Automatically inferring the document class used in a scientific article

Antoine Gauquier

Télécom Paris - École Normale Supérieure

Automatically inferring the document class used in a scientific article

Final report - IA 311

Antoine GAUQUIER - AI Option - Academic year 2022/2023

Supervisor at E.N.S.:
Pierre Senellart

Supervisor at Télécom Paris:
Antoine Amarilli

Contents

List of Figures

List of Tables

Introduction

The purpose of this report is to summarize what I have achieved in the framework of Télécom Paris' AI option research project.

My project, entitled "Automatically inferring the document class used in a scientific article", is carried out within the École Normale Supérieure, and more particularly within the Valda team at Inria Paris. It is co-supervised by Pierre Senellart, university professor at École Normale Supérieure and head of the Valda team, and Antoine Amarilli, associate professor at Télécom Paris.

There are several interests in automatically determining the class of scientific papers. What we call here "document class" or just "class" is the document class command used in LaTeX to structure the paper. It is usually associated with the journal or conference in which the paper is to be published, but it is not necessarily the case. First, achieving this identification task could make it easier to determine whether a paper is available in open access or not, since it is usually linked to the journal in which it is published. This is what the Dissemin project (`https://dissem.in/`) is about. Another interest lies in extracting information directly from the papers. Indeed, the TheoremKB project (`https://github.com/PierreSenellart/theoremkb`) aims to develop a knowledge base consisting of theorems, lemmas and definitions extracted from scientific articles. Thus, knowing the document class of an article informs us about the structuring of the papers (how theorems are structured for instance), and must therefore facilitate the extraction of information relating to its contents.

The field of information extraction from scholarship publications is an already well studied area [Lop09] [BGSF10] [KLN10] (other publications in this field can be found at `https://csxstatic.ist.psu.edu/downloads/software`). However, the subject of predicting document classes, especially with computer-vision based approaches, is something that, to our knowledge, has not yet been studied. The objective of this work is therefore to develop algorithms to identify this document class automatically. However, this task is more complex than it seems, and this is for two main reasons.

The first reason is that we must work on algorithms that are generalizable to (in theory) any document class, from any field (it could be mathematics, social sciences, IT, and so on). This means that we cannot focus on the content of the papers, but only on their structure and properties. The second difficulty lies in the fact that there are hundreds of document classes, some of them containing really heterogeneous documents, and that it is impossible to train an algorithm that observes them all. It is therefore necessary to think about solutions to find a balance between completeness and realism.

My work focused on these two issues. In order to propose solutions, I considered two different approaches. The first approach is to generate, from the PDF documents of the papers, relevant characteristics (or features) allowing to discriminate the different classes of documents. These characteristics are therefore selected manually according to what is known about the document classes, and learning algorithms are then used to make automatic inference. This approach will be presented in the first chapter of this report. The second approach is to rely on computer vision methods. Indeed, we make the hypothesis that by providing the computer with images of the papers, it will succeed in extracting relevant characteristics, potentially different from those that we considered relevant as a human in the first approach, but that are geometric. This will be presented in the second chapter of this report.

Chapter 1

Presentation of the data

In this chapter, we present the data we will be using to train the different models.

First, to be able to learn the document classes from scientific articles, we need a large corpus of scientific articles. We decided to make an extraction of such a corpus from arXiv (`https://arxiv.org/`), which is a free and open-access archive for scientific papers. For each paper that we extracted, we have access to the PDF of the document, as well as the LaTeX source code written in order to generate it.

The processing and extraction of features from the PDF documents will be presented in the next two sections, since each approach exploits the document differently. But the extraction of the document classes (also later called ground-truths values, or simply ground truth), is common for both approaches. This extraction of thanks to the LaTeX archive only can be hard in certain cases. In fact, since it is the `\documentclass` command that defines the document class, we sometimes find comments where this command appears or multiple definitions of different document classes. However, this only represents a really small amount of cases. Thus, by adding some filters in order to handle most of the cases, we will consider that extracting the `\documentclass` command is sufficient to know the document class.

Also, in order to ensure a diversity of papers and fields, we took a set of papers from the year 2018. This set is mostly about papers in sciences (mathematics, physics, computer science and so on), since arXiv mostly focuses on archiving scientific articles in these fields. However, the methods that will be presented in the next two chapters can be adapted to any fields where papers are published, as long as they are generated through the use of LaTeX with a defined `\documentclass`.

We end up having around 100 000 documents for more than 1 200 different document classes. But most documents are written from only a few classes of documents. Figure 1.1 shows the percentage of papers covered by the main document classes in increasing order. By only keeping the 20 main classes, we already cover 90% of the papers. More than half of the different document classes only have one sample, which makes them unusable for a statistical learning model. We must therefore limit ourselves to a sufficiently significant number of classes. We decided to keep only classes that represent at least one thousandth of the total data, which correspond to a little more than 100 distinct samples per class. By considering this criterion, we only keep 43 classes, covering 95% of the data. We also add a 44-th document class "other", in which we will put all the remaining examples, plus the document class "article", which is the most frequent class among our set of papers, and which is the default document class that is too heterogeneous to be considered as one possible document class.

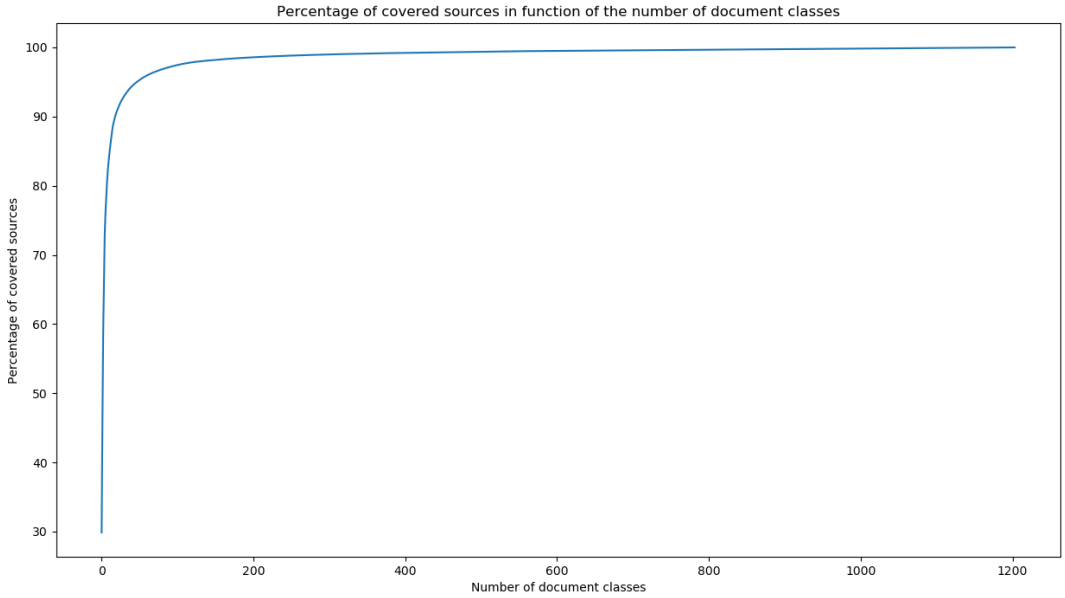


Figure 1.1: Cumulative coverage of papers by the main document classes

By looking at the names of the extracted document classes, we quickly observe similarities between class names. Examples include `aastex6`, `aastex61`, and `aastex62`, `ieeeconf` and `ieeetran`, as well as `amsart` and `amsproc`. It is therefore likely that some classes are in fact very similar, but the only similarity in name is not enough to justify a merge of these classes. We could also miss similar classes whose names are too different to predict that they are similar. That is why we decided to detect classes to be merged by using an editing distance directly on the source codes of the classes of documents. These source codes can be found online, or in our case, directly in the source folders extracted from arXiv (where we also find the `.tex` files). In general, these sources are available in `.cls` format and have the name of the associated document class (for `aastex61` for example, the source file is `aastex61.cls`).

We made the choice to use the term frequency-inverse document frequency (TF-IDF), which evaluates, for a given document, the importance of each term in this document. It therefore computes a vector of size number of words in a document and associates for each word an importance score. Once those scores are computed for all documents (we have previously made sure to align all documents on a same vector of possible words), we can compute a pairwise document-similarity, by computing, for all pairs of documents (i, j) with $i \neq j$:

$$s_{(i,j)} = \langle \text{TF-IDF}_i, \text{TF-IDF}_j \rangle$$

This gives a score between 0 and 1, representing how similar the pair of documents is. We decided to set a threshold at 75% to obtain a limited number of cases to study. The obtained results are presented in Table 1.1 (since this similarity score is symmetric, we removed similarity between A and B if similarity of B and A is already presented).

Given these results, we decided to merge some of the classes. First of all, we regrouped `aastex6`, `aastex61`, and `aastex62`, since they almost are perfectly equal (we only lack less than 1% similarity). We reach the same observation for classes `report` and `wlscirep` (100%), for `llncs` and `svproc` (99%), and for `amsart` and `amsproc` (99 %). `svjour` and `svjour3` also have a high similarity score (95%), as well as `mn2e` and `mnras` (93%), and as `ieeeconf` and `ieeetran` (92%).

Table 1.1: Pairwise similarity between document classes sources (obtained with TF-IDF)

Document class 1	Document class 2	Percentage of similarity
aa.cls	llncs.cls	78%
aa.cls	svproc.cls	78%
aastex6.cls	aastex61.cls	99%
aastex6.cls	aastex62.cls	99%
aastex61.cls	aastex62.cls	100%
amsart.cls	amsproc.cls	99%
iau.cls	jfm.cls	89%
ieeeconf.cls	ieeetran.cls	92%
iopart.cls	jpconf.cls	85%
llncs.cls	svjour.cls	78%
llncs.cls	svjour3.cls	79%
llncs.cls	svmult.cls	87%
llncs.cls	svproc.cls	99%
mn2e.cls	mnras.cls	93%
report.cls	wlscirep.cls	100%
svjour.cls	svjour3.cls	95%
svjour.cls	svmult.cls	79%
svjour.cls	svproc.cls	78%
svjour3.cls	svmult.cls	80%
svjour3.cls	svproc.cls	79%
svmult.cls	svproc.cls	87%

We can also consider merges for classes that share a similarity between 85% and 90%. This is why we decided to merge classes `iau` and `jfm` (89%), as well as classes `iopart` and `jpconf` (85%). We also have `llncs` and `svmult` that share 87% of similarity, which can also be merged. Since `llncs` was already merged with `svproc`, now they form a group of three classes. The relevance of this merge is also confirmed by the fact that `svmult` and `svproc` share 87% of similarity as well.

