DeepTC – An Extension of DKPro Text Classification for Fostering Reproducibility of Deep Learning Experiments

Tobias Horsmann and Torsten Zesch
Language Technology Lab
Department of Computer Science and Applied Cognitive Science
University of Duisburg-Essen, Germany
{tobias.horsmann,torsten.zesch}@uni-due.de

Abstract

We present a deep learning extension for the multi-purpose text classification framework *DKPro Text Classification (DKPro TC)*. DKPro TC is a flexible framework for creating easily shareable and reproducible end-to-end NLP experiments involving machine learning. We provide an overview of the current state of DKPro TC, which does not allow integration of deep learning, and discuss the necessary conceptual extensions. These extensions are based on an analysis of common deep learning setups found in the literature to support all common text classification setups, i.e. single outcome, multi outcome, and sequence classification problems. Additionally to providing an end-to-end shareable environment for deep learning experiments, we provide convenience features that take care of repetitive steps, such as pre-processing, data vectorization and pruning of embeddings. By moving a large part of this boilerplate code into DKPro TC, the actual deep learning framework code improves in readability and lowers the amount of redundant source code considerably. As proof-of-concept, we integrate Keras, DyNet, and DeepLearning4J.

Keywords: Reproducibility, Deep Learning, DKPro TC, Keras, Dynet, DeepLearning4J

1. Motivation

Experiments based on deep neural networks pose huge challenges to reproducibility. An experiment consists not just of the actual neural network architecture, but also of a potentially large number of processing steps to prepare the data. Furthermore, countless network parameters exist, which can greatly affect a network's performance. Reproduction attempts, thus, lead to a high amount of time spent with constructing comparable processing setups. Even if the deep learning code is released, code that applies all preprocessing steps is often missing. Additional effort is often necessary to install and configure required third-party tools.

A potential solution to these reproducibility challenges is DKPro Text Classification (DKPro TC)[1] (Daxenberger et al., 2014). DKPro TC ensures that the same preprocessing is automatically applied to any (new) dataset and provides convenience services such as an automatic installation of third-party tools. DKPro TC experiments are end-to-end shareable, enabling a quick and easy execution of experiments by other researchers. However, until now, DKPro TC only supports shallow learning frameworks. In this work, we present a deep learning extension to DKPro TC called DeepTC. In addition to improved reproducibility, DeepTC also eases architecture analysis by moving boilerplate code for pruning word embeddings and vectorization into DeepTC. This leads to a considerably reduced amount of framework-specific deep learning code. As proof-of-concept, we integrate the deep learning frameworks Keras[2], Dynet (Neubig et al., 2017) and DeepLearning4J[3].

2. DKPro Text Classification (DKPro TC)

We start with an overview of the current state of DKPro TC and discuss how it helps NLP researchers with their daily work. DKPro TC is a Java-based open-source software framework build upon the UIMA architecture (Ferrucci and Lally, 2004) and the lightweight DKPro Lab framework (Eckart de Castilho and Gurevych, 2011) for parameter sweeping experiments. DKPro TC provides an intermediate software layer that harmonizes the use of various machine learning frameworks. The same experimental setup is easily executed with one or more classifiers, which enables a direct comparison of different implementations. The user defines feature extractors, which collect the information the classifier uses for training a model. DKPro TC transforms the extracted feature information into the data format required by the respective classifier. Hence, the user is completely shielded from the intrinsic data format details required by a certain implementation. Furthermore, DKPro TC allows running experiments as train/test or cross-validation setups and takes care of all data splitting operations, execution, and aggregation of results. Required pre-processing components are automatically downloaded and installed. In summary, DKPro TC allows sharing self-contained and executable experiments with other researchers.

As of version 0.9.0, DKPro TC supports: Weka (Hall et al., 2009), LibLinear (Fan et al., 2008), LibSvm (Chang and Lin, 2011), SvmHmm (Joachims, 2008), and CrfSuite (Okazaki, 2007) that cover the common machine learning tasks in NLP, i.e. single outcome, multi-outcome and sequence classification.

2.1. Design Goals

DKPro TC is designed around three design goals: (i) reproducibility, (ii) convenience, and (iii) applicability.

Reproducibility is achieved by using only software components that are released in public repositories such as Maven Central. This ensures that software remains available even if components are no longer maintained. Furthermore, all parametrization details of the experiment, e.g.

