the visible text as we do not want all the platform-specific code & tags to favor over-fitting. Then, one simple method would have been to use the count of each visible word as a feature, but I used a more powerful statistic called TF-IDF (Term Frequency - Inverse Document Frequency). TF-IDF not only counts the frequency of each word, but it reduces the importance of the words that occur all the time such as "the". This way, each webpage is described by the words that characterize this page the most, compared to the rest of the pages.

After comparing models, the K-Nearest-Neighbors classifier worked less good with these features, and anticipating further improvement I used a Logistic Regression. First, it performed well with an accuracy of **0.72** on the easy task and **0.61** on the hard task. Second, it is able to give its `decision_function` which gives a confidence score to the predictions, that will prove very convenient for improvements of Section V-B.

V. Training set improvement

A. Crawler construction

These scores were starting to look ok but were still not very impressive: 0.61 for the hard task is close to a random guess. I thought it was mainly due to the small size of the training set: Machine Learning is usually a good tool when we have a lot of examples. Also, manual collection has its limits, and as I struggled finding new platforms by hand, I designed a way to automatically find new pages and iteratively improve the classifier.

I designed a *crawler* that performs a *BFS* (Breadth-first search), starting from a given webpage (root). The *BFS* then randomly follows links to other pages, goes on iteratively from these new pages and creates a graph of all the visited pages. I can choose the maximum depth of the graph, and how many links to follow from each page. The running time and size of the graph depend exponentially on these parameters. The pre-trained Logistic Classifier is used so that each node of the graph contains: the URL of a page, the True / False prediction, and the `decision_function` (confidence of the prediction).

You can see on Figure 4 two runs of the *BFS* on two unseen platforms: `tutsplus.com` and `medium.com`. Each run lasted approximately an hour for a max-depth of 6 and a breadth of 3 pages. The nodes in green were classified positively, the ones in red negatively, and the grey nodes could not be downloaded properly. I chose these platform because the first one is a course platform and the second an article platform. I thus expected the first one to lead to many positive examples, and the second one to maybe lead to a few, after enough time. At a first glance, the first one (`tutsplus`) led to much more True predictions as the second (`medium`). Also, one can observe that `tutsplus`'s pages are much more densely linked together.

tutsplut: after a manual check, out of the 684 classified nodes, 86 were considered positive. Of out those, most were false positive, except 8 real courses.

medium: after a manual check, out of the 779 classified nodes, the 20 considered positives were all false-positive. 19 of them were articles that managed to fool the classifier with words such as "learning" or "skills". But interestingly the 20th was a list of educational resources: (*http://www.openculture.com/free_k-12_educational_resources*). A few additional steps of the algorithm would maybe lead to actual course pages! But these steps are exponentially expensive.

We already saw here a glance of a possible application of the classifier to find new resources; see Section VI for more. Also, the confidence score of the positive examples were quite close to zero, which means that the classifier is still unsure about its predictions. In next subsection I tried to take advantage of the crawler to improve this.

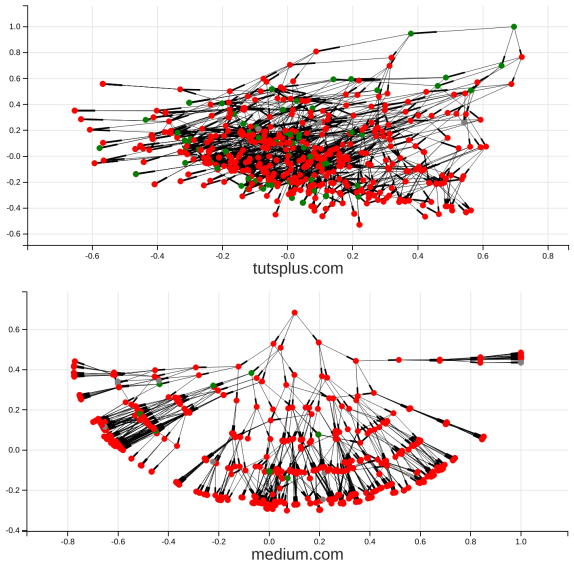

Fig. 4: Discovery graph crawling from two unseen platforms.

