Document classification and prediction using an integrated application

Business objective

Process Optimization

Text classification is one of the significant tasks in **Natural Language Processing**. It is the process of classifying text strings or documents into different categories, depending upon the contents of the strings. Text or document classification has various applications, such as detecting the research areas of scientific papers, classifying an email, classifying blog posts into different categories, automatic tagging of customer queries, etc.

The project we will illustrate here is an application system that allows us to classify many types of documents based on Deep Learning/Machine learning Natural Language Processing (NLP) models. In a second stage, the application will predict the class of new coming unlabeled documents. For optimizing workflow, processing, and usage, we conceived the current project as two separated Flask RESTful APIs coupled with a user interface (using CSS/HTML and JavaScript):

1. **Documents classification and prediction application:** this app contains the entire project workflow, including the training of the classification models based on a labeled documents dataset and, in the second phase, the prediction - based on the previously trained model - of new coming unlabeled documents. The app can run on a local machine and a virtual machine with GPU capabilities (required for training the models).
2. **Documents prediction application:** the second app contains the document prediction workflow and we created it as a web app multi-label able to run everywhere. For the app to run, the specifications of the previously trained model (e.g., model's weights and biases) need to be uploaded.

Note: The dataset we present to the system should be a directory containing all the documents with subdirectories as label names. In the case of multi-label document classification (i.e., when a document can belong to more than one category), you should duplicate multi-labeled documents in all the subdirectories they belong.

Project features

We will outline the five key document classification project features:

1. Accommodate a wide variety of documents: including native document formats as well as scanned documents/images (using OCR);

2. Accommodate a wide variety of datasets: kind of 'one-size-fits-all' models able to deal with binary, multiclass and multi-label classification;

3. Both naive and expert users can use it: ease of use (user-friendly interface) but providing technical information for improvement (e.g., performance metrics, confusion matrix, ROC curve,...);

4. Good performing: looking for good/state-of-the-art practices (e.g., BERT deep learning transformers model);

5. Able to run everywhere: partially deployed in the cloud (second app with the prediction workflow).

Choosing the right models for document classification

One of the biggest challenges of the current project was to construct classification models that could accommodate a large variety of dataset types (for binary, multiclass, and multi-label classifications). That could provide good performance even in realistic (resources-limited) circumstances. Indeed not all potential customers will necessarily have huge datasets of high-quality documents to train the models and serve good predictions further.

At the same time, we made an effort to submit high-quality data to the models and use the most appropriate text cleaning/preprocessing methods for the specific task of classifying documents. NLP indeed covers a wide variety of functions (i.a. translation, text completion, topics modeling, summarization, Named-Entity-Recognition,...), and not all text cleaning/preprocessing methods are appropriate for all NLP tasks. For example, in the current project context, Part-Of-Speech tagging was not relevant as document classification does not rely on word order or sentence structure. Moreover, in some cases, we should tailor text cleaning/preprocessing methods to the type of task and the specific model used.

Options of document classification models

Some recent pre-trained text models (such as BERT, Roberta, or XLNet) have their own tokenizer/vectorizer, and using standard vectorizer (such as TF-IDF) is not relevant. In the same vein, stemmatizing words are not applicable when using BERT (BertForSequenceClassification). BERT's vocabulary is composed of real words (and sub-words) that won't match a substantial part of the stemmatized words.

We have browsed the literature looking for best practices and state-of-the-art Text Classification and Cleaning methods for all these purposes. In the preliminary phase of the project and with this in mind, we have pre-tested several potential good candidate models from both machine learning frameworks (i.e., gaussian naive bayes, support vector machine, XGBoost) and deep learning frameworks (i.e., BERT, RoBERTa, XLNet). After this pre-test phase, we retained two models for the following reasons :

- **Deep learning BERT** (BertForSequenceClassification) model provides the best performance amongst all tested models and does not tend to overfit even with an increasing number of epochs (as opposed to RoBERTa and XLNet);
- **Machine learning XGBoost** (XGBClassifier) model provides excellent performance and is much less demanding than BERT in computing capabilities (run therefore much faster).

Confirming model choice

To confirm our choice, we have tested these two models with various (type of) datasets. For example, trained on the DBPedia dataset (a dataset composed of 342,782 Wikipedia articles with hierarchical categories), BERT provided an f1-score of 99% on the first-class level (9 categories) and of 96% on the second level class (70 categories).

