Development of an End-to-End Deep Learning Pipeline

Matthias Nitsche, Stephan Halbritter
{matthias.nitsche, stephan.halbritter}@haw-hamburg.de
Hamburg University of Applied Sciences, Department of Computer Science
Berliner Tor 7, 20099 Hamburg, Germany

Abstract

Large scale machine learning pipelines come in a variety of flavors and practices. Dealing with a lot of parameters, moving large data quantities, working with large external data sources gets messy quickly. In this paper we present an infrastructure and data pipeline comparing different text classification models on a real world news data set. The infrastructure is provided by the CSTI Lab by ProfKai von Luck. We will explain our hardware setup and show benchmarking results on a basic image classification tasks with two ResNet variations. Since we have the opportunity to work with real-world German news data from the Deutsche Presse-Agentur (dpa), we analyze the dataset and show some applications. Further we will lay out a complete end to end dataflow solution with our data pipeline deployed with Kubernetes, MongoDB and HDFS. We will describe the actual data pipeline as directed acyclic graphs with computations such as transforming, preprocessing, embedding and modelling. Finally and as an outlook what can be done with the data, we present the results of a basic text classification task on the data set.

Keywords – Natural Language Processing, Hardware Benchmark, Knowledge Discovery Process, Data Pipeline, Text Analysis, Topic Modelling, GPU, Language Modeling, Text Classification

1 Introduction

Real world machine learning applications are tough and pose different challenges than working on reference datasets. The first problem is to actually identify what makes the data special and different to other data sets. Careful and clear statistical analysis is needed to understand the implications of the data, especially as there is no known point of reference or comparisson to other studies to compare the results to. Lastly preprocessing pipelines need to be carefully designed to account for the problems occuring in real world datasets such as unbalanced samples. Text elements contain HTML and newspaper specific syntactical elements that are not part of natural language. Fortunately the *Deutsche Presse-Agentur (dpa)* dataset contains a lot of metadata, such as categories, labels, keywords, author information and plenty of different text elements like the article, headlines or descriptions. It is vital to go through the different aspects of the data and comment on the strengths and weak spots.

In this paper we try to explore different aspects of an end-to-end machine learning pipeline that can solve several different tasks. Major components of such a pipeline contain the description of data that is going through it, infrastructure, parsing and normalization of text elements, depiction of CPU and memory bound preprocessing as well as GPU bound model training. Since the dpa data is highly unnormalized we go with intermediate representations, making it possible to replay parsing and normalization easily. We think of the data pipeline as *Directed Acylcic Graphs (DAGs)*, that share access to the data, where computations are nodes and files are shared on the edges. We do not

work directly with convential databases and decided to use simple file formats such as tsv and more advanced ones like Apache Parquet.

DAGs are a convenient abstraction when there are a lot of different computationally expensive jobs that co-depent on each other. Since we are not only interested in creating a pipeline without any task. we will show a very simple text classification based on multinomial Naive Bayes. This provides us a training ground for our follow up work where we will use state of the art deep learning models to learn a good classifier. In theory the pipeline can then be used for any task, such as image captioning or question answering.

The following sections are structured as follows: First we present our setup with a focus on the hardware and show the results of benchmarks for up to 10 GPUs. In the second part, we will present the dataset provided by dpa and make an exhaustive analysis. Section 3 lays out our ideal data pipeline and comments on parts that need further improvement. Also, we lay out our Kubernetes cluster setup with MongoDB and HDFS. Finally, we present a basic text classification task on a baseline model to demonstrate that it works. The idea behind this part is to show capabilities, not as a real benchmark. We close wiht a short presentation of our future directions and the upcoming text classification comparisson of state-of-the-art deep learning models.

2 Hardware Benchmarks

One main concern while building the hardware setup was about the effiency of multiple GPUs per computer. How much computational gain does an additional graphics card provide? How much is the overhead?

Various hardware setups were benchmarked to create points of reference to answer these questions. To allow a comparison of performance measurements beyond our own hardware, we used a set of benchmark scripts published by TensorFlow (2018). There are published results for hardware setups different from ours, which simplifies comparison, e.g. for further hardware purchases.

Mainly, we were interested to measure the impact of additional GPUs on the computational speed. The performance of the underlying model was not of interest, neither the specific model's speed or quality.

2.1 Hardware

2.1.1 Desktop computers

For daily prototyping and code development, two identical desktop computers are available. They are equipped each with an Intel Core i7 4.2 Ghz with 8 threads, 64 GB of RAM and 2 *NVIDIA 1080ti* GPUS with 11GB RAM. Ubuntu 16.04 LTS is used as the operating system. This setup is powerful enough to make first test runs on the experiments and run full training processes on smaller ones.

2.1.2 Renderfarm

These servers are dedicated for GPU-intense computations and are used for image processing as well as machine- and deep learning tasks. Resource demanding computations can be outsourced from the desktop computers to the these servers called *Renderfarms*. Each of the servers is configured with two Intel Xeon E5–2697V4 2.3 Ghz with 36 threads, 396 GB RAM and 10 *NVIDIA Quadro P6000* with 24GB RAM. CentOS is used as the operating system, running in a virtual machine.

2.2 Containerization

A very important aspect of our workflow is the use of containerization. Docker makes it possible to run code in an isolated, well-defined environment. A simple way of bundling specific software independently from the host system is very important for creating reproduceable experiments. It also makes it very easy to run code across different machines and even operation systems. This is especially useful in our setup of multiple smaller computers and only a few high-end servers.

So the development of the scripts, the pipeline, models and so on can occur on any computer, even everyday laptops. Here, the goal is not to train a complete model or preprocess all data, but to

get the basic workflow right. This means for example using small test sets to test that the desired functionality does work. If this hurdle is taken, docker allows to port the code without much changes to the Renderfarm servers, where the computation-intense training on all the data can take place.

nvidia-docker is a wrapper around docker and enables the docker container to use the host system's NVIDIA GPUs. This is essential for our use case. It also simplifies the installation of the base system, as only the NVIDIA GPU and CUDA drivers have to be installed. CUDA Toolkit and NVIDIA cuDNN provide GPU-accelerated libraries in general and for deep neural networks in particular and are already contained in nvidia-docker. Otherwise, these libraries have to be installed manually. This would mean to register an account at the NVIDIA homepage to be able to download all libraries and install them ignoring the operating system's package manager. This could lead to additional dependeny problems and from a system admin's perspective is a nuissance in itself. Updating to another version on one computer could break the usability of the dockerfile on another and so on. A general overview of the nvidia-docker architecture is depicted in Figure 1.

The benchmark scripts itself are written in Python and use the TensorFlow library, more details in the next section. Even though using containerization has some performance costs, we do run the benchmarks with docker. As we have seen, using docker is an essential part of our workflow and the way we run our training models. Running the benchmarks inside docker containers leads to benchmark results that are achieved under the real usage conditions of the hardware.

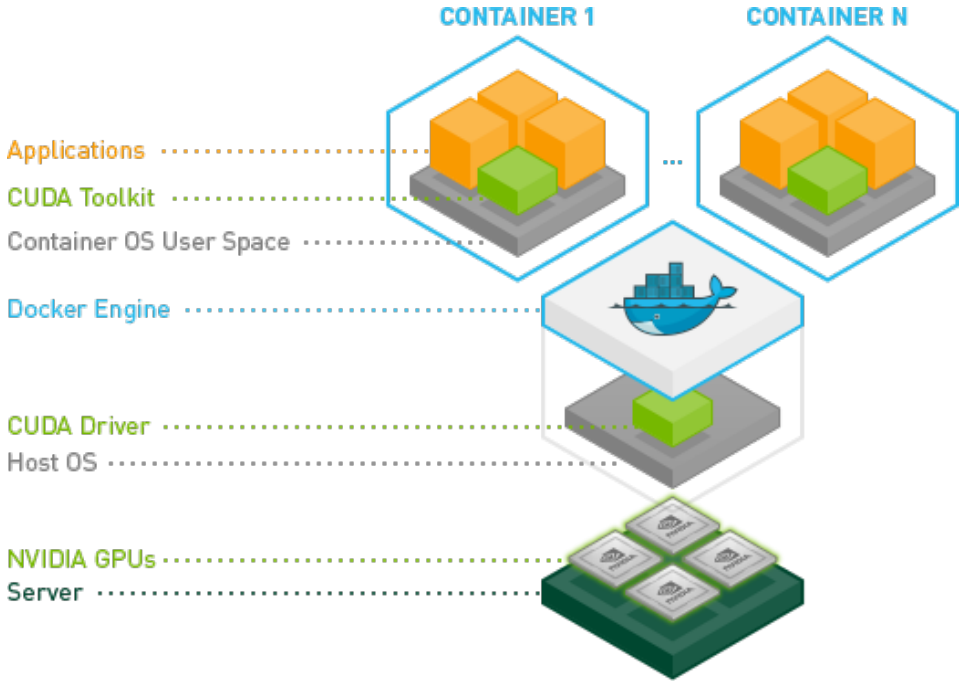

Figure 1: nvidia-docker system and setup (Source: github.com/NVIDIA/nvidia-docker)

2.3 Benchmarks

The *high performance benchmark scripts*[1] implement popular models of Convolutional Neural Networks. These are optimized for speed and parallel use of GPUs. We were running two of the implemented models for the benchmarks.