We still have remaining similarities in the table that do not appear in the merges mentioned above. The remaining ones concern the two (newly) groups of merged classes `svjour/svjour3` and `llncs/svmult/svproc` They all share (i.e., for each possible pairs between the two groups), a pairwise similarity of around 80%. We consider that even if this score is quite important, the merge of these two set of document classes would not be coherent. In fact, the first group mentioned above correspond to Springer document classes for journals, whereas the second group correspond to Springer document classes for proceedings. To confirm this choice, we can observe that classes `aa` shares a similarity of 78% with classes `llncs` and `svproc` (so, from the group `llncs/svmult/svproc`), but only a similarity of 61% with the classes `svjour/svjour3` (not in the table due to the threshold).

With all these merges, we end up having a total of 33 new classes, composed of both classes and groups of merged classes. We therefore reduced the number of classes by 25%. The number of papers associated with each of these classes is presented in Appendix A.

Now that we have defined the set of target classes, as well as the corpus of documents we will be using, we can present the two approaches we used to achieve the classification task. These ones are presented in the two following chapters.

Chapter 2

Inferring document-class with hand-designed features

In this chapter, we present the approach based on a hand-designed set of features to predict the document class of scientific articles.

In the first section, we present PDFAlto, the tool we used to exploit the PDF documents. We then present in the second section the features in question and how they are constructed. Then, in the third section, we do a statistical analysis of the distribution of data through the prism of established features to justify the use of classification methods. Finally, in the fourth and last section, we present a multi-class classification model trained with these features, and the results we get.

2.1 From PDF files to XML representations

We now have at our disposal the document classes associated with the papers that compose our dataset. We must now exploit these papers to extract features from them. Extracting such features directly from documents in PDF formats is complex and difficult for a human.

This is why we use a tool called PDFAlto (`https://github.com/kermitt2/pdfalto`), which automatically extracts information from PDF documents. More precisely, PDFAlto is a tool for parsing PDF files and producing structured XML representations of the PDF content in ALTO format.

We can thus look for specific information about the PDF files by using the tree-structure of the XML format. This is something that can be easily done thanks to the `ElementTree` object from the Python library `xml.etree`.

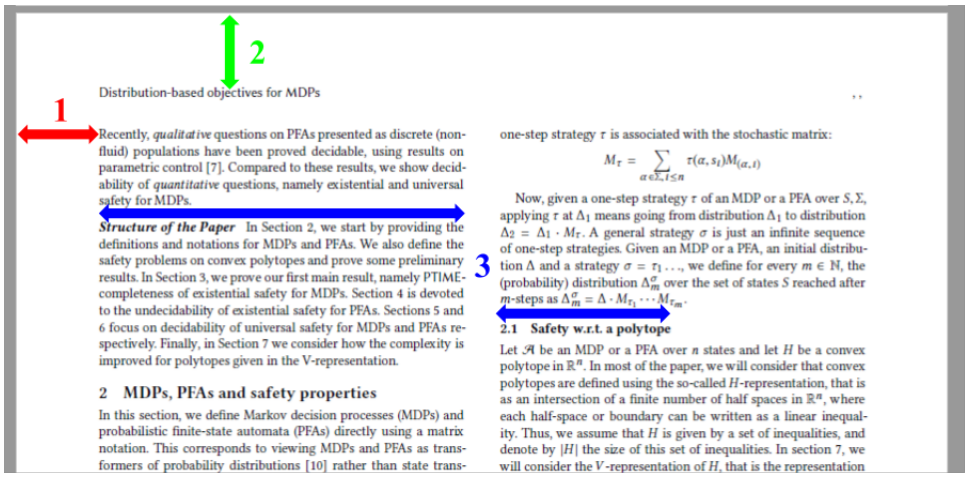
We will now see in the next section what kind of information we want and can use to extract features of interest for our classification problem.

2.2 Extracting hand-designed features from the papers

Before going into the technical details of the extraction of variables, it is interesting to describe the choices of variables that we have made. We follow two main goals through the construction of this set of features. We want to show that a classification approach is relevant, i.e., that our features, when taken per class, are distributed differently and thus make the data separable, but also that the intuition we have that computer vision will be efficient is relevant. This is why we chose to construct a set of five simple, human-understandable, geometrical features. The idea of each feature took birth through the observation of scientific papers, and through the prior knowledge that we have about how scientific papers are usually

organized.

This is why we wanted to use : the left-horizontal margins (1), the top-vertical margins (2), the text-columns width (3), the font-families and the font-sizes. Since the three features can be better understood visually, an illustration on a paper taken randomly is shown in Figure 2.1.



Distribution-based objectives for MDPs

Recently, *qualitative* questions on PFAs presented as discrete (non-fluid) populations have been proved decidable, using results on parametric control [7]. Compared to these results, we show decidability of *quantitative* questions, namely existential and universal safety for MDPs.

Structure of the Paper In Section 2, we start by providing the definitions and notations for MDPs and PFAs. We also define the safety problems on convex polytopes and prove some preliminary results. In Section 3, we prove our first main result, namely PTIME-completeness of existential safety for MDPs. Section 4 is devoted to the undecidability of existential safety for PFAs. Sections 5 and 6 focus on decidability of universal safety for MDPs and PFAs respectively. Finally, in Section 7 we consider how the complexity is improved for polytopes given in the V-representation.

2 MDPs, PFAs and safety properties

In this section, we define Markov decision processes (MDPs) and probabilistic finite-state automata (PFAs) directly using a matrix notation. This corresponds to viewing MDPs and PFAs as transformers of probability distributions [10] rather than state trans-

one-step strategy τ is associated with the stochastic matrix:

$$M_\tau = \sum_{a \in \Sigma, 1 \leq s} \tau(a, s_l) M_{(a,l)}$$

Now, given a one-step strategy τ of an MDP or a PFA over S, Σ, applying τ at Δ_1 means going from distribution Δ_1 to distribution $\Delta_2 = \Delta_1 \cdot M_\tau$. A general strategy σ is just an infinite sequence of one-step strategies. Given an MDP or a PFA, an initial distribution Δ and a strategy $\sigma = \tau_1 \ldots$, we define for every $m \in \mathbb{N}$, the (probability) distribution Δ_m^σ over the set of states S reached after m-steps as $\Delta_m^\sigma = \Delta \cdot M_{\tau_1} \cdots M_{\tau_m}$.

2.1 Safety w.r.t. a polytope

Let $\mathcal{A}$ be an MDP or a PFA over n states and let H be a convex polytope in $\mathbb{R}^n$. In most of the paper, we will consider that convex polytopes are defined using the so-called H-representation, that is as an intersection of a finite number of half spaces in $\mathbb{R}^n$, where each half-space or boundary can be written as a linear inequality. Thus, we assume that H is given by a set of inequalities, and denote by $|H|$ the size of this set of inequalities. In section 7, we will consider the V-representation of H, that is the representation

Figure 2.1: Illustration of the first three features.

However, each of these elements is presented in multiple copies in the document. Indeed, a scientific article often has several blocks of text, several pages and even several fonts. This is why it was necessary to build features composed from the different instances of the elements presented above. Here they are:

- **The weighted average of the left-horizontal margin**. The average is made in terms of text-block length. Let us denote m_i^{h} the left-horizontal margin for i-th text-block of the document, and l_i its vertical length. Then, the weighted average left-horizontal margin $\overline{m^{\mathrm{h}}}$ is defined as follows:

$$\overline{m^{\mathrm{h}}} = \frac{\sum_{i=1}^{N_b} m_i^{\mathrm{h}} \times l_i}{\sum_{i=1}^{N_b} l_i}$$

  where N_b denotes the number of text-blocks in the document. This is a numerical feature (float).

- **The average of first top-vertical margin**. We are here interested in the gap between the top-border of the paper and the first block of content (but not the margin between the top-border and all blocks of content). This is why we will simply average this first top-vertical margin with the number of pages of the document. By denoting m_i^{v} the first top-vertical margin of i-th page of the document, the average of first top-vertical margin $\overline{m^{\mathrm{v}}}$ is defined as follows:

$$\overline{m^{\mathrm{v}}} = \frac{\sum_{i=1}^{N_p} m_i^{\mathrm{v}}}{N_p}$$

  where N_p denotes the number of pages in the document. Again here, this feature is numerical (as a float).

- **The weighted average of the column width**. Technically, it corresponds to the weighted average of a text-block in a document. The weights are computed exactly as they were for the first feature,

i.e., with the length of each text-block. We will here denote w_i the width of i-th text-block in the document, and l_i its vertical length. The weighted average column-width $\overline{w}$ is defined as follows:

$$\overline{w} = \frac{\sum_{i=1}^{N_b} w_i \times l_i}{\sum_{i=1}^{N_b} l_i}$$

where N_b is still the number of text-blocks in the document. It is also a numerical feature (float).

- **The most common font-family in the document**. Choosing the most common font-family requires defining a criteria. We chose to quantify the importance of a font-family by the space it occupies in the document. Let us denote S, the set of all strings in the document, S_i the set of strings associated with the i-th font-family of the document, and N_f the number of different font-families in the document. We define the font-family importance of i-th font-family of the document f_i as follows:

$$f_i = 100 \times \frac{\sum_{s \in S_i} l_s \times w_s}{\sum_{j=1}^{N_f} \sum_{s \in S_j} l_s \times w_s}$$

where l_s is the length of string s, and w_s the width of string s. The product $l_s \times w_s$ thus gives an area, and we compute the portion of area that each font family occupies. To finally get the most common font-family f^{family}, we take the one associated with biggest font-family importance:

$$f^{\text{family}} = \arg \max_{i \in \{1,\dots,N_f\}} f_i$$

This feature is a categorical one, since it is a string of the name of the feature.

- **The most common font-size in the document**. Here we are interested in the font size associated with the most common font-family used in the document. Therefore, we need to compute the fourth feature before this one, and then have a look at the `STYLEREFS` associated with the concerned font-family in the XML document to find this font-size. It is a numerical value (an integer most of the time, but it could be a float).

In the next section, we are going to make a statistical analysis of the dataset we obtain by computing these features for our set of documents.