Keeping in mind the need for dealing with a realistic business context, we have also tested the model on two pdf datasets of reduced size (one multiclass, and the other multilabel) and specially built up for the project purpose and taken from the European Parliament think tank. The documents concern the EU legislation and are sorted in various fields (e.g., culture, environment, economic and monetary issues, etc.).

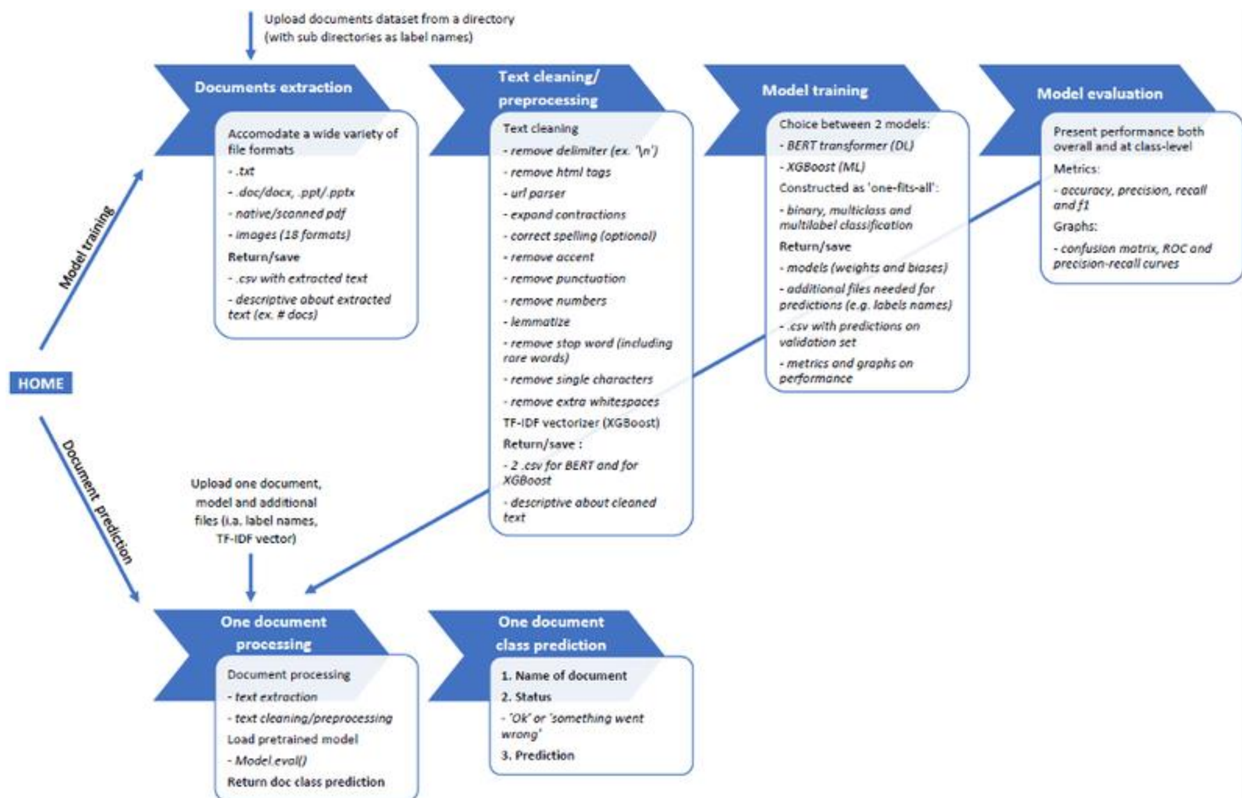
We have intentionally chosen to build datasets of reduced sizes to challenge the model's performances. The multiclass dataset was composed of 261 pdfs, and the multilabel dataset has 313 pdfs with 33 documents multi labeled (with 6

categories and less than 60 documents per category). The BERT model provides an f1-score of 97%, while the XGBoost model gives an f1-score of 92% when trained on the multiclass dataset.

For now, we will now discuss our workflow for our document classification project. Firstly, the app's home page automatically pops up when the user executes the main.py file.

