



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



**TFG del Grado en Ingeniería
Informática**

**NoisyNER - Named entity
recognition in social media
Documentación Técnica**



Presentado por Malte Janssen
en Universidad de Burgos — July 3, 2019
Tutor: Bruno Baruque Zanón

Contents

Contents	i
List of Figures	iii
List of Tables	iv
Appendix A Software plan	1
A.1 Introduction	1
A.2 Project Management	1
A.3 Projects history	2
A.4 Feasibility study	4
Appendix B Requirements Specification	7
B.1 Introduction	7
B.2 General Objectives	7
B.3 Requirements Catalogue	7
B.4 Requirements specification	8
B.5 Use Cases	11
Appendix C Design Specification	13
C.1 Introduction	13
C.2 Data Model	13
C.3 Architecture of the web app	19
C.4 Procedural Design	20
Appendix D Study of applied classifiers	23
D.1 Introduction	23
D.2 Gridsearch of scikit-learn classifiers	23

D.3 Feature extraction	31
D.4 Model Architectures	34
D.5 Parameter Search	35
D.6 Results	37
Appendix E Technical Programming Documentation	39
E.1 Introduction	39
E.2 Directory structure	39
E.3 Programmer's Manual	39
E.4 Installation	42
Appendix F User Manual	43
F.1 Introduction	43
F.2 User requirements	43
F.3 Installation	43
F.4 User manual	43
Bibliography	51

List of Figures

B.1	System's use case diagram	11
C.1	Overall Package Structure	14
C.2	dataLoader Package	16
C.3	logger Package	16
C.4	utils Package	17
C.5	model Package	18
C.6	Web-app Design	19
C.7	get classification sequence	21
D.1	Comparison of learning rates	35
D.2	Comparison of LSTM hidden layer dimensions	36
F.1	Web-application	50

List of Tables

B.1 Requirement «Receive Input»	8
B.2 Requirement «Process input»	9
B.3 Requirement «Classification»	10
B.4 Requirement «Output results»	10
B.5 Use Case - get Classification	12
D.1 Baseline Results	24
D.2 SVM general search	25
D.3 SVM - degree search	25
D.4 Decision Tree - general search	26
D.5 Decision Tree - maximum depth search	27
D.6 Decision Tree - min_samples_split search	27
D.7 Gradient Boosting - Tree parameter search	28
D.8 Gradient Boosting - further optimisation of tree parameters	28
D.9 Gradient Boosting - learning rate + amount of estimator search	29
D.10 Bernoulli Naive Bayes	30
D.11 Multinomial Naive Bayes	30
D.12 Logistic regression - general search	31
D.13 Results of feature-set tests	32
D.14 n-gram backoff functionality	33
D.15 Comparison of model architectures	34
D.16 Final Results	37

Appendix A

Software plan

A.1 Introduction

In the following annex, first the organisational aspects of the study of different NER classifiers and the development of the software are documented. More precisely, the software development process and tools that were used to manage the process are described, followed by an examination of the course of the project. Afterwards follow the results of the studies mentioned in the memoria and a user guide to the software.

A.2 Project Management

Scrum The project's management is inspired by the Scrum model used in agile software development. The model is based on the assumption that projects are too big to be planned in its entirety at the start. Therefore only a rough outline is made at the start. The project is divided into several milestones that provide an agile approach.

Scrum is a team based approach to project management. Due to the fact that this bachelor thesis is only written by one person the majority of the concepts can't be applied exactly as intended by Scrum. Consequently the project management approach is only loosely based on Scrum.

One concept that is applied are Sprints. In this case most sprints cycles had a duration of approximately a month. Some are bigger and some are smaller due to the complexity of tasks at hand and the time available. Sprint

meetings between the author and the project’s coordinator were held every two weeks, usually around the middle and end of each sprint. In the meeting in the middle the tasks progress was discussed, while in the meeting at the end the results of the sprint was discussed and the next sprint was vaguely planned. The second meeting can therefore be seen as the Sprint Planning and Sprint Review. The project’s coordinator can be seen as the Project Owner of the Scrum model, prioritising tasks and guiding the project’s direction.