[1]https://github.com/dkpro/dkpro-tc.git
[2]https://keras.io
[3]https://deeplearning4j.org

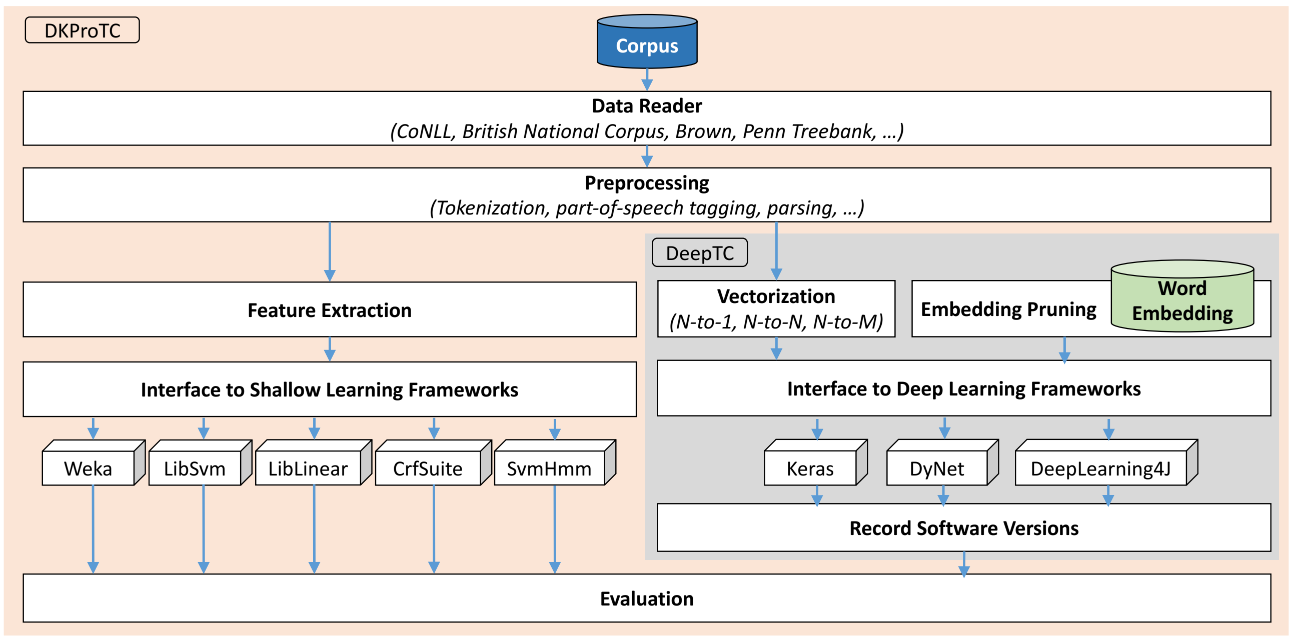


Figure 1: Processing Schema for Experiments in DKPro TC

classifier parametrization, features, and configuration of pre-processing tools, are automatically stored in a DKPro TC project for sharing the project right away.

Convenience is achieved by (i) easy-to-implement feature extractors with frequently needed ones being already pre-defined, and (ii) automatic installation of third-party components from public repositories. Additionally, DKPro TC integrates DKPro Core (Eckart de Castilho and Gurevych, 2014) and thus provides a rich source of tools such as tokenizers, part-of-speech taggers, or lemmatizers, which can be added in a plug-and-play fashion as processing component. These tools are automatically downloaded and installed as Maven artifacts. This provides a high degree of flexibility in terms of experimenting with various pre-processing tools and picking the best working one for a certain task. Of course, researchers can always implement their own UIMA processing components.

Applicability DKPro TC supports all common machine learning setups related to text classification tasks, i.e. single outcome (e.g. sentiment analysis), multi outcome or sequence classification (e.g. part-of-speech tagging), and regression (e.g. assessment of text reading difficulty).

2.2. Shallow Architecture

Figure 1 shows a conceptual overview of DKPro TC.

Reader The corpus data is read into DKPro TC by a reader component. Via DKPro Core dozens of common NLP formats are supported, for instance CoNLL, TEI, or Penn Treebank (Marcus et al., 1993).

Preprocessing In this step, an optional pre-processing can be applied, which might entail tasks such as tokenization or part-of-speech tagging.

Feature Extraction The feature extractors are applied to the data with access to information created during the pre-processing step. The extracted information is temporarily stored in an intermediate data format.

Interface to Shallow Learning Frameworks The feature information is transformed into the data format of the selected machine learning framework.