As illustrated in figure 1, we present two options. Either the user can run the whole process from training models to predict new coming documents. Or, the user can only use document prediction (based on the specification of a previously trained model).

Figure 1. *The overall workflow of the document classification and prediction application*



Python programming for document classification

1. RESTful Flask API

We built two APIs for this project, running concomitantly and for different purposes.

The main API acts as a pivotal framework. It coordinates the whole process and workflow and allowing smooth communication between the user interface and the python objects.

The secondary API is in the main.py file. It retrieves the variables sent to the system when asynchronous requests execute during the navigation through the app (JavaScript). We could not retrieve this information otherwise.

2. Python objects

In total the project counts about ten python class objectsnamong which we present here the most important ones.

The class object 'Extractor' of this file can accommodate a wide number of file formats. This includes native documents files as well as scanned document or images formats. The supported file formats are the following :

- document file formats: .txt, .pdf, .doc/.docx, .ppt/.pptx
- image/scanned document file format: scanned pdfs, .bmp, .dib, .jpeg, .jpg, .jpe, .jp2, .png, .pbm, .pgm, .ppm, .pxm, .pnm, .pfm, .sr, .ras, .exr, .hdr, .pi

The document extractor relies, amongst other things, on the following libraries: win32com, docx, pptx, pytesseract (OCR), pdfminer, pdf2image;

The class object 'TextCleaner' of this file uses the following NLP text cleaning methods: remove delimiter (e.g. '\n'), remove HTML tags, parse URL (keeping only the domain name such as, for example, ‘cosmetic’ for www.cosmetic.be), expand contraction, correct spelling (optional), remove accent, remove punctuation, remove numbers, lemmatize, remove stop words, remove the single character and remove extra white space.

The correct spelling processing is optional, a user may use it or not use it. For instance, during the navigation through the app interface, see the next section. We recommend using this option only when the text may be of low quality. By this, we mean social media contents with their 'jargonized' language and low-quality scans and/or images where the OCR might produce errors. For text known to be of good quality (e.g., scientific papers or abstracts), it is preferable not to use this option. We noticed that the spell correcting library sometimes replaces correct words with other similar words, e.g. 'physical' for 'topological.'

The 'remove stop word' function is based on a list of stop words constructed during the process. This list is partly based on the existing stop word list provided

by some NLP libraries (including sklearn, spacy, gensim, and nltk). This list is also based on the corpus itself, as it includes all the words that only occur in one document. The rationale for including 'rare' words is that they don't discriminate between classes and may be considered noise.

Besides cleaning the text, TextCleaner also produces the TF-IDF vectorizer needed for running the XGBoost model.

The two models we discussed previously are materialized by two other class objects that have quite the same structure :

1. Stratified train-validation split
2. Model training
3. Model evaluation on the validation set (30% of the total sample)

We needed to accommodate both multiclass and multilabel dataset types during the training. Therefore, we chose to operate a stratified train-validation split instead of the more conventional train-validation split.

When confronted with a multilabel dataset during the training, the model considers the number of original classes and all class combinations presented. Thus, the number of all potential class combinations is much larger than the original number of classes. Some class combinations might have very few documents or even no documents.

Figure 2 presents the value counts of each specific class combination (as multiclass one hot encoded) for a given dataset. As mentioned earlier, we can see that some class combinations do not appear in the count (e.g. [1 0 1 1 1 1]) as they have no document, and some others have very few documents (e.g., class combination [0 0 1 1 0 1] has only one document).

Figure 2. *Value counts of class combinations for a multilabel dataset*

```
Entrée [18]: label_counts

Out[18]: [0 1 0 0 0 0]    5120
         [1 0 0 0 0 0]    4910
         [0 0 1 0 0 0]    3610
         [1 0 0 1 0 0]    2285
         [0 0 0 1 0 0]    1636
         [0 0 1 1 0 0]     825
         [1 0 1 0 0 0]     682
         [0 0 0 0 1 0]     443
         [1 1 0 0 0 0]     437
         [0 1 1 0 0 0]     293
         [0 0 0 0 0 1]     209
         [1 0 1 1 0 0]     179
         [0 0 0 1 1 0]     105
         [0 1 0 1 0 0]      99
         [1 1 0 1 0 0]      36
         [1 0 0 0 1 0]      30
         [0 0 0 1 0 1]      24
         [1 1 1 0 0 0]      19
         [1 0 0 0 0 1]       9
         [0 1 1 1 0 0]       9
         [1 0 0 1 1 0]       5
         [0 0 0 0 1 1]       4
         [1 0 0 1 0 1]       2
         [0 0 1 1 0 1]       1
         Name: one_hot_labels, dtype: int64
```