ResNet-50 and *ResNet-152v2* are both of the *ResNet* family of residual networks for image classification (He et al., 2015). They are interesting candidates for benchmarking, as there are a lot of slightly different variations with a rather large number of layers. The ResNet-152 won the ILSVRC 2015 classification challenge, having more than 8 times than the 22 layers of the previous year's winner, the Inception model.

[1] `https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks`

The main difference between the two models is their depth. As already implied by their names, ResNet-50 has 50 layers, while ResNet-152 with its 152 layers is a lot deeper. As a result it is more computational challenging and expected to run longer on a single epoch. ResNet-50 has around 3.8×10^9 *FLOPs (floating point operations)*, ResNet-152 three times as much, 11.3×10^9 FLOPs.[2]

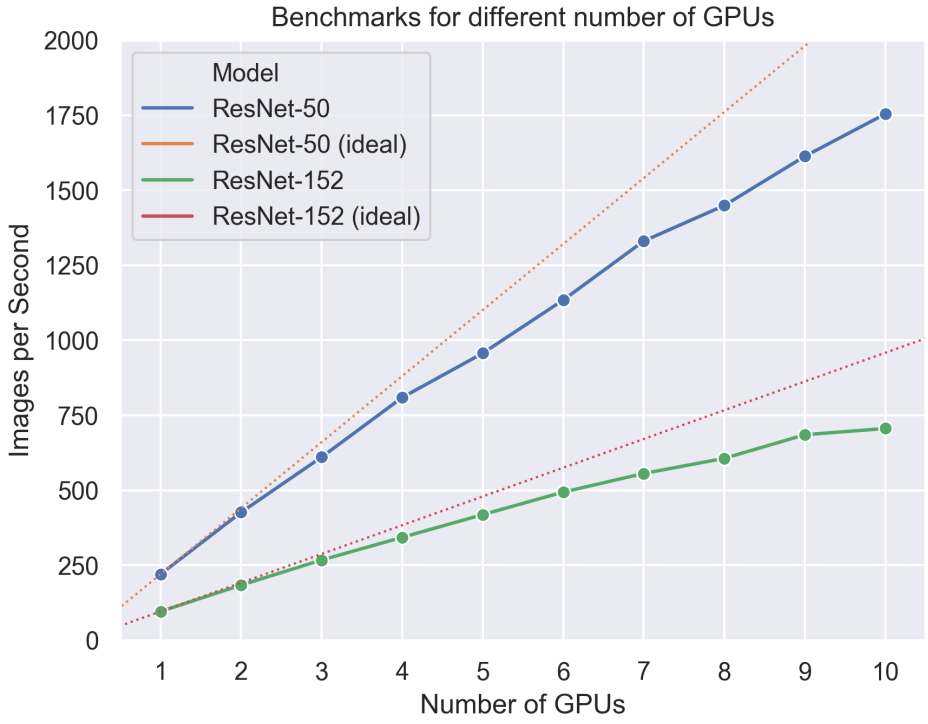

Figure 2: Comparison of benchmarks and ideal linear growths

As already stated, the requirements of these models do not lie in their classification quality, but in the possibility of parallizing the work on multiple GPUs.

Parallelization of the training process is achieved by copying the computational graph onto all GPUs. This way, forward passes and the computation of the corresponding gradients can happen in parallel. The results of each batch are then applied together (e.g. using gradient averaging) at a central point, before the parameters on the GPUs are updated and the next iteration can start. This ensures that the workload is split evenly onto all GPUs, making this method suitable to compare the influence of the number of GPUs on the speed of the training process.

In spite of using real data, synthetic data in the same shape as it could be expected from *ImageNet* is generated and used. This has the effect that there are no I/O operations from reading the data off the hard drive that could influence the performance. If our interest had been in benchmarking the whole pipeline including additional hardware besides the GPUs, e.g. the mainboard, memory or hard disk or an additional preprocessing step on the data, extra benchmarks using real data would have been required. In our test case we could do without this.

The benchmarks were executed on a virtual machine on one of the Renderfarm servers, running CentOS 7.6.1810 as the client's operating system. The benchmark script was executed in a Docker container. The Docker image is based on *tensorflow/tensorflow:nightly-gpu-py3*, which at the time came with TensorFlow 1.13, Python 3.5 and built-in support for NVIDIA CUDA.[3] For more details on why we use docker, see Section 2.2.

[2] Interestingly, this number is way lower than for other models with fewer layers. For example, VGGNet-19 has just 19 layers, but 19.6×10^9 FLOPS.

[3] `https://hub.docker.com/r/tensorflow/tensorflow`

2.4 Methodology

To provide comparable results, the batch size per GPU was set to a fixed value of 64. All other parameters of the script were left on their default values. The computation used synthetic ImageNet data for the reasons stated above.

For each of the possible number of GPUs, 1 up to 10, benchmarks were run with both of the ResNet models. Each benchmark run started from scratch. After creating the synthetic data and preparing the model, 10 warm-up batches were run, their results did not get included in the final results. For each of the next 100 batches, the mean of the number of processed images per second was calculated. The end result consists of the mean of these interim results and is listed in the column *Images/Sec* in Table 1.

2.5 Performance Comparison

There are published benchmarks for other hardware configurations on TensorFlow (2018). This allows a basic comparison between different GPU setups. For example, in Figure 3 the benchmark results of two other setups, published on TensorFlow (2018), are presented. They were benchmarked using the same parameters as described above, also using synthetic data on the ResNet-50 model.

The first setup consists of up to 8 *NVIDIA Tesla K-80*, which is a fairly popular graphics card used in cloud computing setups for deep and machine learning.[4] The *NVIDIA DGX-1* is the first version of a server specialized on GPU power. It is especially marketed for machine and deep learning. Here, it contains up to 8 *NVIDIA Tesla P100* graphic cards.

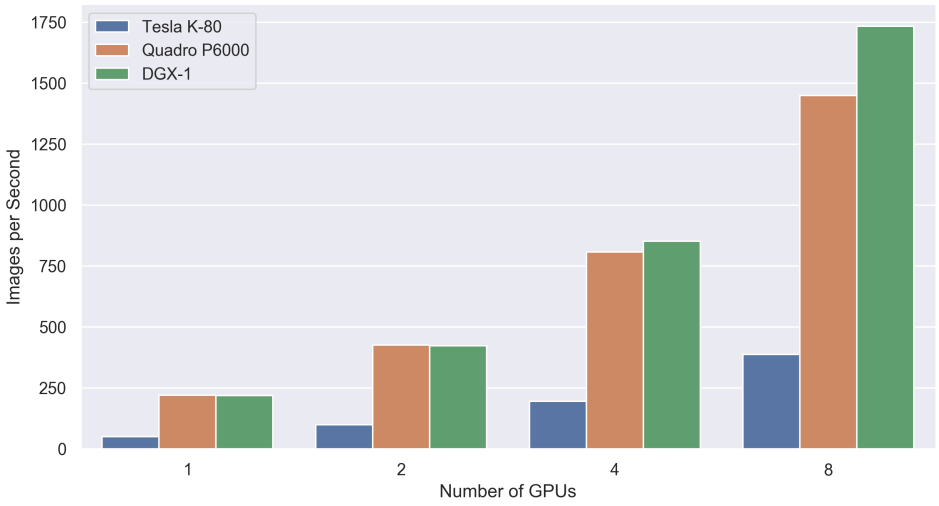

Figure 3: Benchmark result on ResNet-50 with different GPU models

As can be expected, it is obvious that our setup is a lot better than the Tesla K-80. While the NVIDIA DGX-1 is on par with our setup when using 1 or 2 GPUs, it gains a head start with 4 and 8 GPUs. This is probably due to their fine tuning of all the hardware components as well as no overhead due to the renunciation of virtualization and containerization.

2.6 Results

The results are visualized in Figure 2. In addition to the measurement results, the ideal linear performance growth is plotted as a dotted line for each model. It's obvious that there is some overhead for each additional GPU. Even though the gain is not linear, the overall impression is quite satisfactory and more GPUs can clearly provide the means for faster training. Of course that requires the model architecture to support parallel computation.

[4] At least in 2017 when the benchmarks were published.