Evaluation If test data is provided, the trained model is applied to this dataset (after running through the same pre-processing and feature extraction as the train data). Many commonly used metrics such as accuracy, F-Score or Pearson correlation can be computed during evaluation. In case of cross-validation, aggregated results over all folds are automatically provided.

3. DeepTC – A Deep Learning Extension

Software focusing on the *shallow learning* paradigm is not easily extendible to support the *deep learning* paradigm. The conceptual differences between both paradigms make such an extension challenging, i.e. the shallow paradigm learns a model from a representation created from human defined features while the deep paradigm learns a suited representation by itself. Furthermore, a meaningful extension must not just work on a technical level, but also sustain the advantages of taking workload from the user. Consequently, we conducted an analysis of common deep learning setups in the literature to learn about the challenges to reproducibility and convenience. This led to the *DeepTC* extension shown in Figure 1.

3.1. Format

Many deep learning code releases assume a flat file format to demonstrate the usage of a new network architecture. The most common format is a whitespace or tabulator separation of text and labels. This format is quite popular and wide-spread as it allows a rather easy transformation of the textual data into an integer representation. Thus, one

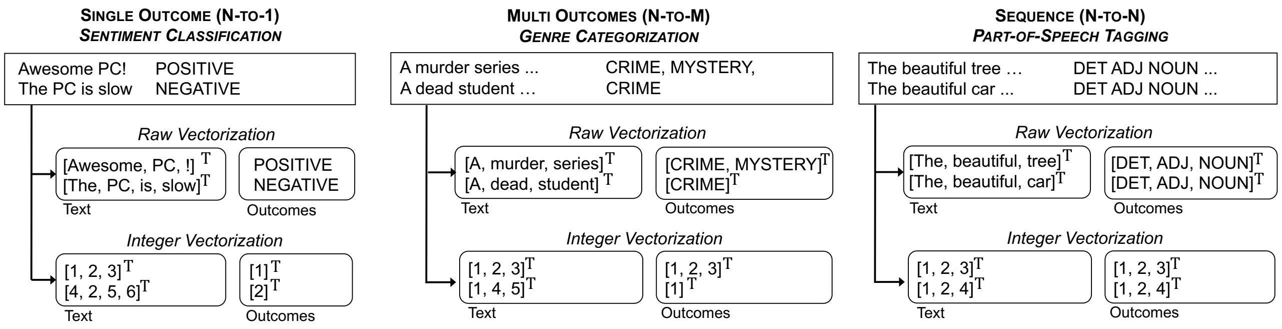


Figure 2: Vectorization N-to-1, N-to-M and N-to-N

must first transform his data into this flat file format before the code can be executed. In case of more complex data formats, such as XML, this leads to considerable additional effort. This challenge is solved in DKPro TC by the many data format readers included in DKPro Core. The seamless integration of DKPro Core allows it to read a large variety of data formats and access information such as lemmas, part-of-speech tags, etc. from DKPro TC. This enables a quick and easy exchange of corpora and data formats. Of course, own readers for highly specific data formats can be easily written, too.

3.2. Vectorization

All textual information has to be transformed into a numerical vector representation before it can be provided to the deep learning framework. This vectorization entails mapping words and labels to integer values. When applying a prototype to unlabelled plain text, the integer values have to be mapped back to their original label to obtain human interpretable results. This is a mandatory task that can be easily automatized. While the general task of vectorization appears straightforward, its details depend on the kind of classification task of which we distinguish the three variants shown in Figure 2:

Single Outcome (N-to-1): In this setting, a single outcome has to be predicted for a text document with N tokens. In classification the outcomes are labels, in case of regression they are numeric values. Use cases for single outcome classification are e.g. sentiment analysis or scoring the reading difficulty of a text (regression).

Multi-Outcome (N-to-M): For a text document with N tokens, M outcomes have to be predicted. For instance, categorization of books into genres, where a single book might have more than just one genre.

Sequence (N-to-N): For a text document with N tokens, an equal amount of N labels has to be predicted. The sequence in which the tokens occur is furthermore informative for predicting the labels. A prominent example is part-of-speech tagging.

Implementation The user is given control as to whether a vector is created with textual information (raw vectorization) or if the words have already been mapped to an integer representation (integer vectorization). Integer vectorization fits most setups and leads to further reduction of user-specific preprocessing code as the mapping process is done automatically by DeepTC. If the network architecture also considers sub-word information, e.g. character- or byte-level information, integer vectorization would be premature as the networks requires access to the actual word forms. For such cases, raw vectorization allows providing the actual words to the deep learning framework. As trade-off, the deep learning code has then to take care of mapping the raw data to an integer representation. This allows DeepTC to be flexible for more complex tasks, but still provide convenience features for common NLP setups.