In such a context, the risk with a conventional train-validation split is that some class combinations that the model has not previously seen during the training may be presented during its evaluation. With that respect, the stratified train-validation split ensures that the train-validation split proportion (in our case 70% vs. 30%) is somehow respected within each class combination and that no unseen class combination is presented during model evaluation.

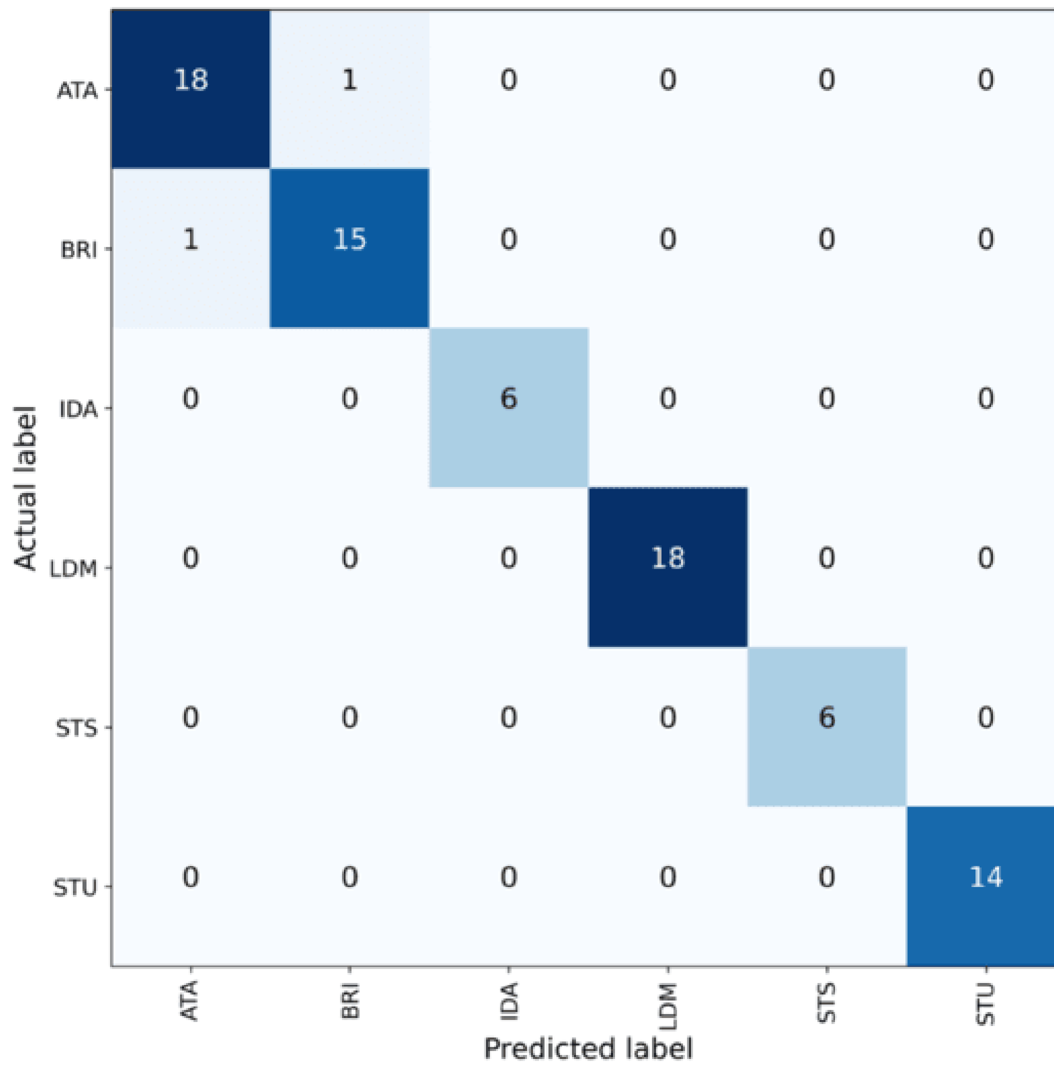
The class object 'Report' offers a detailed and in-depth analysis of the model both overall and at the class level, including, amongst other things, classification