2.3 Statistical analysis

As we did when presenting the features, we are going to proceed feature by feature to present their distributions. We are first going to discuss their general distribution, i.e., without being class specific. Without getting into each class, it is important to note that PDF Alto sometimes encounters difficulties in extracting information. We can therefore from time to time observe some surrealistic values (as negative widths, or even "NaN" values). The results that we present here are given on cleaned data, where we already removed such irrelevant values. We do not consider this observation as a problem, since it is only observed in a small number of cases (for instance, concerning the first feature, we only have four negative widths and two "NaN" values, over around 100 000 samples).

- **The weighted average of the left-horizontal margin**. The value of this feature is expressed in points in the image. In the PDF format (as well as in other formats), a point is defined as $\frac{1}{72}$-th of an inch. The average value of this feature is equal to 159.99 points, its median value is 166.42 points and its standard-deviation is about 38.20. Figure 2.2 shows the plot of the histogram for this feature. The x-axis represents the values taken by the feature (i.e., the number of points, as defined above),

and the y-axis the corresponding frequency of these values (or in fact, these sets of values since it is a continuous feature).

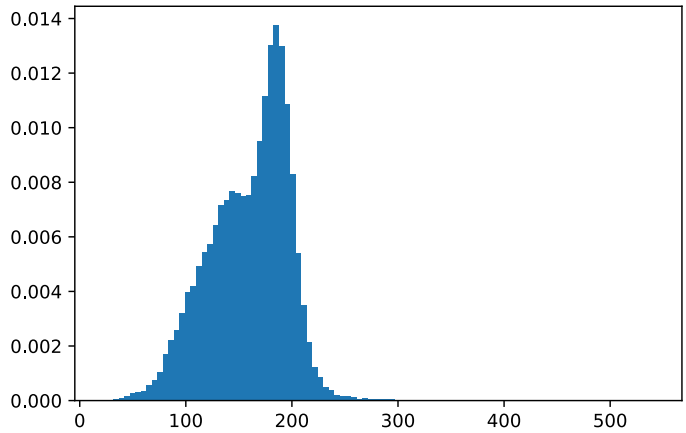


Figure 2.2: Histogram of the values taken by the weighted average of the left-horizontal margin feature

We observe what seems to be the superposition of two Gaussian-like distributions: the first with a mean close to 140 and the other one with a mean close to 190.

- **The average of first top-vertical margin**. Here again, the value of this feature is expressed in points in the image. Its average value is equal to 89.58 points, its median value is 187.39 points and its standard-deviation is about 44.33. Figure 2.3 shows the plot of the histogram for this feature. The x-axis represents the values taken by the feature (i.e., the number of points), and the y-axis the corresponding frequencies.

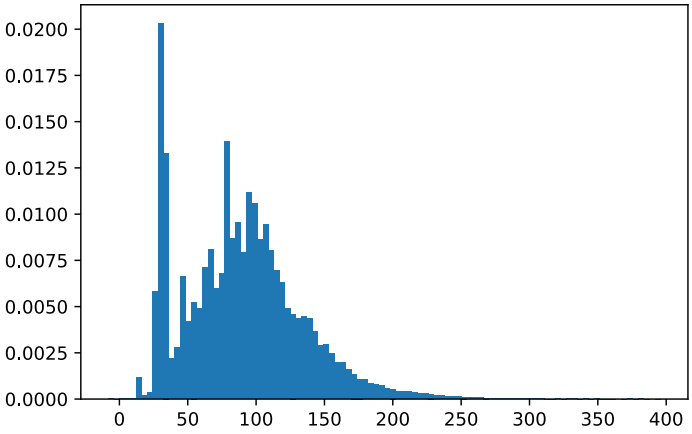


Figure 2.3: Histogram of the values taken by the weighted average of the first top-vertical margin feature

We can also observe a Gaussian-like distribution of the data, around the value 90, with an important peak at approximately 40 pixels.

- **The weighted average of the column width**. Once again, the value of this feature is expressed in points in the image. Its average value is equal to 242.25 points, its median value is 228.51 points and its standard-deviation is about 65.93. Figure 2.4 shows the plot of the histogram for this feature. The x-axis represents the values taken by the feature (i.e., the number of points), and the y-axis the corresponding frequencies.

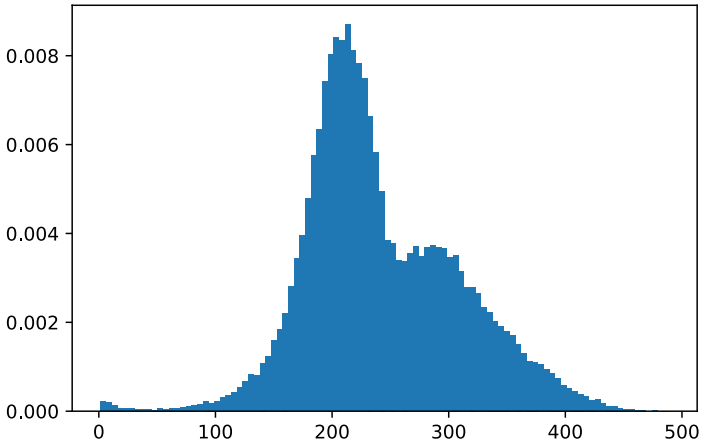


Figure 2.4: Histogram of the values taken by the weighted average of the column width feature

By observing the shape of the histogram, we identify what seems to be the superposition of two Gaussian-like distributions : one which mean is around 200 points, and one which mean is around 300 points. Given the fact that the width of a page is usually around 600 points, and that some space is kept for the margins, these two distributions most likely respectively correspond to papers with two columns (smaller column-widths) and papers with one column (greater column-widths). Moreover, we also observe that this histogram more or less looks like the histogram of the left-horizontal margins with a vertical symmetry. This tends to show that the column-width represents papers with two or one columns : the papers with two columns are more likely to have smaller horizontal margins than papers with a single column.

- **The most common font-family in the document**. This feature is different from the other ones, since it is categorical. We cannot describe it with continuous metrics, but the histogram presented in Figure 2.5 is sufficient to correctly describe its distribution.

It is important to note that we have not displayed all possible font-families, since there are 109 of them, and that would have become impossible to read. Instead, we only kept the font-families that have at least 100 representatives (samples that take this value). The rest is put together into `all_other`. We see that we only have a few significant font-families in terms of frequency (the three most frequent are `cmr`, `nimbusromnol` and `sfrm`). This does not mean that other font families are not important, since we have some big unbalance between document classes.

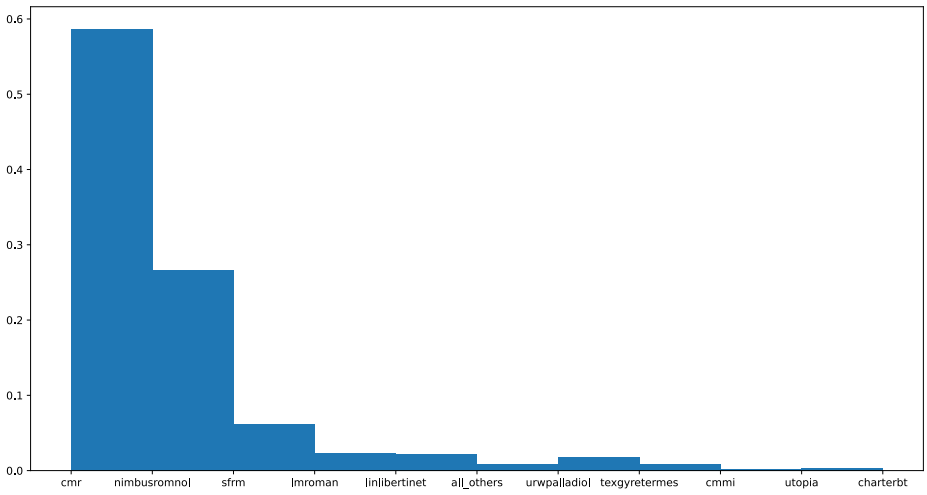


Figure 2.5: Histogram of the values taken by the most common font-family feature

- **The most common font-size in the document**. Finally, the value of this feature represents size. Its average value is equal to 10.44, its median value is 10 and its standard-deviation is about 0.89. Figure 2.6 shows the plot of the histogram for this feature. The x-axis represents the values taken by the feature, and the y-axis the corresponding frequencies.

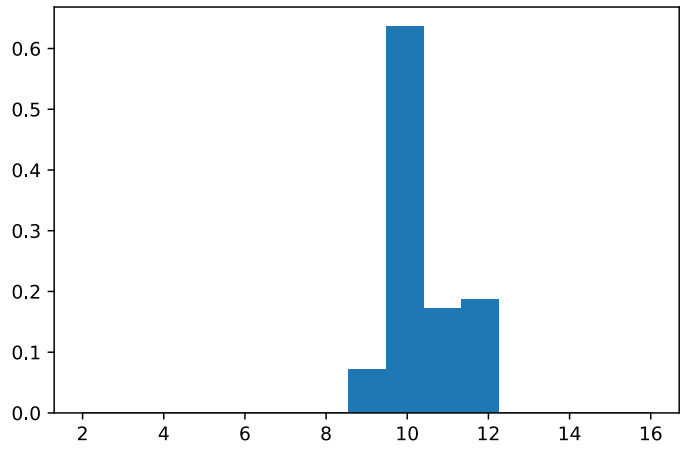


Figure 2.6: Histogram of the values taken by the most common font-size feature

We observe here that, despite the fact that it is a numerical feature, we could consider it as a categorical one. This is due to the fact that the font-sizes associated with the most common font-families follow some standards. The standard font-size value in LaTeX most common font-families is 10 (we see that it corresponds to more than 60% of all font-sizes), but font-sizes between 9 and 12 are also common, as we can see in the figure.

What is also important to validate that our features makes our data separable and thus justify the use

of classification techniques is the fact that the distribution of the features observed per-class differs from one to another (and that they are not all equally distributed). It would not be reasonable to make an exhaustive comparison of each sub-distribution for each class, or even to describe each of them (we would end up displaying and commenting $33 \times 5 = 165$ distributions). Instead, I will briefly comment on the observed trend (if there is so) for each feature. Appendix B presents five images, each image containing the 33 distributions for a given feature. The images are displayed in vector graphics format (as well as the histograms above), such that the reader can zoom on the images and make his or her own observations.

By having a look at the global trend, we clearly see that we have different distributions across the classes. Regarding the features that are numerical, i.e., the average left-horizontal margin, the first top-vertical margin or the average column width, some of the document classes have values that are distributed in a Gaussian-like manner, with mean values that significantly differs from one to another. Other document classes do not exhibit a clear Gaussian-like distribution, and might not be useful for a classification task, the concerned feature being taken alone. In fact, in a general manner, we can compute the correlation of the features between each other, but also their correlation with the associated document classes. The results are presented in the Table 2.1. The features are ordered in the same order as they were presented in the list above.