3.3. Word Embeddings

It is common to use pre-trained word embeddings, which are often quite large with negative effects on the start-up time of experiments. As a consequence, embeddings are usually pruned to contain only words that occur in the vocabulary. Furthermore, in some tasks, words without pre-trained embedding are either dropped or vectors are randomly initialized instead.

Implementation We provide a processing step in which the word embedding is pruned to contain only the occurring vocabulary. The user is given control as to whether words missing in the embeddings are removed or shall be initialized with a random vector. In case no word embedding is provided, this step performs no operation.

3.4. Interface to Deep Learning Framework

The prepared data is provided to the deep learning framework. All necessary files are written to disk and the framework code is executed. The file locations are passed as parameters to the framework code. The framework code is expected to create a file at a specified location which contains the results of the execution.

Implementation An integration of third-party frameworks often leads to challenges how to interface with these frameworks. There might be breaks between programming environments, for instance operating Python frameworks from Java, but also breaks between the data representation in DeepTC and the data format that is expected by a framework. The break between programming environments, i.e. Java to Python to Java, are tackled by defining a protocol of data exchange. For each of the three defined classification tasks, i.e. single outcome, multi-outcome, and sequence classification, a data format is expected in which the framework code provides the predicted outcomes. This allows bridging to deep learning frameworks based on non-Java technologies. The break between data formats is solved by the vectorization processing step which writes the data to disc. As non-Java frameworks work internally with their own data structures, the framework code then can read this data and wrap the vectorized data into the respective data format. This leads to a minimal amount of data conversion overhead that has to take place in the framework code. In case of Keras, for instance, which is based on Python, the vectorized data has to be transformed into the NumPy data type. This defines a simple way of interfacing between different data formats and deep learning environments.

3.5. Software Versions

An important challenge to reproducibility is keeping track of the software versions that are being used for running an experiment. Many deep learning frameworks are still under rapid development and, thus, change quickly with bugs being fixed and APIs being updated. If code is released, it is often not reported which software version was used.

For instance for Keras, which depends on a backend such as TensorFlow, we record not just the Keras version but also the version of the backend and the NumPy library as primary data structure. The software version that is recorded is highly dependent on the respective deep learning framework. This provides a basic software versioning record, which can be released with the experimental code.

4. Limitations

The rapid development of deep learning software creates practical limitations to reproducibility and convenience.

The convenience of automatically installing needed components is easily provided for Java/Maven-based software. For non-Java frameworks, this is not as easily possible and the task of installing software is delegated to the user. We would require a method to serialize the deep learning framework environment into a container that would allow deployment on a third-party computer, i.e. in the case of Keras, which would also entail the respective backend and their dependencies. A common strategy is to use virtualization software such as Docker (Merkel, 2014). A virtualization container is created that capsules a software environment, for instance Python with Keras installed, into a deployable container. At the moment, an automatically creation of such a virtualization container for an experiment is not supported by DeepTC, but one can run DeepTC within such a container if one is prepared beforehand.

A further challenge is to track the names and versions of all involved components the user has to install to reproduce an environment. While using a virtualization container as a black-box environment is convenient, it is often not clear which exact version of the required software is used in a certain setup. Furthermore, recording an entire system configuration setup would lead to an extremely long list of software components with some being more important than others to the reproducibility of results. As a compromise, we record the software versions of the key components, for instance Keras or NumPy, to create an overview of the used software versions to run an experiment.

A further limitation occurs if researchers work on unstable software versions. It is not uncommon that researchers compile their deep learning software from the latest version in a source-code repository to make certain features available, i.e. a bleeding edge version. One would have to record the exact hash-id of the source-code repository from which the software was built to enable reproducibility. Detecting such setups is beyond an automatic detection by DeepTC.

5. Proof of Concept

As proof of concept, we conduct a replication experiment with all three deep learning frameworks in parallel, and compare the results to using a shallow learning framework.[4] We attempt to reproduce the state-of-the-art results for Part-of-Speech (PoS) tagging. Our code is publicly available[5] and demonstrates the usage of DeepTC for deep learning experiments.

DeepTC configuration Figure 3 shows the code snippet for configuring DeepTC for our replication study. Configurations for other (deep learning) classification tasks follow the same structure.