	ResNet-50			ResNet-152v2		
# of GPUs	Images/Sec	Speedup	Overhead	Images/Sec	Speedup	Overhead
1	220.10	1.00	0.00	95.82	1.00	0.00
2	426.24	1.94	0.03	183.57	1.92	0.04
3	610.40	2.77	0.08	267.25	2.79	0.07
4	808.93	3.68	0.08	342.79	3.58	0.11
5	956.59	4.35	0.13	418.56	4.37	0.13
6	1133.95	5.15	0.14	493.64	5.15	0.14
7	1330.60	6.05	0.14	555.75	5.80	0.17
8	1449.08	6.58	0.18	606.17	6.33	0.21
9	1613.13	7.33	0.19	684.82	7.15	0.21
10	1754.39	7.97	0.20	705.72	7.37	0.26

Table 1: Benchmark results

Deeper insight can be gained from the actual results which are listed in Table 1. In the column *Speedup*, the speedup factor in regard to a single GPU is listed. *Overhead* lists the factor by which the result for this number of GPUs is slower than the ideal speed. The ideal speed is calculated by extrapolating the result of the single GPU benchmark to the corresponding number of GPUs of the respective benchmark without including any overhead cost. Speedup and overhead imply a correlation between the performance gain per GPU and the underlying model. The overhead for the model with fewer layers and less FLOPs, ResNet-50, is smaller than for the larger ResNet-152 model. For example, with 10 GPUs and running the ResNet-50, the setup is just about 8 times as fast as a single GPU, while on the more complex ResNet-152, the speedup factor is just around 7.4. Another observation is that although ResNet-152 has three times the FLOPs than ResNet-50, ResNet-50 is not three times faster. That suggests, that the GPUs where not fully occupied with calculations for the smaller model.

From the computational point of view, the absolute gain that can be achieved with an additional GPUs is still a relevant factor, despite the overhead that comes with it. Even though we are confident that the overhead has its source in the additional GPUs itself, it can not be ruled out that it's a result of other restrictions, e.g. in the distribution of the data to the GPUs, allocated resources from the virtual machine or the docker infrastructure. We conclude that in practice, the number of GPUs will not be the bottleneck of a training process and to improve performance other parts of the hardware or software are more promising starting points.

3 dpa Dataset

In the following section we discuss the dataset provided by dpa and explore some of the metadata in depth. The goal is to provide insights into the challenges given this real world dataset as well as the chances that might arise. The original documents are based on the *NewsML-G2* format for exchanging text, images, video, audio news and event or sports data.[5] It defines certain meta attributes for anything related to news, such as category or genre types. An example of a dpa document can be seen in Table 2.

As of this writing the schema has changed to some extent and we received a second corpus containing roughly 1 000 000 documents called weblines. These new documents are not part of this report.

The types are defined as string, list of string, datetime, integer and categorials. The categorials are strings with the property of having a fixed number of symbols. Those are especially interesting for text classification tasks as they represent groups of documents with the same label. For this work we are especially interested in finding discrepancies in the data, e.g. null/empty/nan values. The *guid (Globally Unique Identifier)* is a natural identifier for all documents. The most promising fields for a

[5] `https://iptc.org/standards/newsml-g2/`

No.	Field	Type	Example
1	guid	str	urn:newsml:dpa.com:20090101:170406-99-973029
2	keyword	list	Gesellschaft, Parteien, Justiz, Wahlen
3	headline	str	plaintext
4	description	str	plaintext
5	contentplain	str	plaintext
6	contenthtml	str	html text
7	slugline	str	CDU/SPD/Other
8	genre	str	dpatextgenre:1
9	subject	category	dpasubject:377
10	medtop	category	medtop:20000018
11	category	category	dpacat:sp
12	geo_area_1	str	DEU
13	geo_area_2	str	empty
14	ednotes	list	html text
15	sent	datetime	2017-06-22 14:08:29
16	version	str	46
17	language	str	de
18	creator	list	['dpa-ad:hellmich']
19	contributor	list	['dpa-ad:heimann_a', 'dpa-ad:siegloff']
20	newspublisher	list	['dpasrv:bay', 'dpasrv:bid', 'dpasrv:bdt']
21	urgency	int	1
22	location	list	['Nürnberg']

Table 2: dpa fields with names, types and examples.

rough text classification purposes are the category and the subject field. The bar plots in Figure 4 and 5 display their respective absolute counts.

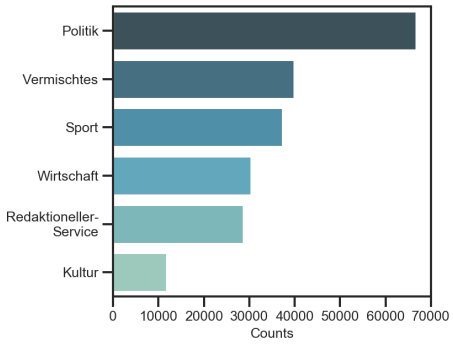

Figure 4: Absolute count of all categories

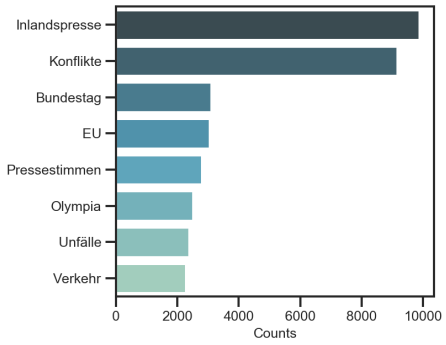

Figure 5: Absolute count of the Top 8 subjects

The fields textitsubject, *medtop* and *category* are categorial values represented by *qcodes*. Qcodes can be resolved with a mapping to their German translations and vice versa. Most prominent categories are *politics*, *sports* and *mixed*. *Politics* is primarily focussed on German relations to other countries and inner politics. As usual the mixed category ranges over all topics that could not be put into any other class. *Sports* is primarily focussed on soccer. The main topic of *economics* is stock exchange information, mostly related to the DAX, the main German stock index. *Editorial services* are dpa internal documents and broadcasts, most of those can be probably dismissed. Finally, *culture* deals with everything related to theater, music and events. Most article narratives evolve around Germany.

The subject field on the other hand is a bit more fine granular with 129 topics in total. Yet still, most of those subjects can be easily mapped to one of the categories mentioned before. In Figure 8 we

can see a word cloud of the *medtop* category which strongly aligns with the categories. Subjects like soccer or stock exchange are next to their respective broader categories.

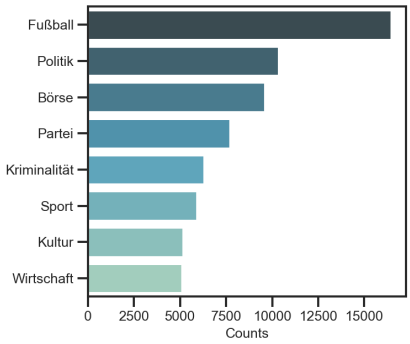

Figure 6: Absolute count of the Top 8 medtops

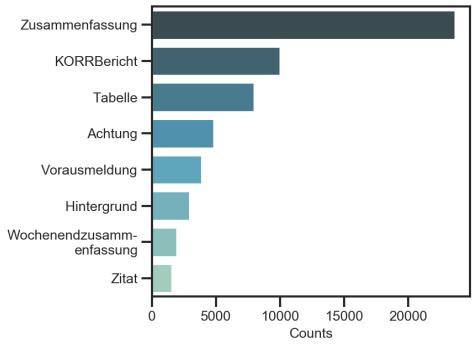

Figure 7: Absolute count of the Top 8 text genres

The text genres in Figure 7 are primarily concerned with summarizations, reports and tables. These information can be especially useful when considering what the correct documents for a certain task are. For summarizations we like texts in the category summarization, news on the stock exchange come usually as table data. Breaking down the data a bit further in Table 3, we get an idea of respective counts and fractional null percentages.

No.	Field	Unique	Not Nulls	Nulls	Perc. not null
1	contributor	35721	213965	760	99.65%
2	creator	827	214391	334	99.84%
3	newspublisher	2449	214725	0	100.00%
4	category	6	214725	0	100.00%
5	genre	30	214725	0	100.00%
6	medtop	230	161721	53004	75.32%
7	subject	129	116611	98114	54.31%
8	urgency	4	214725	0	100.00%
9	version	146	214725	0	100.00%
10	location	8272	140586	74139	65.47%
11	geo_area_1	215	193126	21599	89.94%
13	contenthtml	214491	214725	0	100.00%
14	contentplain	186387	214504	221	99.90%
15	slugline	59247	214725	0	100.00%
16	headline	165107	214725	0	100.00%
17	description	35697	39635	175090	18.46%
18	keyword	15555	146864	67861	68.40%

Table 3: Identified dpa fields with their respective statistics.

The *guid*, *ednotes*, *sent*, *language* and *geo_area_2* fields are dropped due to low variation, missing values or simply because they are non-informative. Low unique values with a low null count indicate categorial variables that can be used for classification tasks. The *description* has the highest null value count, leaving only 18.46% of useful data. This is unfortunate, as this would come very handy for typical sequence-to-sequence modelling tasks like question-answering or document summarization. Other meta information like *medtop*, *subject*, *location* and *keyword* contain a lot of null values. This is also unfortunate since they describe very important information that could be used in specific tasks. If we only subsample fields with 100% of the data intact we would be working on roughly 35 697 documents or in other words, the not-null value count of the *description*.