Table 2.1: Correlation matrix, involving the five hand-designed features (in their order of presentation), and the ground-truths values

	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Ground-truth
Feature 1	-	-20.46%	-65.28%	8.24%	-1.86%	2.44%
Feature 2		-	9.62%	-3.74%	1.17%	-4.51%
Feature 3			-	-4.27%	3.22%	-1.48%
Feature 4				-	-0.59%	-3.65%
Feature 5					-	-0.68%
Ground-truth						-

We can see that none of the features is significantly correlated with the output. But this is not important, since what will really matter is the interaction between several features. In fact, it seems even impossible to predict the document class of a paper with a single feature as an average value of margins or column-width (actually, it could work with some categorical feature such as font-family if the document class is the only one to use the given font-family among all other document classes). But considering at the same time all the geometrical features it is easy (at least, for a human being) to determine the document class, considering he observed some representatives of each class.

Finally, what we can also note, is that our observations about the link between the column-width and the left-horizontal margins is statistically verified, since the correlation coefficient between these features is high and negative (-65.28%).

2.4 Model training and analysis of results

Before presenting the chosen modeling process and its results, we must prepare the dataset for the training and testing. This is why we first make a split over our initial dataset, randomly keeping 80% of it for the training part, and 20% of it for the test part.

If we directly train a model in these conditions, we will end-up having a weak model, because of the existing imbalance between our classes. In fact, the model focuses on maximizing the global accuracy (i.e., the number of correctly classified samples divided by the total number of samples, without considering the document class). As a consequence, with 4 classes that concentrate 75% of the data, the model will focus on these ones and put the other ones away, especially the ones with lowest weight. Since we want our model to be good for all classes, we need to force it to put a more balanced attention on each class.

We must therefore artificially balance the weight of each class in order to overcome this problem: we apply oversampling. Concretely, this consists in having for each of the classes, as many samples as the class of greater weight. Since we have a limited dataset, we will simply randomly duplicate samples in low-weight classes, until they reach a number of samples equal to the class of greatest weight. This oversampling will also have the advantage of strengthening the learning by showing the model many samples. An illustration of oversampling is shown in Figure 2.7.

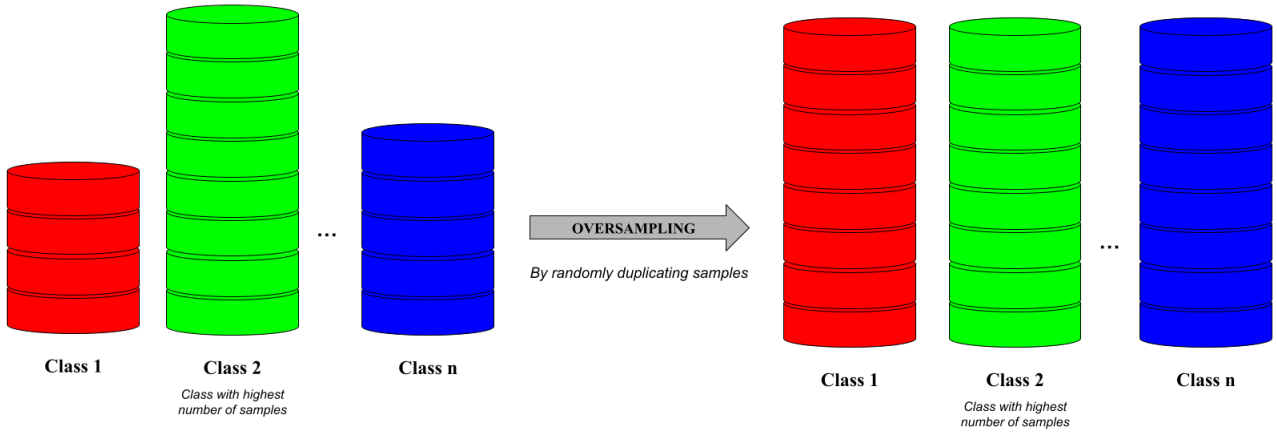


Figure 2.7: Illustration of oversampling technique.

With this oversampling method, we end up having 836.944 samples for the training set, and 210.766 for the test set. Since we have both categorical and numerical features, and since we have a lot of samples and a quite important number of classes, random forests [Bre01] seem to be a good tool for our problem. We tried other classification methods, some less advanced (logistic regression), or more advanced (Support Vector Machines), but it turned out that random forests are giving the best results.

Random forests is a type of model belonging to the ensemble methods. The principle of ensemble methods is to combine multiple weak learners (i.e., models whose performance is not considered as being good) in order to have a global model with good performance. The weak learners of random forests are decision trees, which are trees that try to classify data. On each node of the tree, a condition (usually, an inequality) over one feature of the data is evaluated. Depending on the answer, a branch is followed, and either leads to another node (and thus, another evaluation), or a leaf, corresponding to one of the possible classes. The random forest is thus composed of a set of decision trees (also called estimators), each of them being trained with a different subset of the data. The random forest provides the data to be classified to all the estimators, and finally makes a majority vote to decide which class is to be selected.

We used `RandomForestClassifier` from the Python package `sklearn.ensemble`. We needed to set some parameters to make the training work. First of all, we needed to set a minimum number of samples par leaf in the decision trees of the random forest. In fact, since we have important heterogeneity in our data, and given the quite high number of classes, there is a high risk of overfitting if we let this parameter free (i.e., no minimum number of samples per leaf: we could end up only having one sample per leaf). In fact, without setting this one, we ended up having 100% of accuracy on the training set (classes being now balanced), but a "bad" accuracy on the test set (around 40%). So, we decided to set this parameter to be 0.01% of total observed samples (thus corresponding to around 84 samples, which corresponds to a little less than the size of smallest classes before oversampling). We also decided to set the number of estimators (i.e., the number of decision trees involved in the forest), to be equal to 1000. Usually, the default value is 100, but since we have 33 classes, we wanted the most common decision to be stable enough by letting the algorithm have more estimators.

With these parameters, we succeeded in training the classifier. We are now presenting the results we obtained. Before being more class specific, we can describe the obtained accuracies. By accuracy we denote here, and basically for the rest of this report, the number of correctly predicted samples divided by the number of total samples. As we already mentioned it above, this metric does not absolutely reflect the performance of a model, especially if there is an imbalance among the classes. But since we made some oversampling, this accuracy can also be interpreted as the mean of all observed accuracies in the different classes. These metrics become relevant in the context of our problem, since it reflects how well the model classifies our data, considering that every class is as important as the other ones.

We obtained, on the training set, an accuracy of 87.55% , and on the test set, an accuracy of 65.95%. Before analysing these results, we are going to have a look at the obtained results for each class. Also, in order to have a finer analysis of the model performance, we will use three additional metrics, which are precision, recall and F1-score. Let us denote TP_i the number of "true positives" for class i, i.e., the number of samples that the model classifies in class i and that actually belong to this class. TN_i will then be the number of "true negatives" for class i, i.e., the samples that were rejected by the model (for a multi-class problem, it corresponds to the situation where a model classified the sample **not** to belong to class i, so basically to any other class), while actually not belonging to this class i. We also define FP_i as the number of "false positives" for class i, which is the number of samples that the model classifies in class i while belonging to another class. Finally, we define FN_i the number of "false negatives" for class i, which correspond to the samples that are rejected by the model for class i but that do belong to this class i.

Now that we have defined those values, the precision score of a given class i is defined as follows:

$$\mathrm{precision}_i = \frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FP}_i}$$

The recall score of a given class i is defined as follows:

$$\mathrm{recall}_i = \frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FN}_i}$$

And finally the F1-score, which is a combination of both precision and recall, is defined as follows:

$$\text{F1-score}_i = 2\frac{\mathrm{precision}_i \times \mathrm{recall}_i}{\mathrm{precision}_i + \mathrm{recall}_i}$$

We can now present the obtained results for each class, on training set and test set. They are shown in Table 2.2.

We will not exhaustively present the results of each document class, but we can exhibit some interesting results. We first see that all classes have better performance in train than in test. This is something completely normal and usual in machine learning, but it shows that we don't have a problem in our data split or in the training pipeline.

Then, we observe that some classes are really well classified, on which the model generalizes well. We can cite the document class `bmvc2k` which has perfect scores in training, and around 97%/98% in test. We identify other classes well classified by the model, such as `cms`, `eptcs`, `iau/jfm`, `lipics`, `mdpi`, `sigma` or `wbofc`. It proves that the classification approach is relevant for the data we are interested in, since the model succeeds in classifying an important number of classes very well (i.e., the features are such that the data belonging to those classes are easily separable).

But we also identify some classes that are very poorly classified. In particular, two classes have performance in test (for at least one metric), which is worse than a random classifier. Those classes are `other` and `book`. In fact, we could have anticipated those bad results: these two classes are generic document classes that can be personalized and parameterized as much as the user wishes. This differs from the document classes associated with specific journals or conferences, where there are some standards that cannot be changed. This is a really important result, since those heterogeneous classes will be particularly hard to

classify. This will be further studied in the second chapter of this report. Finally, the remaining classes have performance that vary from around 25% up to 85% for the different metrics.

To sum up, those results are particularly encouraging. But we must keep in mind that we are doing a hard classification task, on 33 classes. Thus, even classes with performance metrics around 25% are way above the results of a random classifier, whose performance would be about $100/33 \approx 3\%$. Moreover, we only use five hand-designed features, chosen from the prior knowledge we have about the domain. The modeling is therefore quite simple, since it is made of both simple features and simple learning architecture.

This proves that not only a classification approach is justified and relevant for this task, but also that using geometrical features is also relevant. We can have a look at the feature importance computed by the random forest while training to verify this. The feature that has the most important weight is the first top-vertical margin, with 32.7% of importance. Then, the four other features have a quite similar importance: the average left-horizontal margin has 19.3% of importance, the most common font-family has 18.4%, the average column width has 16.5%, and the font-size has 13.1%.

These five features are geometrical and all have their importance in the classification (we do not end up having some irrelevant features that have really poor importance on the classification). For all those reasons, it seems relevant to consider an approach that both complexifies the modeling (to further improve the classification of classes whose performance is good but improvable), and keeps this geometrical analysis of the papers. The natural modeling and architecture that corresponds to those criteria are Convolutional Neural Networks (also denoted CNNs), that we will present in the second chapter of this report.