The first lines define the UIMA data reader: one for the training data and one for the test data. As a suitable reader for the WSJ data is already provided by DKPro Core, we can simply us it as-is in our DeepTC experiment without any additional effort. After the readers, we define the so called *parameter space* which configures the experiment. Feature mode and learning mode define the nature of the learning task. In the case of PoS tagging, this is a sequence classification task where a single label is to be predicted for each word.[6] Python installation, pre-trained word embeddings, and user code point to locations in the user's file system. The Python path points to the installed Python version that shall be used to execute the framework code, i.e. the one for which the deep learning framework is installed. In the case of DL4J, which is Java, this parameter is not necessary. Word embeddings points to the file location of the pre-trained word embeddings, this parameter is optional and can be omitted if no pre-trained embeddings are necessary for an experiment. The variable `userCode` points to a file which contains the framework code. The seed value is passed through to the user code to initialize the random generator in the respective framework with the provided value[7]. Integer vectorization is set to `true` to enable the automatically mapping of words to integer values.

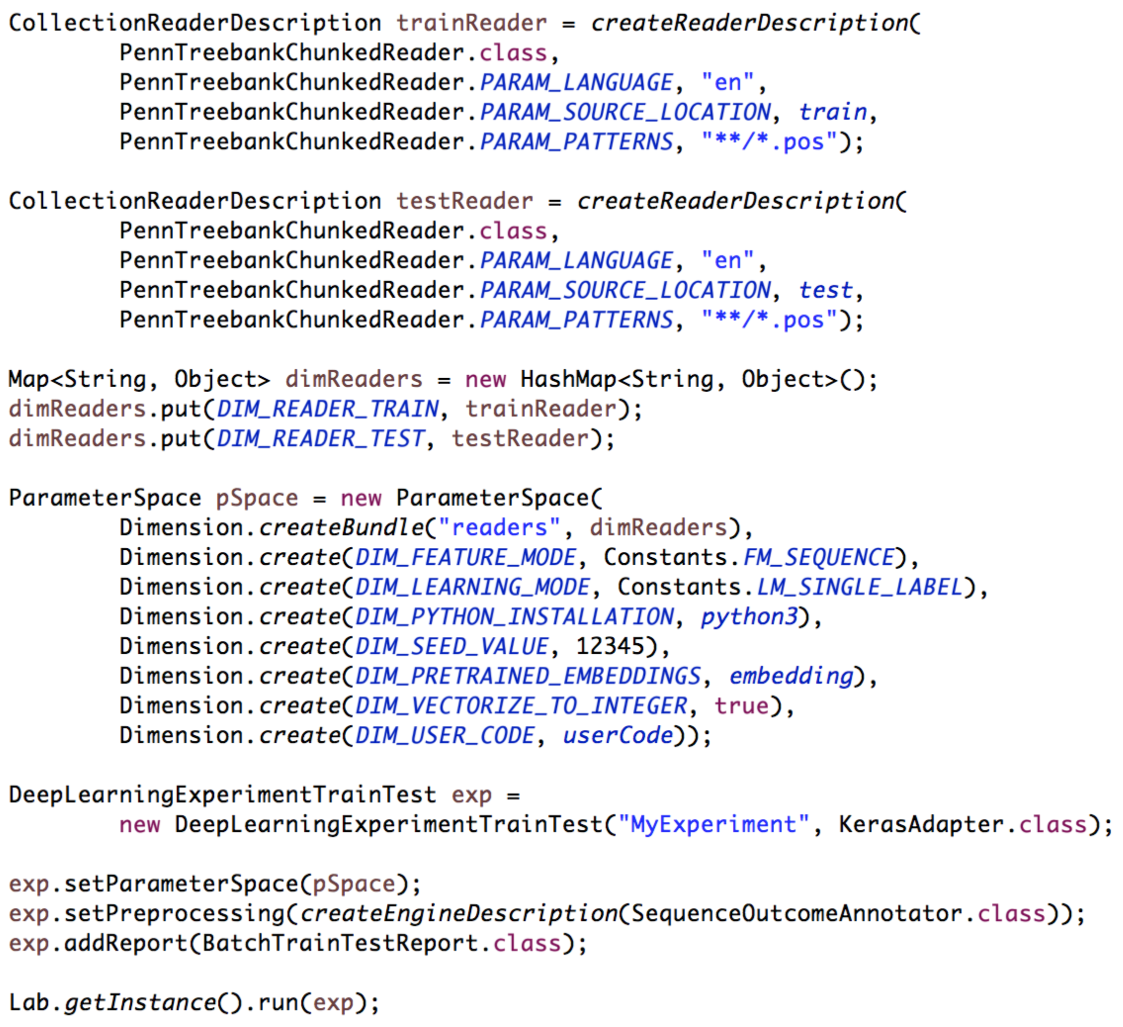
This parameter space is provided to a train-test experiment object (an alternative would be cross-validation), which is then executed by the underlying DKPro Lab environment. Changing the path of the user code and the machine learning adapter allows switching between the deep learning frameworks.