3.1 Metadata and Content Fields

At the center of the data are content fields. Apart from information about authors, locations and taxonomies or tags, relevant fields include: *medtop*, a hierarchically sorted tagging system with a vocabular, a *headline* that explain briefly what the content is about, a list of *keywords* which can be loosely used to describe the article, the *slugline*, mostly repetitions of keywords, locations and named entities and a *description* that typically goes more into details than the headline. Since large documents are hard to model with language modelling techniques, those fields will be helpful in pointing to the relevant passages of the text. The *medtop* hierarchy broadly encodes tags into taxonomies, starting from very broad categories like politics going down to the presidential elections of 2016 or the G20 protests 2017 in Hamburg. *Medtop* is primarily concerned with the stock exchange, politics and soccer, and overall seems aligned with the categories. Since politics is a very broad topic, criminality, party politics, government and so on can be grouped to it.

Figure 8: Wordcloud of most frequent medtop categories

In our dataset we could observe 229 unique categories with 75.32% of values intact. Since it is hard to display 229 categories for *medtop* in a bar chart we generated a wordcloud using Python in Figure 8. Higher fractional counts of a category are represented by larger words. As can be expected most economical questions are concerned with the stock exchange. We will see later that a lot of named entities like Angela Merkel are currently surpressed here, since topical words are much higher in frequency. The preprocessing involved removing frequent German stopwords.

As can be seen in Figure 9 the headline essentially represents the same notions as medtop. However it contains a lot more details about politics, culture and economics and named entities and locations are much more frequent. Sports fractionally stays the same, while shifting from the granularity of soccer to just Sports.

The *slugline* and *keywords* are very similar to the *headline* and *medtop* categories although they have some slight variations. To illustrate how the different fields coerce we employed topic modelling with *Latent Semantic Analysis (LSA)* by Deerwester et al. (1990) and *Latent Dirichlet Allocation (LDA)* by Blei et al. (2003). LSA is a dimension reduction technique separating a document term matrix into three distinct matrices, factoring out important topics by their most important singular values. On the other hand LDA is a topic modelling technique that maximizes a typical language model via variational inference. For all experiments we used n-topics = 100 and concatenated different fields into one. The result of LDA can be seen in Table 4. Please notice that we concatenated the category to each document, so the algorithm is not fully unsupervised.

Figure 9: Wordcloud of most frequent words in headline

The algorithm could successfully group politics (globally in topic 1 and locally in topic 2), economics in topic 3, sports in 4 and 5. Since LDA does not provide any ranking among the topics we display the first 10 of 100. Each word represents a probability belonging to a certain topic and therefore it is possible that each word is part of many different topics. LSA topic modelling on the other hand is much more simple. Each topic is ranked by their best singular eigenvalue. In Table 5, topic 1 represents the most prominent topic. We can easily see that sports is the most prominent topic, followed by politics and economics. Topic 6 represents culture. Since eigenvalues can be ranked, the topic id is important, since each topic explains the total variance of original data to a higher pecentage.

topic id	word 1	word 2	word 3	word 4	word 5
1	bundesregierung	politik	merkel	chile	international
2	csu	politik	parteien	bayern	cdu
3	aktienkurse	börsen	börse	wirtschaft	mdax
4	fußball	bundesliga	münchen	sport	bayern
5	nationalmannschaft	sport	radsport	em	wm
6	slalom	sport	wiederwahl	podest	klar
7	gesellschaft	politik	soziales	kommunen	familie
8	verteidigung	bundeswehr	pferdesport	springen	politik
9	groko	regierung	politik	innenpolitik	tiere
10	fußball	leipzig	bundesliga	sport	sachsen

Table 4: LDA topics for keyword, headline, slugline, medtop and category fields.

Dealing with sports, again, means that we are heavily overfitting on soccer. A model dealing with sports like that will have difficulty dealing with news about other sports. This may affect for example news from counties where other sports like american football, ice hockey or gymnastics are popular. Economics heavily overfit on stock exchange news instead of dealing with macroeconomics. Politics overfits to partisan politics. Since we fed labels via the category and medtop fields into the algorithms, both were able to assign proper clusters around them. The topic models would look much different if there was no direct label associated with each text.

topic id	word 1	word 2	word 3	word 4	word 5
1	sport	fußball	bundesliga	münchen	politik
2	politik	cdu	spd	parteien	csu
3	wirtschaft	aktienkurse	börse	börsen	investments
4	cdu	csu	spd	politik	parteien
5	sport	fußball	bundesliga	tennis	wm
6	kultur	film	leute	literatur	musik
7	wirtschaft	investments	investition	börse	börsen
8	aktienkurse	investments	investition	kurse	investmentfonds
9	btw	wahlen	wahl	bundestag	regierung
10	bundesliga	fußball	wm	nationalmannschaft	confederations cup

Table 5: LSA topics for keyword, headline, slugline and category fields.

Another way of looking at news articles is by inspecting their named entities. We tested the *Named Entity Recognizer (NER)* of *SpaCy* (Honnibal and Montani, 2017), which is trained with an average perceptron model on the German Wikipedia. The German NER has 4 classes: *PER* for persons, *LOC* for location, *MISC* for miscellaneous and *ORG* for organizations. In Table 6 we can see the result on running on the meta text fields with the highest named entity count for each class. In PER, major political players such as Trump or Merkel are involved, in LOC mostly countries, in MISC stock exchange news and ORG political parties.

	PER		LOC		MISC		ORG	
	Entity	Count	Entitiy	Count	Entity	Count	Entity	Count
1	Trump	1122	Berlin	1466	MDAX	721	SPD	1836
2	Merkel	947	USA	1087	TecDAX	702	EU	1011
3	Schulz	798	Türkei	1076	DAX	694	POLITIK	608
4	Gabriel	583	Russland	1051	Frankfurter	521	AfD	565
5	Seehofer	358	China	828	SDAX	467	CDU	549

Table 6: Top 5 named entities for each of the 4 entity categories.

Since we can only see the Top 5 entities for each label, we analysed the Top 50 of each class to get a feeling about the true negatives and intra classification rate. In the following are some examples from the corresponding Top 50 lists of the different entities, which were clearly associated with the wrong label. Their absolute count is noted in the parentheses.

PER august (338), 07 (182), eurokurs (103), dax (94)

LOC trump (513), live-sport (411), interbanken-kurse (229), märkische oderzeitung (227)

MISC karlsruhe (415), facebook (255), sport (103), google (102)

ORG nordkorea (224), halle (158), ezb-referenzkurs (231), mainz (246)

Some words are merely misclassified as named entities at such as 07 or sport which are obviously not correct. More grave is the intra classification, where named entities are correctly identified as entities but incorrectly labeled: Nordkorea is not an organization, Trump is not a location, Eurokurs is not a person and Facebook should definitely be an organization. Everything put to *MISC* is a valid label, however there are much more suitable labels. Since we searched for such samples in the Top 50 it would be interesting to see how the named entity recognition fares with much lower occurence counts. Since a lot of different applications rely on named entity recognition as a preprocessing step, it is expected to drive down accuracy in following tasks.

3.2 Content

The actual content is probably the most useful and yet unstructured part of the document. Headlines and descriptions are good for quick prototyping and for pointing into the right direction if the content is too large. However natural language processing systems should actually work on language. All language parsing tasks are done with SpaCy. The total corpus as of this writing contains 64 472 262 million tokens. There are 451 564 unique words, which is 0.7% of all words. The average word count per document is 300.5 with a minimum value of 7 and a maximum value of 3853.

In the following we have a selection of different topics per text field per category which can be any of Editorial Service (see Table 7), Politics (see Table 8), Sports (see Table 9), Economics (see Table 10), Art (see Table 11) and Mixed (see Table 12). The method used is LSA like before. None of the topics are handpicked, these are raw results.

topic id	word 1	word 2	word 3	word 4	word 5	word 6
1	meldung	berlin	termine	berichterstattung	finden	themen
2	meldung	berlin	themen	beitrag	liegt	termine
3	berlin	meldung	geplant	rede	spd	themen
4	berichtet	versendet	berlin	korritalk	oton	ansprechpartner
5	spieltag	gruppe	runde	fußball	vorrunde	einzel

Table 7: LSA topics for category Editorial Service (dpacat:rs)

The Editorial Service news items are typically internal messages, updates or breaking news. This is enforced by words like Meldung or Themen. Topics 1 to 4 seem like duplicates with slight changes to each other. They also contain a lot of daily summarizations for the other categories. The fifth topic however deals with news about soccer.

topic id	word 1	word 2	word 3	word 4	word 5	word 6
1	berlin	meldung	spd	cdu	politik	merkel
2	trump	berlin	usa	präsident	spd	donald
3	meldung	spd	prozent	union	merkel	cdu
4	trump	berlin	prozent	meldung	usa	politik
5	berlin	spd	trump	türkei	schulz	euro

Table 8: LSA topics for category Politics (dpacat:pl)

The Politics news items are messages concerning German politics (topic 1), the German-US relations (topic 2) or the European Union, Germany and Turkey (topic 5). Some topics are simply duplicates e.g. topic 3 and 4.

topic id	word 1	word 2	word 3	word 4	word 5	word 6
1	spieltag	fc	meldung	fußball	gruppe	bundesliga
2	meldung	fc	geplant	sv	fußball	sc
3	meldung	spieltag	fc	gruppe	hauptrunde	geplant
4	eurosport	fc	sport	trainer	sky	spiel
5	gruppe	eurosport	fußball	bundesliga	sky	sport