Table 2.2: By class performance metrics for the random forest classifier model

	Train set			Test set		
Document class	Precision	Recall	F1-Score	Precision	Recall	F1-Score
aa	86%	90%	88%	77%	83%	80%
aastex	84%	95%	89%	51%	52%	51%
aastex6/aastex61/aastex62	73%	67%	70%	57%	58%	57%
achemso	79%	100%	88%	58%	65%	61%
acmart	98%	95%	97%	90%	95%	93%
amsart/amsproc	57%	32%	41%	19%	28%	23%
bmvc2k	100%	100%	100%	98%	97%	97%
book	96%	100%	98%	0%	0%	0%
cms	98%	100%	99%	94%	100%	97%
elsarticle	66%	58%	62%	40%	52%	45%
emulateapj	78%	84%	81%	55%	63%	59%
eptcs	97%	100%	98%	89%	88%	88%
iau/jfm	96%	100%	98%	96%	86%	90%
ieeeconf/ieeetran	67%	85%	75%	53%	84%	65%
imsart	83%	98%	90%	43%	31%	36%
iopart/jpconf	80%	79%	80%	59%	66%	62%
lipics	97%	100%	98%	86%	86%	86%
llncs/svmult/svproc	80%	77%	79%	63%	73%	68%
mdpi	96%	100%	98%	87%	100%	93%
mn2e/mnras	92%	96%	94%	62%	95%	65%
other	63%	3%	7%	24%	3%	6%
pos	95%	99%	97%	84%	87%	85%
report/wlscirep	90%	99%	95%	58%	47%	52%
revtex4	83%	61%	71%	68%	60%	64%
scrartcl	79%	92%	85%	40%	44%	42%
siamart171218	94%	100%	97%	58%	63%	60%
siamltex	93%	100%	96%	56%	43%	49%
sig	95%	100%	97%	81%	26%	39%
sigma	99%	100%	100%	94%	100%	97%
spie	89%	100%	94%	84%	80%	82%
svjour/svjour3	90%	79%	84%	62%	64%	63%
webofc	96%	100%	98%	89%	93%	91%
ws	85%	100%	92%	54%	65%	59%
Mean	**87%**	**88%**	**86%**	**64%**	**66%**	**64%**

Chapter 3

Inferring document-class with computer vision methods

In the previous chapter, we saw that a computer-based approach was relevant given the classification task we are trying to achieve. In particular, Convolutional Neural Networks are widely used in the domain of image classification. They consist of the successive application of filters to the original image and a final flattening step to get numerical labels. Their popularity is mainly due to the fact that, compared to fully-connected deep learning methods, such as Multi-Layered Perceptron for instance, the use of filters as parameters drastically reduces the number of parameters, and makes the training and evaluation steps way faster.

In the first section, we are going to present the data that will be used to train and test our CNN model, as well as the pre-processing steps required to generate them. Then, in a second section, we will choose and define the architecture that the computer vision models will have, and definitely confirm the relevance of the use of such an approach to our classification task, by solving a 2-class classification problem. Then, in the third section, we will train and evaluate a multi-class computer vision model on the same set of document classes we used for the random forest model. Finally, in the fourth and last section, we will try to use the observation about heterogeneity we made in the first chapter to define a new model and improve our classification.

3.1 Data generation and image pre-processing

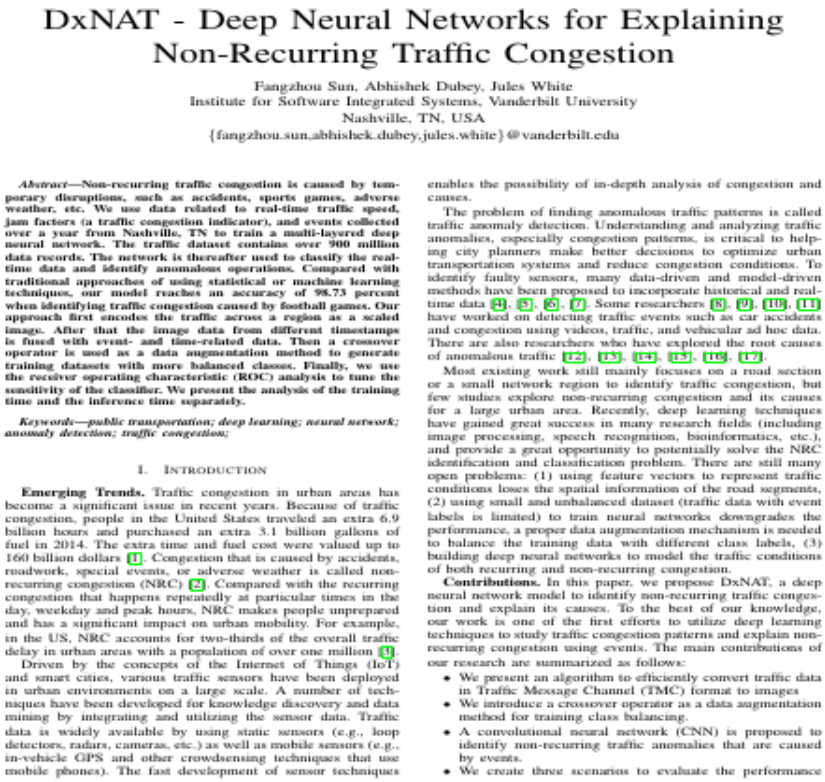
In this approach, we still use the same set of scientific articles from arXiv we used in the first chapter of this report. In order to train an image classification model, we need images of the papers. In fact, we decided to only generate one image per document, which is the image of the first page of the document, and this for several reasons. First, we think that this first page contains a lot of geometrical properties that might be helpful to determine the document class. We will (mostly always) find in the first page the title of the paper, the authors, the title announcing the abstract, the abstract itself, and sometimes other information that are specific to the journal or conference where the paper is published (for instance, the document classes associated to some journals or conferences related to the Association for Computing Machinery have a specific line describing the ACM category to which the paper belong to). Secondly, using multiple images of a single paper would be technically harder, and would require using more advanced learning techniques. For instance, if we want to feed our model image by image while still being able to proceed document by document, we would need to use models that have a memory about the past such as R-CNN (use of both Recurrent Neural Networks and Convolutional Neural Networks in a same architecture). We might not need such complex architectures to train our model. We could still consider these if the results are not satisfactory at the end.

Technically speaking now, we decided to convert, for each paper, the first page of the PDF version of it as an image of size 512 x 512 pixels in Portable PixMap format (`.ppm`), with a Digit Per Inch (D.P.I.)

resolution of 200. The choice of the size is justified by the fact that we wanted re-scaled squared images of uniform size to handle potential differences in page formats among all documents. This re-scaling combined with the resolution quite strongly reduces the clarity of the images. However, what matters the most is not the textual content of the papers, but the geometrical patterns we will be able to identify in the images.

We also needed to modify a bit the image, by erasing the band of text that arXiv automatically adds to the PDF of the papers, which most of the time contains an abbreviation of the document class we try to predict, which would add an important bias to our classification model (since we want our model to be able to consider papers coming from any platform, and not necessarily from arXiv).

An example of such a generated image from the PDF of a paper taken randomly in our dataset is shown in Figure 3.1, exactly as they bill be taken as input by the model (or, at least, by the model pipeline, since an extra step will be presented later).



DxNAT - Deep Neural Networks for Explaining Non-Recurring Traffic Congestion

Fangzhou Sun, Abhishek Dubey, Jules White
Institute for Software Integrated Systems, Vanderbilt University
Nashville, TN, USA
{fangzhou.sun,abhishek.dubey,jules.white}@vanderbilt.edu

***Abstract*—Non-recurring traffic congestion is caused by temporary disruptions, such as accidents, sports games, adverse weather, etc. We use data related to real-time traffic speed, jam factors (a traffic congestion indicator), and events collected over a year from Nashville, TN to train a multi-layered deep neural network. The traffic dataset contains over 900 million data records. The network is thereafter used to classify the real-time data and identify anomalous operations. Compared with traditional approaches of using statistical or machine learning techniques, our model reaches an accuracy of 98.73 percent when identifying traffic congestion caused by football games. Our approach first encodes the traffic across a region as a scaled image. After that the image data from different timestamps is fused with event- and time-related data. Then a crossover operator is used as a data augmentation method to generate training datasets with more balanced classes. Finally, we use the receiver operating characteristic (ROC) analysis to tune the sensitivity of the classifier. We present the analysis of the training time and the inference time separately.**

***Keywords*—public transportation; deep learning; neural network; anomaly detection; traffic congestion;**

I. Introduction

Emerging Trends. Traffic congestion in urban areas has become a significant issue in recent years. Because of traffic congestion, people in the United States traveled an extra 6.9 billion hours and purchased an extra 3.1 billion gallons of fuel in 2014. The extra time and fuel cost were valued up to 160 billion dollars [1]. Congestion that is caused by accidents, roadwork, special events, or adverse weather is called non-recurring congestion (NRC) [2]. Compared with the recurring congestion that happens repeatedly at particular times in the day, weekday and peak hours, NRC makes people unprepared and has a significant impact on urban mobility. For example, in the US, NRC accounts for two-thirds of the overall traffic delay in urban areas with a population of over one million [3].

Driven by the concepts of the Internet of Things (IoT) and smart cities, various traffic sensors have been deployed in urban environments on a large scale. A number of techniques have been developed for knowledge discovery and data mining by integrating and utilizing the sensor data. Traffic data is widely available by using static sensors (e.g., loop detectors, radars, cameras, etc.) as well as mobile sensors (e.g., in-vehicle GPS and other crowdsensing techniques that use mobile phones). The fast development of sensor techniques enables the possibility of in-depth analysis of congestion and causes.

The problem of finding anomalous traffic patterns is called traffic anomaly detection. Understanding and analyzing traffic anomalies, especially congestion patterns, is critical to helping city planners make better decisions to optimize urban transportation systems and reduce congestion conditions. To identify faulty sensors, many data-driven and model-driven methods have been proposed to incorporate historical and real-time data [4], [5], [6], [7]. Some researchers [8], [9], [10], [11] have worked on detecting traffic events such as car accidents and congestion using videos, traffic, and vehicular ad hoc data. There are also researchers who have explored the root causes of anomalous traffic [12], [13], [14], [15], [16], [17].