B. Automatic improvement

I did a full-week experiment to try to improve the classifier, by following each day these 5 steps: 1. Run a *BFS* for 1-2h during the night, 2. submit the predictions to be verified by a user, 3. add the verified examples to the training set, 4. train a better classifier to be used during next *BFS*, 5. check accuracy of the new classifier with the Easy-Hard metric.

I had to choose which webpages to start the *BFS* from (roots). With enough time, computing power, and user verification time it would be possible to start from anywhere (for instance the `medium.com` site) and find a lot of true / false examples. I had to choose my starting points intelligently to be able to gather examples in only a few hours, with a minimal user verification, so I used Google Search's API and constructed queries formatted as `Online Course <field>`, with `<field>` one of {`Mathematics, Science, Health, Art, Music, Dance, Leadership, Algebra, Life Science, Social Studies, Geography, Programming, History, Biology`}, and started the BFS from 5 random pages part of Google's top results.

Each run traverses on average 780 webpages, which is a lot to verify, so the initial idea was that the user verifies in priority the nodes with a confidence score close to 0.0. In practice, the classifier was really good at rejecting negatives, but it made mistakes for the high-confidence True predictions. So, to make the most impact I first verified the highest confidence scored pages and labeled the false-positives (very meaningful error). The positive examples were usually from the same platform, and it would pollute the training set to add all of them, so I tried to keep the number of new positives per platform close to 6. This important criteria unfortunately made the final number of addition per day very low. Also, three of the runs did not produce any new positive examples because the platforms were already known or the course were not available online (university syllabus), so they did not match the definition.

The final number of added samples was quite disappointing: only 5 days over 8 brought actual data, with on average only 20 new examples, found after an average of 20 min of user verification time. This made the training set size jump from 190 samples to 292, which is far lower that expected. Although, as you can see on Figure 5, these new additions did affect a little bit the easy task, and the classifier improved significantly for the hard task, probably because we mainly added new examples from course platforms and not a lot of other webpages. Final accuracies: **0.755** for the easy task, **0.726** for the hard task, which are in the end quite good for a task of this kind.

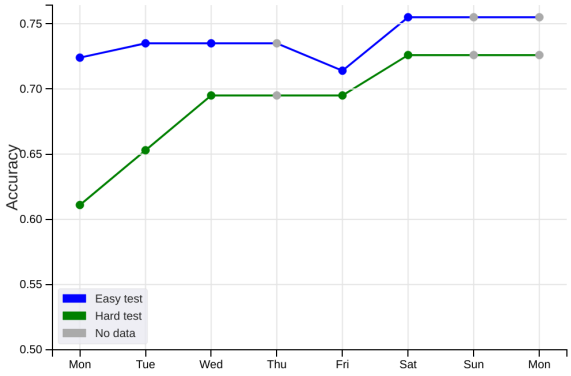

Fig. 5: Week experiment improvements

VI. Possible applications

Let us discuss potential applications of the tool that has been built throughout this project. Each of them requires some level of human intervention and the goal is to optimize the ratio of found resources over human verification time.

A. Random root crawling

The easiest application would be to start from any webpage (root) and run the crawler for a sufficient amount of time to be sure to traverse different domains & platforms and encounter

many educational resources. The problem is that the running time of the algorithm is exponential in the depth and breadth of the graph. Also, the total number of "course pages" is very low compared to the total number of other webpages on the web, so the probability to find one randomly is low. The human intervention time will be high, even if we only look at the positive predictions with the highest confidence scores, we might end up with a lot of false positive to manually exclude.

The crawling from Section V-A on the `medium` emulated this procedure, since `medium` is a webpage that has in principle nothing to do with education. 1h of running time was not enough to find a single course page (but we assume we were close).