[4]All used frameworks are part of DKPro TC, which makes it easy to implement such comparison between multiple classifiers

[5]`https://github.com/Horsmann/LREC2018-DeepTC`

[6]Other configurations would allow, for instance, a multi-label classification on full documents rather than classifying single words in sentences. See Figure 2 which defines the different vectorization modes for different classification tasks.

[7]This ensures reproducibility by using a fixed seed value for initialization, which leads to the same random numbers being generated between several executions of the framework code



```
CollectionReaderDescription trainReader = createReaderDescription(
        PennTreebankChunkedReader.class,
        PennTreebankChunkedReader.PARAM_LANGUAGE, "en",
        PennTreebankChunkedReader.PARAM_SOURCE_LOCATION, train,
        PennTreebankChunkedReader.PARAM_PATTERNS, "**/*.pos");

CollectionReaderDescription testReader = createReaderDescription(
        PennTreebankChunkedReader.class,
        PennTreebankChunkedReader.PARAM_LANGUAGE, "en",
        PennTreebankChunkedReader.PARAM_SOURCE_LOCATION, test,
        PennTreebankChunkedReader.PARAM_PATTERNS, "**/*.pos");

Map<String, Object> dimReaders = new HashMap<String, Object>();
dimReaders.put(DIM_READER_TRAIN, trainReader);
dimReaders.put(DIM_READER_TEST, testReader);

ParameterSpace pSpace = new ParameterSpace(
        Dimension.createBundle("readers", dimReaders),
        Dimension.create(DIM_FEATURE_MODE, Constants.FM_SEQUENCE),
        Dimension.create(DIM_LEARNING_MODE, Constants.LM_SINGLE_LABEL),
        Dimension.create(DIM_PYTHON_INSTALLATION, python3),
        Dimension.create(DIM_SEED_VALUE, 12345),
        Dimension.create(DIM_PRETRAINED_EMBEDDINGS, embedding),
        Dimension.create(DIM_VECTORIZE_TO_INTEGER, true),
        Dimension.create(DIM_USER_CODE, userCode));

DeepLearningExperimentTrainTest exp =
        new DeepLearningExperimentTrainTest("MyExperiment", KerasAdapter.class);

exp.setParameterSpace(pSpace);
exp.setPreprocessing(createEngineDescription(SequenceOutcomeAnnotator.class));
exp.addReport(BatchTrainTestReport.class);

Lab.getInstance().run(exp);
```

Figure 3: Configuration of a DeepTC experiment

Experimental setup As training data, we use Wall-Street-Journal (WSJ) (Marcus et al., 1993) corpus sections 0-18 and test on sections 22-24, which is the usual evaluation data split of this corpus (Collins, 2002). For each deep learning framework, we implement a plain, bidirectional Long-Short-Term-Memory (LSTM) network (Hochreiter and Schmidhuber, 1997; Graves et al., 2005). As the purpose of the experiment is to show that we can execute arbitrary network code within DeepTC and not to obtain the highest possible results, we use 'default' parameter choices (as much as there is already something like a default in the field). Our bi-LSTM uses 100 hidden units, the output layer applies a softmax function, and we use cross-entropy as loss function during model training. We use the 64-dimensional Wikipedia word embeddings by Al-Rfou et al. (2013). We train 20 epochs with a learning rate of 0.1 using statistical gradient decent.

We compare the deep learning results to the results of a shallow learning PoS tagger that we also implement with DKPro TC. We use Conditional Random Field (CRF) (Lafferty et al., 2001) for implementing this shallow classifier, which is based on the implementation by Okazaki (2007) that is provided in DKPro TC. We use a minimalistic feature set of a tri-gram word window as local word context and provide the cluster-id of a the word in focus if it is contained in a Brown (Brown et al., 1992) word cluster, which was created from 100 million tokens of Twitter messages. The information obtained by Brown clustering is comparable to the information contained in the word embeddings that are used in the neural networks, i.e. both encode distributional knowledge. It is common to also use character ngrams, which we excluded in this case to sustain comparability to the neural network setup, which only use word-level information.