Most existing work still mainly focuses on a road section or a small network region to identify traffic congestion, but few studies explore non-recurring congestion and its causes for a large urban area. Recently, deep learning techniques have gained great success in many research fields (including image processing, speech recognition, bioinformatics, etc.), and provide a great opportunity to potentially solve the NRC identification and classification problem. There are still many open problems: (1) using feature vectors to represent traffic conditions loses the spatial information of the road segments, (2) using small and unbalanced dataset (traffic data with event labels is limited) to train neural networks downgrades the performance, a proper data augmentation mechanism is needed to balance the training data with different class labels, (3) building deep neural networks to model the traffic conditions of both recurring and non-recurring congestion.

Contributions. In this paper, we propose DxNAT, a deep neural network model to identify non-recurring traffic congestion and explain its causes. To the best of our knowledge, our work is one of the first efforts to utilize deep learning techniques to study traffic congestion patterns and explain non-recurring congestion using events. The main contributions of our research are summarized as follows:

- We present an algorithm to efficiently convert traffic data in Traffic Message Channel (TMC) format to images
- We introduce a crossover operator as a data augmentation method for training class balancing.
- A convolutional neural network (CNN) is proposed to identify non-recurring traffic anomalies that are caused by events.
- We create three scenarios to evaluate the performance

Figure 3.1: Example of an image generated from the PDF of a paper randomly taken in our dataset

Once all the images are generated, we have a set of images which is ready to be fed to a computer vision learning algorithm. We already have the associated document classes from the first chapter.

3.2 Defining the architecture of the computer vision models and testing it with a 2-class classification problem

Before diving into our multi-class classification task, we will demonstrate the relevance of the use of CNNs to predict the document class of scientific papers using the first image of the document. To do so, we will train

multiple CNN models with some binary outputs, by randomly selecting two document classes and feeding the corresponding images to the model.

Before training the model, we must load the data and do the last pre-processing step we talked about earlier. This step consists of transforming the R.G.B. image into a gray-scaled image. We do this to make sure that the model focuses on the geometrical patterns, and not on the patterns he could find in the used colors.

Also, in order to have results that make sense and to be sure that our models will put an equal attention to the concerned classes, we do some oversampling before training the models. However, we will not directly duplicate the data as we were doing the first chapter. In fact, since we are dealing with images now, it would require too much disk space to directly store the duplicated images. Instead, we are using a method from `keras.preprocessing.image.ImageDataGenerator`, which is called `flow_from_dataframe`. This method takes several arguments as input, and especially a `pandas DataFrame` containing both image names and their associated ground-truth, as well as the path in which the images are stored. This therefore solves our problem of disk space saving, since we just have to construct the `DataFrame` with all our images and their ground-truth, and then do some oversampling on this `DataFrame`.

Finally, since we are randomly taking two classes over our 33 document classes, it is possible that we end up choosing two classes with a small initial number of associated papers. As a consequence, we might not really improve the total number of samples by doing oversampling, and the model might not have the opportunity to converge towards stable results by looking at the images only at once. This is why we decided to show the data to the model five times, to be sure that the obtained results are stable.

The data are now ready to be presented to the CNN model. But we have not defined the architecture of this model yet. There were a lot of several possibilities, as the literature in this domain is well provided. Some standard architecture that became famous are LeNet [LBBH98], AlexNet [KSH12], VGG16 [SZ14] or more recently ConvNext [LMW+22]. The more recent the models, the more parameters they contain. The most recent models have shown excellent results on image classification tasks. But they were designed to handle images with really complex patterns, mostly in the context of pattern recognition or object detection over images from the real-world (mostly pictures). The task we are trying to do does not really relate to this kind of context, and we make the assumption, at least from our human perception, that geometrical patterns over an image of a PDF document are not as complex as the one used to classify animal images for instance. This observation, combined with the important amount of data we are processing, leads us to the use of less complex models to do our classification task.

This is why we decided to use an architecture only made of three layers. Each of these layers will be made of a convolutional layer (i.e., a set of filters), an activation function such as the REctified Linear Unit (RELU), which is defined as $f(x) = \max(0, x)$, a down-sampling operation, which is initially inspired by the brain's functioning where regions of the visual field are associated with "receptive fields" in the visual cortex (we will use the max-pooling operation, which consists in taking the maximum value observed on a sub-part of fixed size of an image and assign this value to this subpart, or to a sub-part of reduced size), and finally a dropout operation, which consists in randomly deactivating a neuron to make some regularization and avoid overfitting.

Here are the successive layers of the model we built:

1. **Zero-th layer**. The zero-th layer is apart from the following ones, since it must take into account the initial size of the images and since it will learn the first basic patterns of the images. We will therefore not apply down-sampling and dropout operations on it.
   - 2-Dimensional convolutional layer, with 32 filters, kernel size of $(3, 3)$.
   - RELU activation function.
2. **First layer**.
   - 2-Dimensional convolutional layer, with 64 filters, kernel size of $(3, 3)$.
   - RELU activation function.
   - Max-pooling operation, with kernel size of $(4, 4)$.

- Dropout operation, with probability of 25%.

3. **Second layer**.
   - 2-Dimensional convolutional layer, with 64 filters, kernel size of $(3, 3)$.
   - RELU activation function.
   - Max-pooling operation, with kernel size of $(4, 4)$.
   - Dropout operation, with probability of 25%.
4. **Third layer**.
   - 2-Dimensional convolutional layer, with 128 filters, kernel size of $(3, 3)$.
   - RELU activation function.
   - Max-pooling operation, with kernel size of $(4, 4)$.
   - Dropout operation, with probability of 25%.

We end up having a set of 128 filters, which can be interpreted as images (they are matrices of numbers). They all share the same dimension (same length and width), which depends on the previous operations (especially on the down-sizing operations). But we want to do image classification, which requires, for each image given as input to the model, an output which is a class (in the case of this section, a binary class), and not a set of filters. We therefore need to add an extra layer to the model, composed of the following elements:

- Flatten operation. This operation consists in transforming the 2-D arrays that embody the final filters into a 1-D array (or vector).
- Dense layer of size 32. A dense layer is a fully-connected layer, where each neuron is connected to every neuron of the previous layer.
- RELU activation function.
- Dropout operation, with probability of 25%.
- Dense layer of size 1.
- Sigmoid activation function. The sigmoid activation function is defined as $f(x) = (1 + \exp^{-x})^{-1}$. It is an activation function that is adapted to binary classification, since it tends towards the value 1 when x takes increasing positive values, and towards the value 0 when x takes decreasing negative values.

The output of the function will therefore be a score between 0 and 1. To learn, the algorithm need to know which optimization algorithm to use (the one that will update the weights of the model, i.e., the values of the filters), as well as which loss function to use (the function that computes how far is the model prediction from the actual ground-truth). We chose to use standards parameters for binary classification, which are the ADAM optimization algorithm, and the Binary Cross-Entropy loss (BCE), which is defined as:

$$\text{BCE} = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

where y denotes the ground-truth of a given sample, and $\hat{y}$ the prediction for this sample. Since the predicted score is a continuous value between 0 and 1, we simply need to threshold it at 0.5 to predict the class of non observed samples.

By using this architecture, we end up having around 330 000 parameters for images of size 512 x 512 (over 3 color channels). We therefore succeeded in building an architecture with a relatively small number of parameters, compared of the standards model we mentioned earlier: for instance, the VGG16 architecture applied on images of size 224 x 224 (over 3 color channels) has a total number of parameters around

138 000 000, i.e., 418 times more than our architecture.

The results obtained for a few models with the architecture presented above are shown in Table 3.1, with two metrics we already presented above in the report: the accuracy, with 10^{-2} precision, and the F1-Score, with unit precision (thus, equal to 100% if above 99.5%). We also ensured ourselves to make a train and test split before doing the oversampling over the `DataFrame` (we therefore created two `DataFrame` objects, one with 80% of the images, and the other one with 20% of it, and then we did the oversampling over both of them).

Table 3.1: Performances of several CNN models trained with images of papers associated with two randomly selected document classes

		Train set		Test set	
Class name 1	Class name 2	Accuracy	F1-Score	Accuracy	F1-Score
bmvc2k	llncs/svmult/svproc	100.00%	100%	100.0%	100%
pos	book	99.77%	100%	99.57%	100%
acmart	imsart	99.78%	100%	98.79%	99%
aa	llncs/svmult/svproc	99.92%	100%	99.92%	100%
mdpi	cms	99.31%	99%	100.00%	100%
amsart/amsproc	aa	99.91%	100%	99.90%	100%
acmart	amsart/amsproc	99.70%	100%	99.85%	100%
revtex4	achemso	99.53%	100%	98.68%	99%

We immediately observe that the model succeeds in achieving the classification task. In fact, it got almost perfect results on a few examples (where the classes were taken randomly). In particular, we can notice that it also succeeded in correctly classifying an heterogeneous document class, on the model presented in the second row of the table (`pos` and `book`). Sometimes we observe test scores that are better to train scores: this is due to the fact that we sometimes only have a few samples to test (for instance, if we have two document classes with around 100 representatives, we end up having only 40 samples in the test set).

These results thus definitely confirm the relevance of the use of a computer-vision approach for our document-class classification task. We will now present the model we use to achieve this multi-class classification, as well as the results it gives once trained.

3.3 First modeling: single multi-class classification model

In this first computer-vision based modeling, we will simply adapt the classification task we did on random forest to our CNN architecture.

The data are pre-processed and oversampled exactly as they were in the previous section. The architecture of the model is almost the same as the one from the previous section, with some few modifications. Those modification only concern the last (fourth layer) :

- Flatten operation.
- Dense layer of size 32.
- RELU activation function.
- Dropout operation, with probability of 25%.
- Dense layer of size 33 (the number of classes).
- Softmax activation function.