Table 9: LSA topics for category Sports (dpacat:sp)

The Sports news items all concern soccer. As soccer is the German national sport it's leading throughout the Top 5 topics. The different topics are pretty much duplicates of each other and not discernable.

topic id	word 1	word 2	word 3	word 4	word 5	word 6
1	prozent	euro	meldung	berlin	unternehmen	punkte
2	meldung	prozent	berlin	wirtschaft	fotoarchiv	euro
3	de	veränderung	stand	prozent	nl	dividende
4	de	werte	nl	älter	dividende	wert
5	euro	prozent	milliarden	millionen	kg	punkte

Table 10: LSA topics for category Economics (dpacat:wi)

The Economics news items are messages concerning the Euro and high economical values. For some reason the German stock exchange is left unnoticed. They probably occur in other topics.

topic id	word 1	word 2	word 3	word 4	word 5	word 6
1	meldung	kultur	berlin	termine	geburtstag	ausstellung
2	stichwörter	korr	jahreschronik	listicle	jahres	film
3	film	stichwörter	euro	leben	kultur	berlin
4	euro	film	berlin	hamburg	millionen	platz
5	ausstellung	film	kunst	museum	künstler	documenta

Table 11: LSA topics for category Art (dpacat:ku)

The Art news items are messages about culture, films, exhibitions or concerts.

topic id	word 1	word 2	word 3	word 4	word 5	word 6
1	meldung	berlin	termine	panorama	prozess	fotoarchiv
2	prozent	millionen	zuschauer	rtl	meldung	marktanteil
3	foto	archiv	jährige	polizei	prozent	termine
4	berlin	foto	prozess	panorama	archiv	polizei
5	polizei	foto	meldung	berlin	archiv	panorama

Table 12: LSA topics for category Mixed (dpacat:vm)

The Mixed messages have nothing special other than police reports, statistics or dates. Most of the topics should be easily separable based on their content. However it is questionable whether we need to actually work with the Editorial Service and Mixed categories at all. They do not distinguish themselves enough in any kind of way. Their informative value seems rather low for natural language tasks and the target audience. This is obviously different within a news curator, that relies on the Editorial Services.

4 Knowledge Discovery in Databases

The *knowledge discovery process in databases (KDD)* is the major methodology. In machine learning applications, especially with natural language, there is an immense need for preprocessing and filtering. Since most of the dpa data has a variety of textual fields and metadata, it is vital to persist them into a database for structured retrieval. As we would like to perform exploration and classification on the data we need a pipeline that transforms the raw representations into the relevant feature space.

Long raw texts need a lot of power and are often constrained by RAM and CPU. The KDD is a method of efficiently parsing multiple raw inputs into clean and efficient representations that can be easily queried and processed in experiments. As depicted in Figure 10, we decided to model the dependencies with a preprocessing scheme derived from *Extract-Transform-Load (ETL)* pipelines.

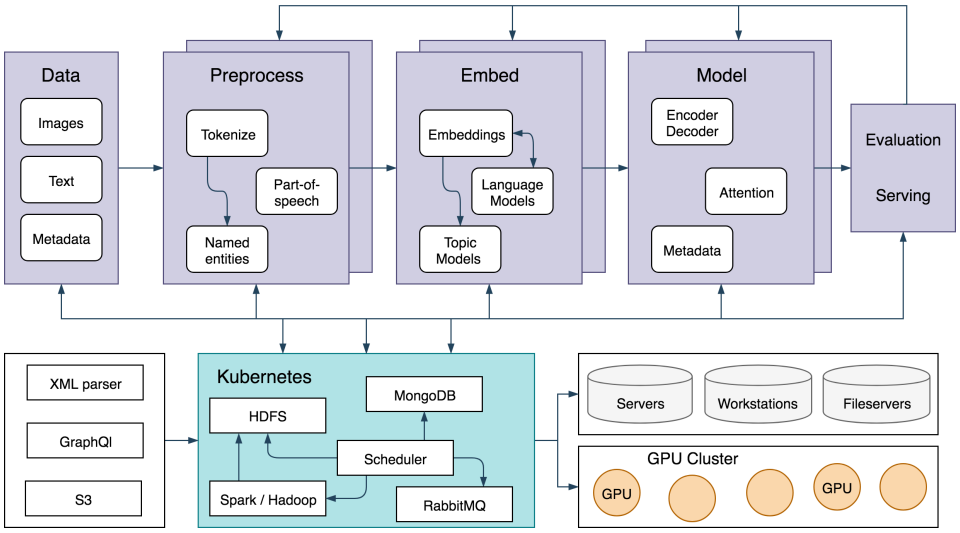

Figure 10: Extract-Transform-Load Graph

The graph has two major parts, namely the infrastructure and the data pipeline. In the following will give an extensive overview of our system and how the various components relate to each other.

4.1 Infrastructure

The infrastructure has three major components, as can be seen in Figure 11, dealing with scraping and retrieving, offering services for persistence and computation via Kubernetes and hardware like GPU clusters or servers as well as storage.

The first module is for scraping and retrieving raw XML in the NewsML-G2 format on S3 as well as parsing JSON from S3, with different data mappings. The mappings can be seen in Section 3 before. The second component is a Kubernetes cluster with deployed services. Vital to this pipeline are HDFS, MongoDB and Hadoop/Spark for file storage and fast RAM access to the data. We store all the parsed data of step 1 into the MongoDB for easy data retrieval and a raw document backup. HDFS is our distributed file server where we will store different file formats ranging from CSV, JSON, Parquet, npy or the h5py format. The idea is to make it accessible in all steps during the pipeline, enabling the caching of intermediate results and computing in parallel.

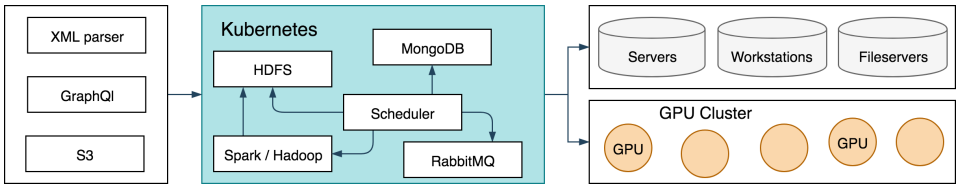

Figure 11: KDD infrastructure

The scheduler will be the component to schedule jobs of the pipeline that in turn store results in our HDFS. It should ideally have the capabilities to model the data flow as a directed acyclic graph. For this to work we will probably use Apache Airflow that makes it easy to model preprocessing as a DAG. Airflow is a platform to programmatically author, schedule and monitor workflows. Workflows are essentially DAGs that can be defined in Python and run via command line and web interfaces using a variety of tools like Spark, MongoDB or HDFS.

4.1.1 Parsing and MongoDB

MongoDB is our primary document storage and easy to setup. Its flexible document data model makes it easy to import, export and update data. It is a great for structured document querying and

pagination. This is ideal for our setup, since the external documents come in a variety of formats and might come from a variety of external sources. Hence it is necessary to provide a modular way of extending our module to each format and data source.

Parsing the NewsML-XML format is necessary to obtain a representation we can work with. Since JSON is a widely adopted format and MongoDB uses the BSON format, we first parse the XML documents to JSON, using a Golang module. The JSON encoded content is then mapped with a mapping function to a flattened and sanitized document and fed into a MongoDB for easy data retrieval. The parsing tool currently has a simple interface. For each data source there is, it creates a data mapper of the following form

$$
\begin{aligned}
retrieve\_files &:: string \rightarrow [file\_obj] \\
parse\_format &:: file\_obj \rightarrow obj \\
map\_object &:: [obj] \rightarrow [mapped\_obj] \\
store &:: mongo\_db\_con \rightarrow mapped\_obj \rightarrow bool
\end{aligned}
$$

After supplying this interface, for instance for XML, we need to implement a file retrieval method that can read the raw data from the external source. This can be S3, a local file system or the HDFS. Then we need to parse the format formally, e.g. XML/JSON/Parquet. When the files can be properly parsed we need to traverse through the document with a mapping function. This sanitizes field names and flattens the hierarchy before providing a handler for storing into the MongoDB. In this step, everything is still considered as important, so no filtering takes place. The parse_format and map_object methods must raise errors early and stop the entire process for review. The system can be enhanced by putting invalid documents on a review queue.

In general we need to find a good error threshold. It has the task to detect when too many errors mean that something is generally broken, while at the same time should not stop the entire system for small hiccups. Currently we just stop the entire import, which can be necessary because it is not certain that the error did not propagate into the other documents as well. It would also be nice to schedule these methods automatically. An example for automation could be to check every n hours if there is a change in the external source and trigger a job for scraping the documents when necessary. At the moment we trigger the jobs ourselves.

4.1.2 Hadoop Distributed File System

Since MongoDB is not sufficient to do fast lookups over the network and scrolling over all data fields does not scale well, we use the *Hadoop Distributed File System (HDFS)* for raw file storage. Since we will at some point introduce Spark as well, HDFS fits best. According to the Apache site, HDFS is "highly fault-tolerant and designed to be deployed on low-cost hardware while at the same time providing high throughput access to application data especially suitable for large data sets" (Hadoop, 2018). As can be seen in Figure 12, HDFS consists of two basic components, the namenode and the datanode.