Results Table 1 shows that the different classifiers reach comparable results. Keras and DyNet reach the same result, which is not surprising as they both use Python and the NumPy library. The Java based Deeplearning4J gives slightly lower results. As we are using exactly the same setup and configuration, this is already a finding which could not have been easily achieved without DeepTC. Furthermore, the neural network results are competitive to the 96.5% by Brants (2000) and the 97.6% by Choi (2016).

DeepTC allowed us to avoid a large part of the repetitive work, and limited the manual effort for writing framework-specific code to defining the network architecture and few data-type wrapping method calls. Furthermore, the created experiments are immediately shareable with other re-

	Framework	Acc (%)
DeepTC	Deeplearning4j	95.8
	DyNet	96.4
	Keras	96.4
ShallowTC	CRF	94.2

Table 1: Accuracy on WSJ sections 22-24 using shallow and deep learning classifiers in DKPro TC

searchers to allow a quick and easy replication of our experiments.

6. Related Work

There are several software projects that aim at providing (shallow) machine learning tools over a common interface, e.g. ClearTK (Ogren et al., 2008), NLTK (Bird et al., 2009), Mallet (McCallum, 2002), Scikit-learn (Pedregosa et al., 2011), or Weka (Hall et al., 2009). These projects provide building blocks for creating text classification experiments, but still require a considerable amount of programming by the user. Most similar to DKPro TC are ClearTK and Weka. ClearTK is also UIMA-based and provides a similar middle-layer for defining feature extractors and shares many machine learning tools with DKPro TC. Weka provides many classifiers that work with a Weka-specific data format. An abstraction layer that extracts certain feature values from a dataset is not provided, and the user is responsible for compiling a file in the Weka data format. None of these projects intends to provide a self-contained environment.

There are many deep learning frameworks such as Tensorflow, Theano, DyNet, DeepLearning4J, Torch (Collobert et al., 2002), or Chainer (Tokui et al., 2015) to name just a few. Software such as Keras, Lasagne (Dieleman et al., 2015) or Fuel&Blocks (van Merriënboer et al., 2015) provide a simplified, building-block like interface to an underlying, low-level deep learning framework such as Theano. Data loading capabilities are included to some extent for instance for the well-known MNIST (Lecun et al., 1998) dataset with hand-written digits for image processing tasks. Furthermore, there are approaches to analyze what a neural network actually learns when applied to image and text processing tasks (Yosinski et al., 2015; Li et al., 2016). The deep learning software, thus, provides means to build prototypes quickly, but provides no means to ensure replicability by a third-party researcher. This means that all processing components must be manually provided and configured by the researcher who wants to run a certain prototype.

Hence, the DeepTC extension fills a gap in the software landscape, which will improve reproducibility of deep learning experiments.

7. Conclusion

We presented DeepTC, a deep learning extension of the NLP experiment framework DKPro TC. We discussed the current state of DKPro TC, which is limited to shallow learning frameworks and discussed the need for a software environment that also supports reproducibility for deep learning experiments. As frequent challenges to reproduction of deep learning experiments, we identified incomplete data preparation steps, embedding preparations tasks, and the vectorization of data into an integer representation. DeepTC takes care of those steps and allows to share a self-contained experiment to improve reproducibility. DKPro TC installs necessary pre-processing tools automatically and applies all processing steps to any dataset. Furthermore, by performing the data preparation inside DKPro TC, the high code duplication of typical deep learning code is avoided, which leads to a higher code readability of the actual network code. As proof of concept, we implemented support for three deep learning frameworks: Keras, DyNet, and DeepLearning4J. In a replication experiment, we showed that this setup allows to replicate state-of-the-art result and demonstrated the usage of DeepTC.

Acknowledgement

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under grant No. GRK 2167, Research Training Group “User-Centred Social Media”.

8. Bibliographical References

Al-Rfou, R., Perozzi, B., and Skiena, S. (2013). Polyglot: Distributed Word Representations for Multilingual NLP. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 183–192, Sofia, Bulgaria. ACL.

Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python*. O’Reilly Media Inc.

Brants, T. (2000). TnT: A Statistical Part-of-speech Tagger. In *Proceedings of the Conference on Applied Natural Language Processing*, pages 224–231, Seattle, Washington. Association for Computational Linguistics.

Brown, P. F., DeSouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. (1992). Class-Based n-gram Models of Natural Language. *Computational Linguistics*, 18:467–479.

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A Library for Support Vector Machines. 2(3):1–27.