We only changed the two last items of this fourth layer, to adapt the architecture to a multi-class classification problem. The first change consists in having 33 outputs instead of 2, and the second change in changing the final activation function. In fact, we saw earlier that Sigmoid was adapted to binary-classification problems, since it outputs a single value between 0 and 1 from a single (real) input value. But now, we have 33 different values as input. The Softmax function $\sigma : R^{N_c} \to (0,1)^{N_c}$, where N_c denotes the number of classes (in our context, 33), is defined as follows:

$$\sigma(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{N_c} e^{x_j}} \text{ for } i = 1, \ldots, N_c \text{ and } \mathbf{x} = (x_1, \ldots, x_{N_c}) \in R^{N_c}$$

This Softmax function therefore returns, for each class, a "confidence" score between 0 and 1. We also observe that:

$$\sum_{i=1}^{N_c} \sigma(\mathbf{x})_i = \frac{\sum_{i=1}^{N_c} e^{x_i}}{\sum_{j=1}^{N_c} e^{x_j}} = 1$$

Therefore, this "confidence" score can also be interpreted as a probability distribution. To make some new predictions, we only need to choose the class associated with the maximum probability in the vector. Formally, if we denote $\hat{y}$ as the prediction, we have:

$$\hat{y} = \arg \max_{j \in \{(1,\ldots,N_c\}} \sigma(\mathbf{x})_j$$

Finally, there is one more modification we must do to adapt the model to our multi-class classification problem, which concerns the loss function. We previously used the Binary Cross-Entropy, which is in fact an adapted version of the Cross-Entropy (CE) for binary classification. We thus now can use the Cross-Entropy, which is defined as:

$$\text{CE} = -\sum_{i=1}^{N_c} y^i \log(\hat{y}^i)$$

where y^i is the ground-truth of a given sample for class i (basically, 1 if the sample belongs to this class, 0 either), and $\hat{y}_i$ its prediction for class i, i.e., the probability score outputted by the Softmax function for class i.

We can then train the model by using `ImageDataGenerator` objects as in the first section. Once the training phase is over, we can evaluate the performance of the model. First, we observe a global accuracy of 98.42% on the training set, and of 93.31% on the test set. We can also have a look at more precise metrics (precision, recall and F1-score) class by class. Those results are presented in Table 3.2, with the metrics already presented above in the report.

Those results are globally very good. There is only a minor proportion of miss-classifications on both train and test sets. But what we do observe on the results, and in fact especially on the test set (which is the one that really matters since we can only measure how the model generalizes on it), is that most of classes have excellent results, and that the overall performance of the model is severely badly impacted by only a few classes. Among them, we have the class `other`, the class `book` and the class `report/wlscirep`. They all have a F1-score below 70% on the test set.

This is an observation that we already made on the random forest classifier presented in the first chapter: these classes contain a lot of heterogeneity since, despite the other classes, the users can personalize those almost as much as they want (this is not completely true for class `report/wlscirep`, where only `report` is used as a "generic" document class, but it was still merged with `wlscirep` due to their similarity in terms of source code).

Table 3.2: By class performance metrics for the CNN model with 33 classes

	Train set			Test set		
Document class	Precision	Recall	F1-Score	Precision	Recall	F1-Score
aa	100%	100%	100%	100%	100%	100%
aastex	98%	98%	98%	96%	94%	95%
aastex6/aastex61/aastex62	98%	99%	99%	93%	99%	96%
achemso	99%	100%	100%	98%	98%	98%
acmart	99%	98%	99%	96%	99%	98%
amsart/amsproc	94%	99%	96%	77%	98%	86%
bmvc2k	100%	100%	100%	100%	100%	100%
book	99%	100%	100%	79%	39%	52%
cms	100%	100%	100%	100%	100%	100%
elsarticle	98%	98%	98%	98%	97%	97%
emulateapj	99%	100%	99%	98%	97%	97%
eptcs	100%	100%	100%	97%	97%	97%
iau/jfm	100%	100%	100%	100%	100%	100%
ieeeconf/ieeetran	93%	99%	96%	82%	99%	90%
imsart	99%	100%	99%	97%	85%	91%
iopart/jpconf	99%	100%	99%	99%	99%	99%
lipics	100%	100%	100%	100%	94%	97%
llncs/svmult/svproc	99%	98%	99%	95%	97%	96%
mdpi	100%	100%	100%	100%	97%	99%
mn2e/mnras	100%	100%	100%	99%	99%	99%
other	93%	68%	79%	64%	68%	66%
pos	100%	100%	100%	100%	98%	99%
report/wlscirep	99%	99%	99%	60%	78%	68%
revtex4	90%	97%	93%	87%	96%	91%
scrartcl	97%	97%	97%	97%	83%	90%
siamart171218	99%	100%	100%	96%	93%	95%
siamltex	99%	100%	100%	93%	94%	93%
sig	99%	100%	100%	99%	94%	96%
sigma	100%	100%	100%	100%	100%	100%
spie	100%	100%	100%	100%	98%	99%
svjour/svjour3	98%	99%	99%	95%	97%	96%
webofc	100%	100%	100%	100%	100%	100%
ws	100%	100%	100%	100%	90%	95%
Mean	**98%**	**98%**	**98%**	**94%**	**93%**	**93%**

This is why the idea of finding a way to treat the heterogeneous classes separately emerged. Succeeding in doing so could not only improve the overall performance of the model by getting rid of the miss-classifications in the concerned class (the "False Negatives" cases of this class, described earlier), bit also by getting rid of the miss-classifications in the other classes (the "False Positive" in the other classes, i.e., the images predicted to belong to other classes while belonging one of the heterogeneous classes). It would in fact simplify the multi-class classification task. This is what we are studying in the next and last section of this chapter.

3.4 Second modeling: reject option and multi-class classification model on non heterogeneous document classes

The aim of this last modeling is to find a technique allowing us to put apart the heterogeneous classes so that the multi-class classification becomes simpler. The idea of judging a subset of a dataset as "irrelevant" for a machine learning model is something that has already been studied under the name of **reject option**. It takes its name from the idea that we want our model to be able to reject a prediction (or a training sample)

if it belongs to the irrelevant subset of our data (with respect to the classification task we are considering, the data are not irrelevant themselves).

By studying the subject through a relatively recent, quite exhaustive survey [HPVdP+21], it turned out that we already integrated a certain kind of reject option in our previous models, by regrouping all papers belonging to `article` and all document classes which didn't have enough representatives to be considered in the learning. But they are still two other significant document classes that are still not considered in the `other` class.

In this paper, the authors distinguish two types of rejections. First, the ambiguity rejection, which corresponds to the situation where our learned model didn't put attention to certain regions of the instance space. Concretely, it could correspond to a document which contains properties of two distinct classes, and where the model didn't learn other discrimination criteria to clearly know to which class it belongs. Then we have the novelty rejections, where the classification model is given a new sample that is too dissimilar to what has been observed. This second case is particularly of concern for our problem, since it is linked to the notion if heterogeneity we already mentioned several times (in test, we can observe images of known heterogeneous classes that are very dissimilar, but even images belonging to classes we don't even know with respect to the train set).

Moreover, by integrating a supplementary class aiming at gathering all the data to be rejected in a classification model, we make the choice of an integrated architecture, where the rejector is directly integrated in the classification model. This phenomenon is in opposition with our will to make the classification task simpler. We thus have two remaining options: a dependent or a separated rejector. The dependent rejector uses the output of the predictor (the classification model), to decide whether or not to keep the prediction or not. This led us to choose a separated rejector. By separated, it is meant separated from the predictor. Unfortunately, choosing this architecture directly prevents the rejector from doing ambiguity rejection. This is something that is directly linked to the fact that it is separated from the rejector: since the rejector does not know what predictions the model is making, it cannot know whether the decision will be too ambiguous or not. But as we told earlier, what interests us the most is novelty rejection, which can be done with separated rejectors.

Concretely speaking, this kind of rejector most of the time takes the form of a filter acting upstream of the classification model. We will therefore have two models to consider : the rejector, and the multi-class classification model. The idea is that we will be able by training the multi-class model only on the non heterogeneous classes by feeding it only with the samples that are not filtered by the rejector. This imply the use of a simultaneous learning : each model will have its own architecture, and even its own datasets (which will be presented later), in opposition with sequential learning, where both rejector and predictor are trained at the same time (this is what we were doing with the ;`other` class in the modeling of last section).

The paper also presents two kind of models that can be used as rejector: the generative models and the discriminative models. The generative models are mainly based on some probabilistic approaches, by inferring the posterior distribution (in this case, whether it must be rejected or not), thanks to the prior knowledge and the joint distribution. Bayesian classification allows such approaches, oftenly combined with assumptions about the behavior of the random variables when evolving in complex systems, such as hidden Markov models. This kind of modeling is not very well suited for our dataset, since we are dealing with images. It could be possible to use them, but the second approach is more adapted. In fact, the discriminative approach directly focuses in finding the relationship between the input and the target classes. Methods that can do such tasks are Support Vector Machines, decision trees classifiers, and also neural networks. This is perfect for us, since we do have a tool based on neural networks that is well suited in the context of images classification : CNNs. This is therefore the architecture we will use to design our rejector: a CNN doing image classification with two (binary) possible output : rejected, and thus directly classified as too heterogeneous to be associated with a specific class, or not rejected, and then provided to the multi-class model.

We will now present in the next two sub-sections the rejector model and the multi-class classification model.

3.4.1 Rejector

The generation and pre-processing of the data are basically the same as the ones we used in the previous computer-vision based models. The only difference we will observe is the associated target labels that now become either 0 or 1. Especially, all documents that come from the original classes `other`, `report` or `book` will be assigned to label 1 (to be rejected), and all the other ones will be assigned to label 0 (not to be rejected).

Now regarding the architecture of the model, we could directly take the one we built in the second section of this chapter, where we already did binary-classification with a CNN to validate the computer-vision based approach. We will use the same architecture, but with slightly different parameters. Here are the successive layers of the rejector that will be trained:

1. **Zero-th layer**. The zero-th layer is apart from the following ones, since it must take into account the initial size of the images and since it will learn the first basic patterns of the images. We will therefore not apply down-sampling and dropout operations on it.
    - 2-Dimensional convolutional layer, with 64 filters, kernel size of $(3, 3)$.
    - RELU activation function.
2. **First layer**.
    - 2-Dimensional convolutional layer, with 64 filters, kernel size of $(3, 3)$.
    - RELU activation function.
    - Max-pooling operation, with kernel size of $(4, 4)$.
    - Dropout operation, with probability of 25%.
3. **Second layer**.
    - 2-Dimensional convolutional layer, with 128 filters, kernel size of $(3, 3)$.
    - RELU activation function.
    - Max-pooling operation, with kernel size of $(4, 4)$.
    - Dropout operation, with probability of 25%.
4. **Third layer**.
    - 2-Dimensional convolutional layer, with 256 filters, kernel size of $(3, 3)$.
    - RELU activation function.
    - Max-pooling operation, with kernel size of $(4, 4)$.
    - Dropout operation, with probability of 25%.
5. **Fourth layer**.
    - Flatten operation.
    - Dense layer of size 64.
    - RELU activation function.
    - Dropout operation, with probability of 25%.
    - Dense layer of size 1.
    - Sigmoid activation function.