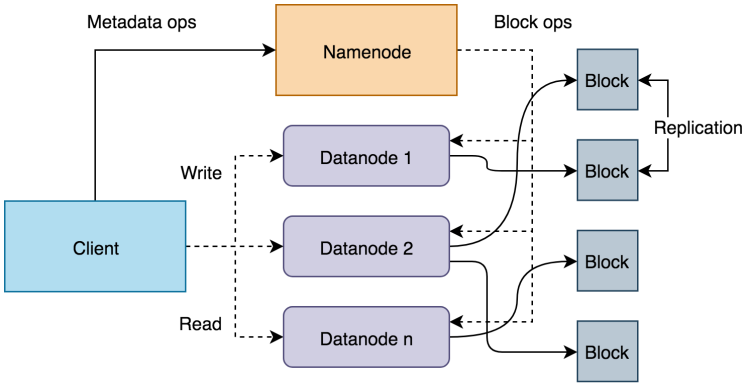

Figure 12: Hadoop Distributed File System

Namenodes store meta data, e.g. where files are stored, how they are stored or their file size. It processes metadata operations such as list or stat. The datanodes actually store data as fixed length blocks on different datanodes with replications. Datanodes and namenodes exchange messages about storage blocks (block ops). When retrieving a stored file, the client lookups the location of the file in the namenode and the namenode sends back a list of datanodes that have the relevant blocks. Clients then proceed to gather blocks from all datanodes directly and combine them to receive the original file. This procedure is the same with read and write actions. Since the data is based on the POSIX system, it can store any kind of file format that runs on it. Files are best for machine learning models, as in theory the CPU processing power becomes the bottleneck. Without any network overhead with suitable caching this will be the direct access for all intermediate preprocessing results and model results (for example stored in Apache Parquet, CSV, NPY, HDF5).

4.1.3 Pandas, Apache Arrow and Apache Parquet

Pandas is the major tool for storing data in-memory with fast serialization capabilities via Apache Arrow to several different file formats like TSV, Apache Parquet and JSON. Pandas provides high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Its major data structure is the dataframe. Dataframes store data columnwise and as a result the access to columns is extremely cheap, but more expensive for rows. We use it primarily for the metadata fields discussed in the previous section.

Apache Arrow on the other hand is a project that tries to standardize columnar and hierarchical data storage for in-memory data. Its major selling point is the table data structure that memory maps large data sets to disk and is able to convert it to multiple different formats such as Pandas dataframes.

Apache Parquet is the columnar and typed equivalent to a CSV file. It supports data types such as String, Integer, Float, Time, Boolean or List and takes away with issues like CSV escapings and delimiters. It is also incredibly fast in conjunction with Apache Arrow and therefore Pandas. Another promising but not yet used library is Dask, a library with parallel computing interfaces to major libraries like Pandas or Numpy. Since multiprocessing is an issue in Python due to the Global Interpreter Lock (GIL), Dask makes it possible to scale Pandas dataframe or Numpy array operations to multiple cores and even clusters. Hadoop/Spark are on the table for replacement, but up until now, Pandas was sufficient for the tasks.

4.2 Data Pipeline

The data pipeline is the logical core. As depicted in Figure 13, it is divided into three steps: preprocessing, embedding and modelling. Each step is governed by different parameters and dependent computations. The first step is to preprocess unstructured text to structured representations with tokenization, part-of-speech tagging and named entity recognition.

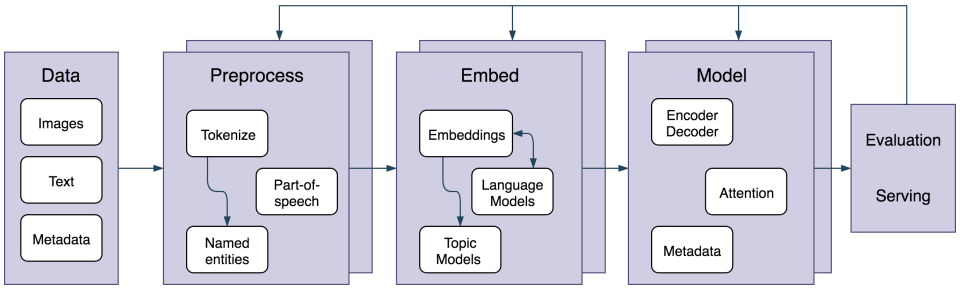

Figure 13: ETL Graph pipeline

Each step maps the computations to a corresponding file to HDFS. HDFS therefore corresponds to a typical cache, persisting computations. Before preprocessing the data, there is however a pre-step that maps each document in the MongoDB to a flattened, easy-to-query collection for later retrieval. It is important to keep the relations between the metadata and field names intact with the guid. Since the dpa dataset contains a lot of bytes or words per document, we have to move the tokenization, *part of speech (POS)* tagging and named entity extraction into distinct long running jobs.

The second major step is to embed the tokens into a latent domain with language models like GloVe (Pennington et al., 2014), Word2Vec (Mikolov et al., 2013), FastText (Bojanowski et al., 2016; Joulin et al., 2016), CoVe (McCann et al., 2017), ULMFiT (Howard and Ruder, 2018) or the Transformer (Radford, 2018). Since the pipeline will at some point also support images this step is enhanced to transform images into their spatial domain.

The third and last step is the actual model that optimizes some loss function with a chain of differentiable functions, calculating the error between gradients and loss. This comes in many different flavors like a *Convolutional Neural Networks (CNN)*, *Long-short-term memory networks (LSTM)* or *attentional encoder/decoders*. While providing a useful model that approximates the given task it is important to evaluate the results against some distance function given some gold standard. We act on the evaluation to improve the overall pipeline, since every step from preprocessing to modelling will directly contribute to the score. On a side note: it should be possible to serve the model via a REST API that can consume a specific query and give an output based on the learned weights.

4.3 Directed Acyclic Graphs (DAG)

A DAG is a datastructure with unidirectional edges between nodes without any cycles. This very basic graph allows us to make use of simple and advanced parallelization schemes. A simple data pipeline could be translated as the DAG presented in Figure 14. Nodes where incoming edges are dashed can be safely parallelized, while straight lines symbolize synchronous operations that depend on running all incoming edge operations.

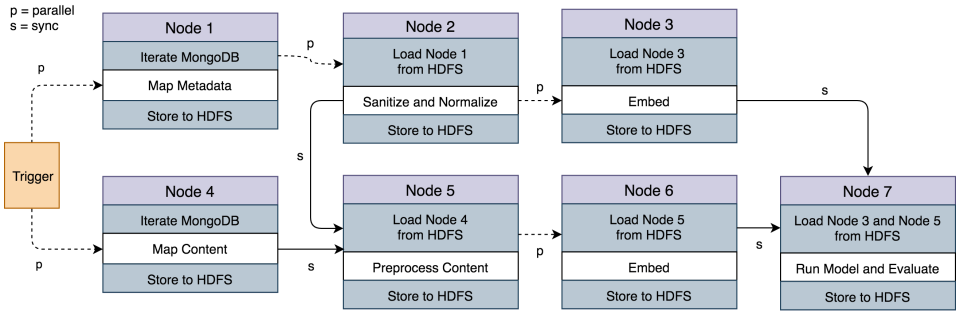

Figure 14: Directed acyclic graph (DAG) with a sample pipeline

Each step begins and ends with loading data from HDFS and storing data to HDFS with various file formats. Each node therefore needs to know what kind of data structures the beforegoing nodes persisted to ingest them. This is a powerful abstraction, especially when working on multiple experiments. Frameworks such as Airflow were made for this particular problem. Define DAGs through various operators that represent nodes and chain them together, providing a global service layer such that each node has access to HDFS and can listen to changes.

The idea behind DAGs is not new, but with the arising complexity of tooling around data processing for instance through the Apache Stack, projects like Airflow represent the single point of truth and configuration for running dependent jobs in parallel on a multitude of different services. We have not yet decided whether to use Airflow or simpler python frameworks like Python Bonobo (an ETL framework). The beauty of it is that everything is optimized by the respective schedulers that arange the DAG and assigns resources such that it runs fastest in distributed environments.

4.3.1 Preprocess

Natural language is unstructured for the computer. Thusly as said before we need to tokenize and pos tag the text. Preprocessing however is a lot more and concerns questions like do we want to incorporate extra knowledge with external knowledge bases? If yes, to what extend? Do we need to represent texts hierarchically e.g. in paragraphs or sentences? Do we need special pos tags like nouns or adjectives dinstinctly? In Figure 15 we can see what different syntactical layers must be captured as well as the governing parameters that restrict them for later use in the models. We use SpaCy for any language parsing that goes beyond simple string spliting.