B. Google search root crawling

To make sure to obtain educational resources quickly, we can start the crawling at a set of specific webpages that are the result of a Google Search. Section V-B already used this technique, but it ended up being a little disappointing: it did find a lot of positive examples, but usually from the same platforms: Google results are skewed towards the biggest platforms such as `coursera` or `edX`.

Potential improvements : 1. Start from Google results with a worse ranking, to be able to find new & smaller platforms or 2. Start with more roots, to diversify the traversed platforms.

C. Match-then-backtrack

I thought of this method making the assumption that courses are usually listed by a *hub page*. This proved right for most platforms I encountered so far, there usually was a "course list" page that pointed to each courses of a category. *Match-then-backtrack* would use this assumption to crawl fast through domains that are not course-related, and crawl slower on domains that seem to be containing courses.

One way of implementing this would be to start at a random root like in Section VI-A, but to only follow one link on each traversed page, so that the complexity is **linear** in the maximum-depth and not exponential anymore. This would allow for a very fast web traversal (we would quickly jump from domain to domain). Then, as soon as a positive prediction is made, we backtrack a few nodes and start crawling broader. After we are done with the found domain, we continue to follow one link on each page to find other domains.

We can also automatically detect potential *hubs* (they point to a lot of positive-predicted webpages) and follow every link from these pages, as they likely are course pages.

For an even more advanced version of this, one could implement the HITS algorithm (Hubs and Authorities) that bases itself on the same assumptions: it would then treat the course pages as *Authorities* and the course lists as *Hubs*.

D. Comparison

On Table II, you can find an informal comparison of the three applications, with the advantages marked in bold:

By *Running time*, I mean how much time is needed to find a fixed amount of new positive examples. By *Human intervention*, I mean how much time must a human spend on validating the output of the method. By *Platform diversity*, I mean how many different course platforms are going to be discovered in a fixed amount of time.

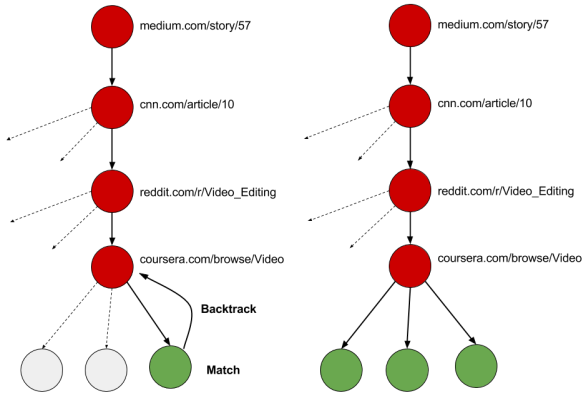

Fig. 6: Match-then-Backtrack

Method	Running time	Human intervention	platform diversity
Random root	very long	high	moderate
Google search root	**very fast**	**low**	low
Match-then-Backtrack	**fast**	moderate	**high**

TABLE II: Comparison of the three applications

VII. Conclusion

With this project, I was able to build a webpage classifier that can differentiate between what we defined as an *online resource* and any other type of webpage. Both page-structure and NLP features were used, NLP ending up being the most efficient one. I underestimated the time it would take to manually build a quality training set which ended up being a lot smaller that expected. But the semi-automatic method used for improvement worked: although it didn't meet my expectations on the size augmentation of the training set it nonetheless improved the accuracy on both the easy and the hard task to reach final accuracy scores of respectively **0.755** and **0.726**. Finally, three potential applications of the classifier were presented and informally compared, including the promising *Match-then-Backtrack* method I invented.

Acknowledgment

I would like to thank Fu-Yin Cherng that followed the project before it was changed to its current version, and Prof. Pierre Dillenbourg who took over its supervision for the whole semester.

References

[1] X. Qi and B. D. Davison, *Web Page Classification: Features and Algorithms* Lehigh University, 2009, http://bit.ly/2vhSELY.

[2] N. Zhong and J. Liu, *Intelligent Technologies for Information Analysis* Springer, 2004, http://bit.ly/2D2hu87.