Basically, we changed the number of filters in each layer and the number of neurons in the first dense layer. Even though we only have 2 classes, we are here trying to achieve a hard task, that we specifically separated from the multi-class model since it was too hard to handle everything at the same time. Facing a more difficult task needs to find a more complex modeling, and improving the number of parameters (can) help the model do a good separation. With this architecture, we end up having a little more than 1.200.000 parameters, which remains reasonable compared to the standard architectures already presented.

The architecture being defined, we can now train the model. As usual, we splitted the dataset into a train set and a test set, and we then applied oversampling to have balanced classes. We obtain an overall accuracy of 91.47% on the train set, and 90.52% on the test set. A more detailed by class analysis with usual metrics is shown in Table 3.3.

Table 3.3: By class performance metrics for the CNN model used as a rejector (binary classes)

	Train set			Test set		
Class	Precision	Recall	F1-Score	Precision	Recall	F1-Score
Not to be rejected (0)	86%	99%	92%	85%	99%	91%
To be rejected (1)	99%	84%	91%	98%	82%	90%
Mean	**92%**	**91%**	**91%**	**92%**	**91%**	**90%**

Given the difficulty of the task, the results are satisfactory. We see that the model is better at detecting the images that must not be rejected than the ones that must. We can see it thanks to the fact that recall of class "to be rejected" is close to one, and therefore the number of false positives is close to zero (there are really few images predicted as heterogeneous while actually not being). We also see it with the recall of the "not to be rejected" class: since it is close to one, then the number of false negatives is close to zero (there are really few images classified as not being part of not heterogeneous classes while currently being).

The model is not that accurate for the other class, where we do have more false positives for class not heterogeneous and more false negatives for class heterogeneous. This will obviously have an impact on the performance when using both predictor and multi-class model on new data, since there will be samples presented to the second model that do not belong to any classes of the model (since they are in reality in the heterogeneous class). Still, the performance stay interesting, since the accuracy of the heterogeneous class is 82.45%, which is better than the best accuracy among the heterogeneous classes in the version with 33 classes (with an exception for `report`, which was merged with a standard class which has really good performance as we are going to see in the next subsection).

3.4.2 Multi-class classification model (with 31 classes)

Now that the heterogeneous classes are put apart with the rejector, we can train a CNN to do our document class classification without those. To be more specific, we are removing for the list of 33 classes, all the images that came from the **original** classes (44 classes) `other`, `report` and `book`. We just have to remove `other` and `book` on the set of 33 classes since these are the same as the original ones. It is not the case for `report` class, which was merged with `wlscirep`. To solve this, we just have to remove from the images of class `report/wlscirep`, all the images that were in class `report` at the beginning. As a consequence, we end up having 31 possible classes for our classification problem.

Regarding the architecture, we will use the exact same architecture as the one presented in the third section, except that the last dense layer will have 31 output neurons instead of 33. The rest remains unchanged.

We can now train the model. As always, we do a train and test split of the dataset (80% and 20% respectively), and we then apply oversampling to it. We obtain an overall accuracy of 99.20% on the training set and 97.15% on the test set. A more detailed by class analysis with usual metrics is shown in Table 3.4.

Table 3.4: By class performance metrics for the CNN model with 31 (non heterogeneous) classes

	Train set			Test set		
Document class	Precision	Recall	F1-Score	Precision	Recall	F1-Score
aa	99%	100%	99%	99%	100%	99%
aastex	99%	96%	98%	99%	91%	95%
aastex6/aastex61/aastex62	97%	99%	98%	94%	99%	97%
achemso	100%	98%	99%	100%	98%	99%
acmart	99%	98%	98%	91%	99%	95%
amsart/amsproc	99%	99%	99%	85%	99%	92%
bmvc2k	100%	100%	100%	100%	100%	100%
cms	100%	100%	100%	100%	100%	100%
elsarticle	94%	99%	96%	97%	98%	98%
emulateapj	98%	100%	99%	100%	99%	99%
eptcs	100%	100%	100%	97%	97%	97%
iau/jfm	100%	100%	100%	100%	100%	100%
ieeeconf/ieeetran	99%	98%	99%	90%	98%	94%
imsart	100%	100%	100%	92%	94%	93%
iopart/jpconf	100%	100%	100%	95%	99%	97%
lipics	100%	100%	100%	98%	94%	96%
llncs/svmult/svproc	100%	100%	100%	97%	97%	97%
mdpi	98%	98%	98%	100%	97%	98%
mn2e/mnras	99%	99%	99%	100%	100%	100%
pos	99%	99%	99%	100%	98%	99%
wlscirep	100%	100%	100%	100%	98%	99%
revtex4	98%	98%	98%	97%	98%	97%
scrartcl	99%	99%	99%	97%	91%	94%
siamart171218	99%	99%	99%	96%	96%	96%
siamltex	100%	100%	100%	96%	96%	96%
sig	100%	97%	98%	100%	90%	95%
sigma	100%	100%	100%	100%	100%	100%
spie	100%	100%	100%	100%	98%	99%
svjour/svjour3	98%	98%	99%	100%	95%	97%
webofc	100%	100%	100%	98%	100%	99%
ws	100%	100%	100%	100%	92%	96%
Mean	**99%**	**99%**	**99%**	**97%**	**97%**	**97%**

We end up having excellent results now that the heterogeneous classes are not in the model anymore. Not only their absence improves total accuracy, but they were also impacting the performance of other classes (by being predicted as part of some non heterogeneous classes, while actually belonging to one of the heterogeneous). For instance, class `revtex4` had for precision, recall and F1-score, on the test set, respectively 87%, 96% and 91% in the model with 33 classes, and upgraded to 97%, 98% and 97%. We still observe some decreases in performance, but not as high as the increases: it is most of the time a matter of one or two percents. For instance, `webofc` had for precision, recall and F1-score, on the test set, 100% on each score, and decreased to respectively 98%, 100% and 99%.

Finally, it is hard to tell if the combination of both rejector and multi-class classification model without non heterogeneous classes performs significantly better than the model with all the 33 classes. But we showed that this was this heterogeneity that was preventing the model to be very accurate, and that there exists quite performant methods to detect these too heterogeneous images.

Conclusion and improvements

To sum up, we have successfully completed the task of classifying document classes of scientific papers. We first showed the relevance of such a task, by extracting five relatively simple features, which were supposed to be sufficient to exhibit characteristics allowing classification. We conducted a statistical study on these variables to achieve this. Then we trained a random forest model with this data, which not only confirmed that the classification task was feasible with more complex modeling, but also that a geometric approach was relevant to the task being performed.

This is how we studied the use of computer vision methods for the classification of these document classes. In particular, the use of CNN made it possible to complexify the modeling, based on geometric characteristics, while limiting the number of parameters to be trained. The use of this type of model has proved effective for our classification task, but we have observed that certain classes, whose heterogeneity is too great, limit the model's performance. This is why we conceptualized a last modeling, based on the use of a rejector, allowing to filter these too heterogeneous classes and to result in a multi-class model containing only homogeneous classes. The performance obtained by the two models are very satisfactory, given the difficulty of the task to be performed.

However, it is certain that the limited time of the project did not allow us to make a thorough study of the subject, and that improvements could be imagined, particularly for computer-based vision models, since these were the ones with the most promising results. First, we chose to use a very simple CNN architecture to make our classification. We made this choice based on the principle that given the first results, it was not necessarily to choose more modern architectures containing more parameters. However, we then chose to use a rejector because the performance was too negatively impacted by too heterogeneous classes. But perhaps the use of more complex modeling could help to cope with this difficulty related to heterogeneity, and to keep only one classification model, instead of having to set up two successive CNNs.

Considering now that we would like to keep this modeling by CNNs with few parameters, we only looked at the novelty rejections, but not ambiguity rejections. It might be interesting to set up a system to avoid making a decision if a scientific article is too close to two (or more) of the 31 CNN classes. In particular, we could use a dependent-rejector architecture, based on the probability distribution explained in chapter two. This could be done by quantifying how far is the probability distribution of such an example from the distributions of the other predicted examples (especially the ones of the closest classes). In particular, we could use the Local Outlier Factor [BKNS00], which is based on the local density of each example with respect to the others.

Finally, it would be interesting to exploit the prediction of document classes to perform other, more complex, tasks. For instance, we could try to see the performance of extracting theorems, definitions and other mathematical contents from scientific article (in the context of the project TheoremKB, already presented above), by combining what we know about how document classes structure the mathematical contents and the document classes we extract thanks to our classification models.

Bibliography

[BGSF10] Jöran Beel, Bela Gipp, Ammar Shaker, and Nick Friedrich. Sciplore xtract: Extracting titles from scientific pdf documents by analyzing style information (font size). In Mounia Lalmas, Joemon Jose, Andreas Rauber, Fabrizio Sebastiani, and Ingo Frommholz, editors, *Research and Advanced Technology for Digital Libraries*, pages 413–416, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BKNS00] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: Identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 93–104, New York, NY, USA, 2000. Association for Computing Machinery.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[HPVdP+21] Kilian Hendrickx, Lorenzo Perini, Dries Van der Plas, Wannes Meert, and Jesse Davis. Machine learning with a reject option: A survey, 2021.

[KLN10] Min-Yen Kan, Minh-Thang Luong, and Thuy Dung Nguyen. Logical structure recovery in scholarly articles with rich document features. *Int. J. Digit. Library Syst.*, 1(4):1–23, oct 2010.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[LBBH98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LMW+22] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s, 2022.

[Lop09] Patrice Lopez. Grobid: Combining automatic bibliographic data recognition and term extraction for scholarship publications. In Maristella Agosti, José Borbinha, Sarantos Kapidakis, Christos Papatheodorou, and Giannis Tsakonas, editors, *Research and Advanced Technology for Digital Libraries*, pages 473–474, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

Appendix A

Number of papers per document class, after class merges

Document class	Number of papers
other	31801
revtex4	17256
amsart/amsproc	15329
ieeeconf/ieeetran	9494
elsarticle	3903
llncs/svmult/svproc	3183
mn2e/mnras	3031
aastex6/aastex61/aastex62	2282
acmart	2034
aa	1541
svjour/svjour3	1434
iopart/jpconf	1228
emulateapj	1030
pos	559
scrartcl	484
aastex	445
report/wlscirep	394
iau/jfm	332
imsart	330
lipics	320
achemso	306
spie	284
ws	249
siamart171218	220
eptcs	183
mdpi	178
siamltex	175
webofc	174
sig	146
bmvc2k	134
cms	129
book	122
sigma	113

Appendix B

Distributions of the hand-designed features, for each document class (from feature 1 to feature 5)