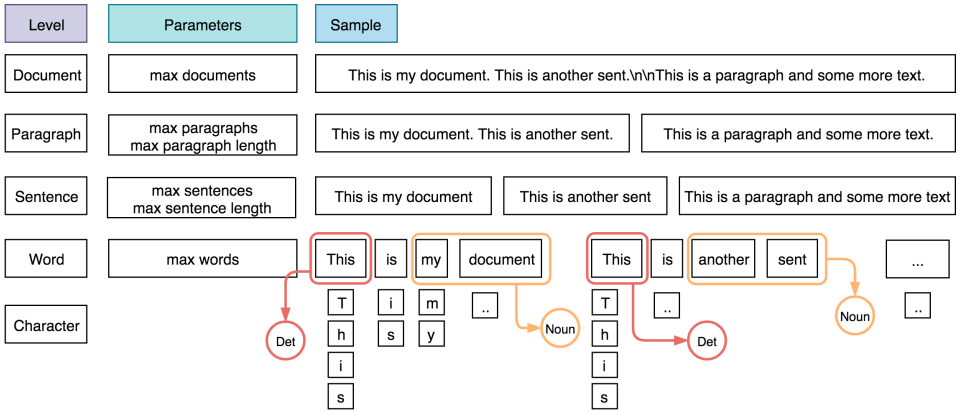

Figure 15: Text hierarchy and parameters

The further down we go the more expensive will the model be as we need to have mechanisms of dealing with different kinds of hierarchies, which often means to add additional layers. The figure only captures the basic levels of encoding but in reality we can subdivide words and characters even more with Byte-pair encodings or based on pos tag groups. The parameters are especially important. Since text representations suffer gravely from the curse of dimensionality, we need parameters that govern how much and what parts of a text will be consumed. That means we have to decide early how our model will consume the input and of what shape that input will be and if it is possible at all to compute the model with the given parameters.

Normally we will need a well rounded dependency parser and pos tagger that chunks a document into lists of sentences and sentences into list of words. This is vital since every additional dimension could mean that our model is able to learn more aspects while at the same time introducing more bias. Further, most of these representations have different morphological preprocessing steps manipulating words and leveraging sentence structures. Some more morphological operations to consider are

- stemming
- lemmatization
- spelling correction
- stopword removal
- only keep noun phrases/verbal phrases etc.
- weight in named entities

Typically these methods generalize words (lemmatization/stemming), drop insignificant words (stopword removal) and parse noun phrases and named entities to attend to the relevant words of a text. Since we also embed the texts further into embeddings, the original morphological feature space will vanish inplace for latent vectors. While efficient, a single run on a subset of documents takes up several hours of processing time, so we need to carefully fine tune or needs and persist results accordingly. With the above defined DAG, this should be no problem.

4.3.2 Embed

Modern day models do not operate on a symbolic level anymore. Each of the above symbolic representations is transformed into a multidimensional tensor space with either some initial probability distribution which is later trained or instantiated with a language model (LM) from another domain (e.g. trained on Wikipedia). These LM are commonly frozen and not updated during training. Thusly we gain a third dimension of preprocessing, the embeddings. These can be categorized on what part of language they are based on. This can start on a very small scale on single characters and can go up to whole documents. Usually, a distinction is made in the following four variants:

- character based (Bojanowski et al., 2016; Joulin et al., 2016)
- word based (Mikolov et al., 2013; Pennington et al., 2014)

- phrase or document based (Le and Mikolov, 2014)
- hierarchical LMs (Radford, 2018; Howard and Ruder, 2018; McCann et al., 2017)

We currently only use character based embeddings based on n-grams and Word2Vec in our experiments, but will add hierarchical LMs as well. The reason for primarily using the FastText vectors is that they are easily accessible on their website.[6] FastText is trained on CommonCrawl and Wikipedia and the learned weights are provided in all available languages. We plan to add several language models in German once the infrastructure is ready to compute them. Since the objective of language models is rather expensive and our corpus is growing this step outpaces our current capacities until the infrastructure is fully functional.

4.3.3 Model and Evaluate

The last component is running a model on GPUs. We will see later how to apply this on a real world classification task. Tensorflow and Keras are the major libraries to create highly scalable deep learning architectures that can make heavy use of the GPUs. Tensorflow can create a computation graph much like the DAGs mentioned before. However since neural networks are just differentiable functions, every step needs a forward and backward computation. The DAG is therefore two folded, first forward propagate and calculate the respective chained functions. Second, the error of the loss function against the labels, is propagated back through the graph updating the respective parameters with their gradients. Tensorflow essentially let us build a DAG through differentiable programming techniques where common functions such as ReLu or the sigmoid have forward and backward implementations. It has three interesting properties: it scales with additional GPUs and is highly parallelized, it automatically differentiates functions into computational graphs and it is heavily optimized to work with efficient numerical methods.

Keras is a high-level framework and can run on top of Tensorflow (it also supports other frameworks like Microsoft's CNTK or Theano). It abstracts away some of the tedious steps to create the respective DAGs. Tensorflow can be time consuming to write and has a very fickle API, while it can take several hundred lines of code to declare your neural network, Keras can get away with much shorter abstract blocks. It is officially vendored into the Tensorflow distribution. A common way of prototyping is to start coding in Keras and use the lower-level Tensorflow for the details if neccessary. As both can be interchangeably used in the same network it brings the right balance between abstraction and configurability.

Every model comes with a scoring or loss function, for instance categorial cross entropy. The loss function can be seen as a function to measure progress, the basic gist being that a lower loss corresponds to a better algorithm. There are several metrics to keep in mind when training neural networks, like the validation/training accuracy and loss. We will see later which ones are useful to consider. The evaluation is crucial since we need to improve the models based on it. It is best practice to test your model against a standard academic dataset to make sure no implementation detail is missing. Every software has bugs and higher complexity likely results in more bugs. Neural networks are increasingly difficult to write and abstract since the layers get more complex. This increasing complexity comes with the price of uncertainty and untestable code fragments.

5 Outlook and Baseline

In the previous sections we have shown benchmark results of the used hardware, presented the dpa dataset in detail and have shown how to create a powerful preprocessing pipeline. Instead of just giving a peek on open questions and research ideas we would like to introduce a very simple baseline for text classification as an example what can be done with the data. In the future we will explore much more complex and powerful deep learning models with the downside of having very high memory and runtime requirements. The goal of this section is not to compare different methods but using one battle proven method.

[6] `https://fasttext.cc/docs/en/crawl-vectors.html`

5.1 Baseline Text Classification

The multinomial Naive Bayes classifier is a very simple and powerful classifier. For an extensive overview, see Eyheramendy et al. (2003). Whenever starting with a classification task the Naive Bayes is a strong baseline yielding surprisingly superb results for its low runtime requirements. It boils down to the following two equations.

$$P(d \mid c) = P(c) \prod_{k=1}^{|d|} P(t_k \mid c) \tag{1}$$

$$\underset{c \in C}{argmax} \left[log\, \hat{P}(c) + \sum_{k=1}^{|d|} log\, \hat{P}(t_k \mid c) \right] \tag{2}$$

Equation 1 can be read as the probability that a document d is in class c is equal to the product of all conditional probabilities that each word t_k of d cooccurs with class c. Since we are dealing with Bayes, the prior probability $P(c)$ is the probability that any document without evidence is in class c. This objective is then estimated with *maximum a posteriori (MAP)*, see Equation 2.

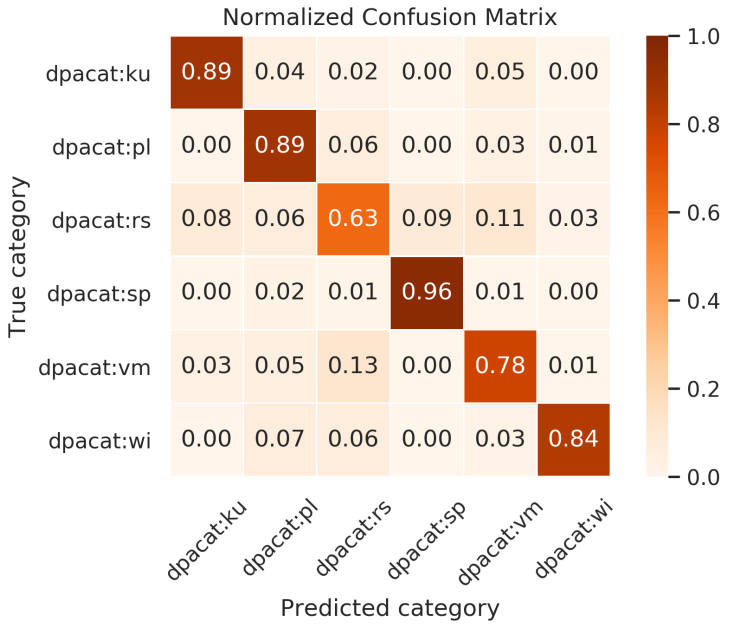

Figure 16: Multinomial Naive Bayes Confusion with dpacat:rs

Since floating point errors of the product of term probabilities of a document can skew the result siginificantly, we instead maximize the sum of log probabilities. That is we select the highest conditional probability of a document d being part of c. We use the sklearn implementation with a few additional optimizations and features. The sklearn implementation uses a laplacian smoothing of $\alpha = 1.0$ that reduces the zero counts to very small values. However since we do not use simple counts in the probability estimation but rather a *term-frequency inverse-document-frequency (tf-idf)* cooccurence matrix we set the term to $\alpha = 1\text{e}{-10}$.