Choi, J. D. (2016). Dynamic Feature Induction: The Last Gist to the State-of-the-Art. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 271–281, San Diego, California. Association for Computational Linguistics.

Collins, M. (2002). Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1–8, Philadelphia, USA. Association for Computational Linguistics.

Collobert, R., Bengio, S., and Mariéthoz, J. (2002). Torch: a modular machine learning software library. Technical Report IDIAP-RR 02-46, IDIAP.

Daxenberger, J., Ferschke, O., Gurevych, I., and Zesch, T. (2014). DKPro TC: A Java-based Framework for Supervised Learning Experiments on Textual Data. In

Proceedings of ACL: System Demonstrations, pages 61–66, Baltimore, Maryland. Association for Computational Linguistics.

Dieleman, S., Schlüter, J., Raffel, C., Olson, E., Sønderby, S. K., Nouri, D., Maturana, D., Thoma, M., Battenberg, E., Kelly, J., Fauw, J. D., Heilman, M., de Almeida, D. M., McFee, B., Weideman, H., Takács, G., de Rivaz, P., Crall, J., Sanders, G., Rasul, K., Liu, C., French, G., and Degrave, J. (2015). Lasagne: First release. `http://dx.doi.org/10.5281/zenodo.27878`.

Eckart de Castilho, R. and Gurevych, I. (2011). A Lightweight Framework for Reproducible Parameter Sweeping in Information Retrieval. In *Proceedings of the Workshop on Data infrastructurEs for Supporting Information Retrieval Evaluation*, pages 7–10, New York, NY, USA.

Eckart de Castilho, R. and Gurevych, I. (2014). A broad-coverage collection of portable NLP components for building shareable analysis pipelines. In *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT*, pages 1–11, Dublin, Ireland. Association for Computational Linguistics and Dublin City University.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. (2008). Liblinear: A Library for Large Linear Classification. *Journal of Machine Learning Research*, 9:1871–1874.

Ferrucci, D. and Lally, A. (2004). UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment. *Natural Language Engineering*, 10(3-4):327–348.

Graves, A., Fernández, S., and Schmidhuber, J. (2005). Bidirectional LSTM Networks for Improved Phoneme Classification and Recognition. In *Proceedings of the 15th International Conference on Artificial Neural Networks: Formal Models and Their Applications - Volume Part II*, pages 799–804, Warsaw, Poland. Springer-Verlag.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA Data Mining Software: An Update. 11:10–18.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780.

Joachims, T. (2008). SvmHmm: Sequence Tagging with Structural Support Vector Machines. `https://www.cs.cornell.edu/People/tj/svm_light/svm_hmm.html`.

Lafferty, J. D., McCallum, A., and Pereira, F. C. N. (2001). Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 282–289, San Francisco, CA, USA.

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

Li, J., Chen, X., Hovy, E., and Jurafsky, D. (2016). Visualizing and Understanding Neural Models in NLP. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 681–691, San Diego, California, June. Association for Computational Linguistics.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

McCallum, A. K. (2002). MALLET: A Machine Learning for Language Toolkit. `http://mallet.cs.umass.edu`.

Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239), March.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., Duh, K., Faruqui, M., Gan, C., Garrette, D., Ji, Y., Kong, L., Kuncoro, A., Kumar, G., Malaviya, C., Michel, P., Oda, Y., Richardson, M., Saphra, N., Swayamdipta, S., and Yin, P. (2017). DyNet: The Dynamic Neural Network Toolkit - Technical Report.

Ogren, P. V., Wetzler, P. G., and Bethard, S. J. (2008). ClearTK: A UIMA Toolkit for Statistical Natural Language Processing. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC)*.

Okazaki, N. (2007). CRFsuite: A fast implementation of Conditional Random Fields. `http://www.chokkan.org/software/crfsuite/`.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *The Journal of Machine Learning Research*, 12.

Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: a Next-Generation Open Source Framework for Deep Learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*.

van Merriënboer, B., Bahdanau, D., Dumoulin, V., Serdyuk, D., Warde-Farley, D., Chorowski, J., and Bengio, Y. (2015). Blocks and Fuel: Frameworks for deep learning. *CoRR*, abs/1506.00619. `http://arxiv.org/abs/1506.00619`.

Yosinski, J., Clune, J., Nguyen, A., Fuchs, T., and Lipson, H. (2015). Understanding Neural Networks Through Deep Visualization. In *Deep Learning Workshop, International Conference on Machine Learning (ICML)*.