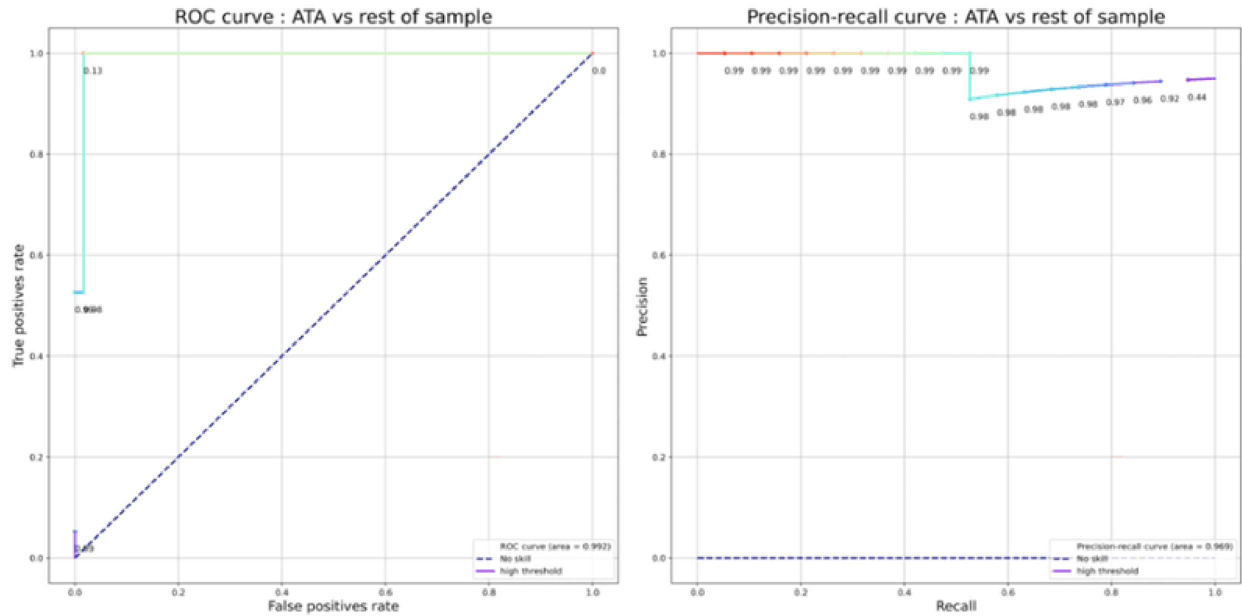
reports, confusion matrices, ROC, and Precision-recall curves. Figure 3 shows some examples of outputs from this report.

Figure 3. *Examples of outputs from the performance report*

Classification report

	precision	recall	f1-score	support
ATA	0.94	0.89	0.92	19.00
BRI	0.88	0.94	0.91	16.00
IDA	1.00	1.00	1.00	6.00
LDM	1.00	1.00	1.00	18.00
STS	1.00	1.00	1.00	6.00
STU	1.00	1.00	1.00	14.00
accuracy	0.96	0.96	0.96	**0.96**
macro avg	0.97	0.97	0.97	79.00
weighted avg	0.96	0.96	0.96	79.00


Confusion matrix
Actual label
ATA
BRI
IDA
LDM
STS
STU
18
1
0
0
0
0
1
15
0
0
0
0
0
0
6
0
0
0
0
0
0
18
0
0
0
0
0
0
6
0
0
0
0
0
0
14
ATA
BRI
IDA
LDM
STS
STU
Predicted label