In general the preprocessing we use is simple. We keep the raw text as is, lowercase it and remove special characters with whitespace tokenization. We employ tf-idf to embed words with respect to their relevance and contribution to the corpus and document. We cut off the most frequent and least frequent words, as well as dropping stop words with a static list. We do not employ n-grams since they did not boost the performance. Before presenting the results let us quickly review the dpa categories in Figure 13. In the following we will present two experiments, one with all 6 classes and one where dpacat:rs is removed, keeping 5 classes.

Name	Field Name	Abs. Count	Percentage
Culture	dpacat:ku	11985	5.52
Politics	dpacat:pl	67410	31.05
Editorial Service	dpacat:rs	29067	13.39
Sports	dpacat:sp	37740	17.38
Mixed	dpacat:vm	40245	18.54
Economics	dpacat:wi	30648	14.12
Total		217095	100.00

Table 13: Overview over categories for classification

In the first experiment, we keep all 6 classes resulting in a 83.0% accuracy. In Figure 16 we can see that dpacat:rs has a relative high account of misclassifications. This concerns all categories, but especially dpacat:vm. This leads to the conclusion that it is hard to seperate dpacat:rs from the other categories.

In the second experiment, we drop dpacat:rs and only keep the remaining 5 categories. This results in a 93.2% accuracy. In Figure 17 we can see that it is now dpacat:vm that's misclassified the most and with almost all categories. But leaving out the error of dpacat:rs, the performance improved by almost 10%.

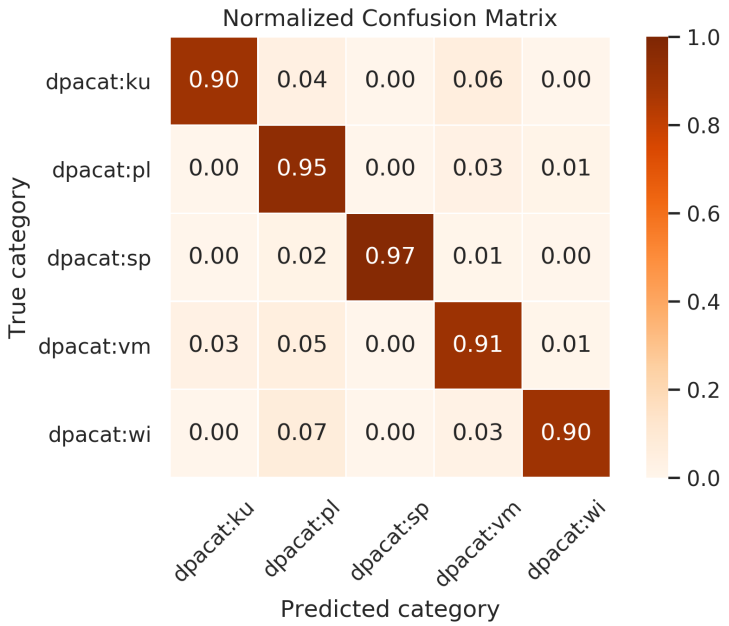

Figure 17: Multinomial Naive Bayes Confusion without dpacat:rs

For a real world dataset this is actually high, given that the model runs in under 1 second with a grid search trying several combinations of the alpha and prior probability initialization. In a follow up paper we will look into the classification problem with much more complex and apt models as well as preprocessing strategies. The current baseline scores are 83% accuracy with dpacat:rs and 93.2% accuracy without.

5.2 Outlook

Several research questions open up after this basic classification experiment. What kind of preprocessing steps are important to this dataset in order to achieve better performance? Recent trends have shown that NLP with deep learning lead to superb results. Is it possible to achieve much better

performance on the classification task? The rise of language models within just the last year with more complex training schemes like ULMFiT (Howard and Ruder, 2018) or ELMO (Peters et al., 2018) and much more involved models like BERT (Devlin et al., 2018) and the Transformer-XL (Dai et al., 2019), sky rocketed perplexity and accuracy scores. Instead of training a fully fledged classification, question answering or summarization model, we instead train a language model that is capable of solving different tasks at once. The general idea can be seen in Figure 18.

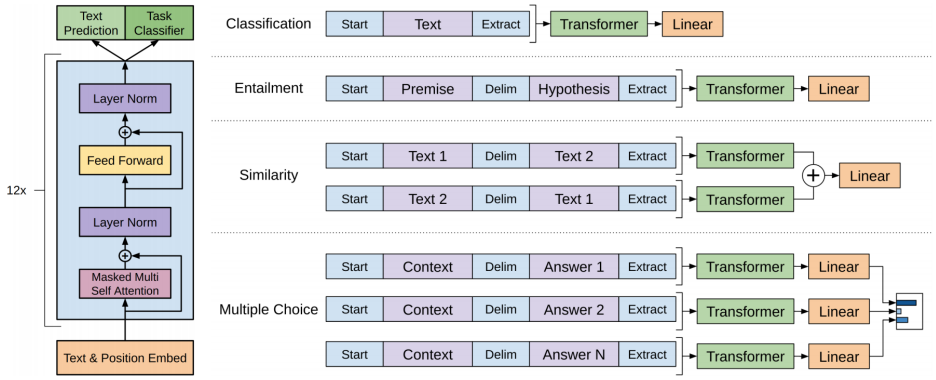

Figure 18: Transformer language model as in Radford (2018)

The difference to many applications before is that instead of rolling a fully fledged model for a specific task, a much simpler model according to the data sources with a highly capable language model is used. This is akin to ImageNet in computer vision, where it is common practice for a few years now to first train a general feature extractor and connect it to a downstream task. In NLP it is common practice, using fixed length context vectors like GloVe (Pennington et al., 2014) or Word2Vec (Mikolov et al., 2013) trained on a large corpus. The models are portable, fixing the problem of polysemy and syonymy.

One of the newest addition to the research community by OpenAI implicate that more power and ressources may lead to much better results. Their large Transformer-based language model named *GPT-2* has 1.5billion parameters and is trained on 40GB of text from 8 million webpages (Radford et al., 2019a,b). The results seem impressive but the costs are enorm. And although this model achieves state of the art results in 7 out of 8, it ironically still underfits the data. This may indicate that learning from an even larger dataset than the used 40GB dataset could still lead to much better results.

These newer language models should in theory overcome some of the caveats of embeddings. First, hierarchical information is learned. Second, they might be capable of overcoming the domain drift, for example that a dataset does not fit the pretrained context vectors. Third, instead of learning a single context vector (one size fits it all) and overfitting to a different problem space, we instead are interested in several layers of generalized feature extraction. A question arises then: Are the new language models much better at dealing with these issues, or do embeddings with complex neural networks for a specific task still have their place?

Acknowledgement

We thank Dr. Gerd Kamp for the great dataset and Professor Kai von Luck and all people at the CSTI for their support and especially for making the hardware available.

References

Blei, D., Ng, A., and Jordan, M. (2003). Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3(4-5):993–1022.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2016). Enriching Word Vectors with Subword Information. *CoRR*, abs/1607.04606.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J. G., Le, Q. V., and Salakhutdinov, R. (2019). Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. *CoRR*, abs/1901.02860.

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by Latent Semantic Analysis. In *Journal of the American Society for Information Science*, volume 41, pages 391–407.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR*, abs/1810.04805.

Eyheramendy, S., Lewis, D. D., and Madigan, D. (2003). On the Naive Bayes Model for Text Categorization.

Hadoop (2018). HDFS Architecture - Assumptions and Goals. `http://itm-vm.shidler.hawaii.edu/HDFS/ArchDocAssumptions+Goals.html`. Accessed: 2018-12-15.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385.

Honnibal, M. and Montani, I. (2017). spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. *To appear*.

Howard, J. and Ruder, S. (2018). Fine-tuned Language Models for Text Classification. *CoRR*, abs/1801.06146.

Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. (2016). Bag of Tricks for Efficient Text Classification. *CoRR*, abs/1607.01759.

Le, Q. V. and Mikolov, T. (2014). Distributed Representations of Sentences and Documents. *CoRR*, abs/1405.4053.

McCann, B., Bradbury, J., Xiong, C., and Socher, R. (2017). Learned in Translation: Contextualized Word Vectors. *CoRR*, abs/1708.00107.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. *CoRR*, abs/1310.4546.

Pennington, J., Socher, R., and Manning, C. D. (2014). GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. *CoRR*, abs/1802.05365.

Radford, A. (2018). Improving Language Understanding by Generative Pre-Training.

Radford, A., Wu, J., Amodei, D., Amodei, D., Clark, J., Brundage, M., and Sutskever, I. (2019a). Better Language Models and Their Implications. `https://blog.openai.com/better-language-models/`. Accessed: 2019-02-17.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019b). Language Models are Unsupervised Multitask Learners.

TensorFlow (2018). Benchmarks. `https://www.tensorflow.org/guide/performance/benchmarks`. Accessed: 2018-12-17.