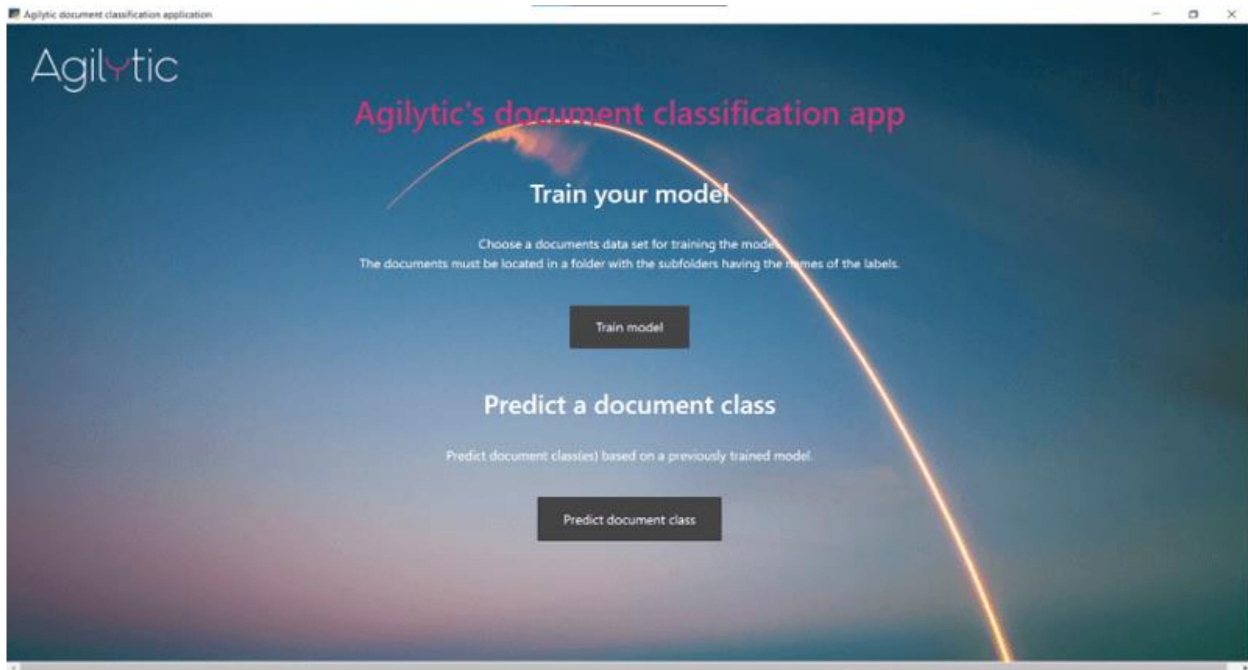
When the user is satisfied with a model's performance, the object 'Prediction' can predict an incoming unlabeled document class. Before running its code, this object calls the objects 'Extractor' and 'Cleaner' to preprocess the documents' text precisely as during the training.

Navigation through the app user interface for document classification

For displaying the HTML files conceived for the user interface, the current project relies on a python-specific graphical user interface (GUI) wrapper (the pywebview package) as it allowed some specific functionalities required for running the code and that would not be permitted with more commonly used web-based GUI (e.g., Google Chrome). During the process, the system needed to retrieve the full path of some files or directories, for example, treating the documents' dataset. While this is perfectly feasible with pywebview, Chrome would upload the file and retrieve the file name (without the path) in similar cases.

We designed the user interface to be self-explanatory and intuitive. Moreover, we tried to give the user control of each step of the process and offer good visibility on how things are running (intermediary reports provided after each step and before launching the next one). The intent here was to help the user to have a good insight into the data, the data processing, and the model quality. Figure 4 shows the home page of the app's user interface.

Figure 4. *Home page of the app's user interface*



The document prediction web app

As previously mentioned, the web app only covers the document prediction workflow (and not the model training workflow). In this sense, it uses the same python code as the whole project (but only the classes and functions used for prediction), but some changes must be made regarding the user interface and deployment.

When the user uploads a document, the prediction is automatically processed, and the user is thereafter redirected to the prediction report page, which includes the following output:

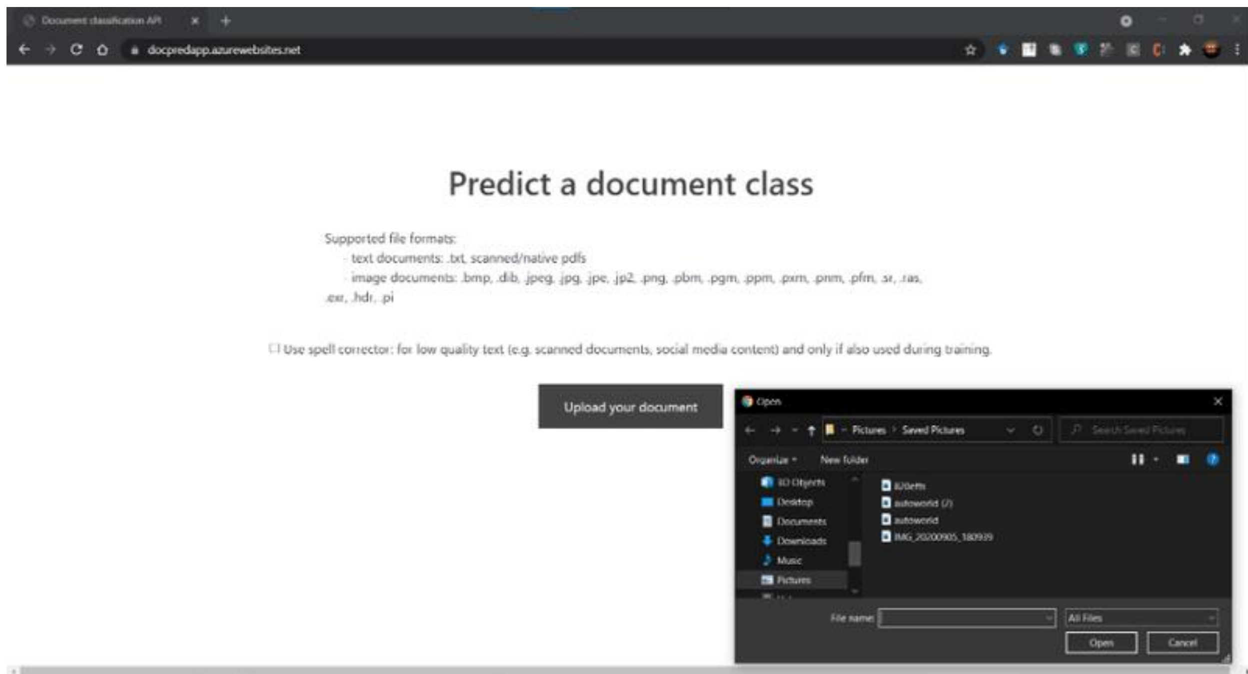
- The name of the document

- Status: 'ok' or 'ko' (meaning that something went wrong: the file format is not supported, or the document is opened in another system/software

- Predicted class

Another cge compared to the local app is that the HTML files had to be extracted from pywebview to render them in any web browser/GUI (see Figure 5).

Figure 5. *Home page of the web app*



The web app was deployed on Microsoft Azure for testing its proper functioning. We used Docker to create the development environment. For using Docker, two files were created:

- The 'Dockerfile' allows us to create a Docker image of the project with all the requirements for running it (including all python packages used by the system).

- The file 'docker-compose.yml' gives all the necessary instructions to the hosting environment for launching the app and making it running.

Travis CI is a hosted continuous integration service used to build and test software projects hosted on GitHub. As a complement to Docker, we also used Travis CI to make a direct connection pipeline between the local repository (cloned from the current repository) and the web app deployed on Azure. In the interest of this project, it is easier to update the system with newly trained models.

Conclusion and potential further developments for document classification

Overall, the current project has reached a point where it is well-performing and highly functional (should it be used by inexperienced users or by expert users). As long we present the data to the system correctly (the directory containing the dataset documents with subdirectories as label names), the app can

accommodate various file formats and dataset types (for binary, multiclass or multi-label document classification).

In this sense, this project offers highly versatile tools that we can adapt to various business cases. Finally, business applications of the type of the current project are also virtually infinite. To name just a few examples:

- Automating the company's documents dispatching (to the right person or the right department);
- Automating email dispatching and sorting
- Improving the performance of document/text content search engines
- Detecting spam or fraudulent/atypical documents
- Treating text from medical records to detect pathologies and comorbidities (multilabel-like situation). Although such a system should usefully integrate additional medical data from patients, i.a. medical imagery, test results.

About Agilytic

Since 2015, Agilytic helps innovative leaders solve their biggest challenges through the smarter use of data. With over 150 successful projects to date, we have perfected a pragmatic approach to putting data at the service of business goals, be they commercial, operational, financial, or human. Reach out today for a quick introduction, we'd love to hear from you.