Github and ZenHub The project is hosted on Github. At the beginning to organise tasks waffle was used. After waffle shut down, project organisation was done with the help of ZenHub. Zenhub provides a board to visualise tasks, as well as it provides an overview of the remaining workload. It also offers several tools to inspect work velocity. Each Github issue is assigned a amount of story points, estimating the tasks size. In this case each point is the equivalent of the workload of about 1-2 hours.

A.3 Projects history

The Kick-Off Meeting took place in the second week of December 2018. The elemental ideas of the project were discussed. Due to exams and other private responsibilities the project wasn’t directly started after the meeting. Instead the 9. of January marked the beginning of the project.

Some milestones were smaller than others in a similar time frame. This is due to responsibilities of other classes and exam periods which reduced time availability during the semester. Also since nearly all concepts and libraries used were new it took a while to get the ball rolling. The next paragraphs give an overview over the the phases of development.

Milestone 1 (9.01-14.01.2019): Initial project setup/ Research regarding NLP

In the first relatively small Milestone the projects infrastructure was set up. Also some research regarding the theoretical concepts of NLP and NER were done and documented.

Milestone 2 (8.02-31.02.2019): NLTK-Experiments/ Preprocessing Data

The second Milestone consisted of researching and experimenting with one of the libraries (NLTK) used later on in the project. An initial NER chunker was introduced and functions preprocessing the Data were written. Ideally this milestone would have been bigger and the project would have advanced much more, but my laptop broke and had to be send in. Due too the bad infrastructure of the university not much could have been worked on.

Milestone 3 (01.03-08.04.2019): NLTK

Milestone 3 was the biggest Milstone until that point. Things got serious as different NLTK chunkers got introduced and a training script was written. Initially the due date was the end of march. But some complications in having to figure out how to train NER chunkers without any use cases in the internet combined by the midterm exam period it had to be postponed by a week.

Milestone 4 (09.04-13.05.2019): ~~Other Classifiers~~ Deep Learning Classifier

In the initial plan it was planned to create a few other classifiers with other libraries such as Stanford Core, Sklearn and OpenNLP. But that plan changed a bit. Together with the projects coordinator the decision to develop a different classifier using more advanced machine learning techniques was made. Do to the postponement of the last milestone, exams and working on a bigger project in another class most of April couldn't have been used for this milestone as the commit history suggests. The project of the other class was related to this theme as it covered sentiment analysis with the sckikit-learn library. The concepts learned and applied there were adapted towards named entity recognition and incorporated into this project.

Milestone 4 (01.05-16.06.2019): ~~Evaluation~~ Evaluation/ Improve Deep Learning Classifier

At the start of the project this milestone was called Evaluation in which all prior classifiers were supposed to be evaluated and documented. But due to the complexity of the task at hand the deep learning classifier didn't perform at its best. Therefore this phase was only partially used to do experiments and the bigger focus was improving the performance of the deep learning

classifier. The validation performance was increased from a validation F1 score of 7 to a F1 score of around 35. All work was done by the 16.06 but the commit didn't go through and probably shown error message was overseen. Therefore it appears like the milestone dragged on for longer than it was worked on.

Milestone 5 (17.06-23.06.2019): Web App

The fifth, short milestone was used to create a web app with the help of flask and HTML which posed several challenges.

Milestone 6 (14.06-02.07.2019)

The last remaining time was mainly used to document the steps that were not documented in the prior milestones yet. Also a comparison of different deep learning architectures was carried out. Furthermore the code was organised differently in some cases.

A.4 Feasibility study

Since this project was not made to create an application that would be sold but rather a scientific study this viability study is rather short.

Economic viability

The study was realised by the author of the presented bachelor thesis. As the university degree is not yet finished, the hourly wage is estimated at 14 €/h. In order to calculate the total development costs, the time invested in the project is examined. Because the realisation of the bachelor thesis is evaluated with 12 ECTS points and a workload of 30 hours per ECTS point is usually assumed, a total amount of around 360 hours were dedicated to the projects development. This also coincides with the amount of story points done. About 200 story points were finished and with a calculation of one to two hours per story point this coincides with the calculated 360 hours..

This leads to the following cost calculation://

$$360h * 14 = 5040\text{€} //$$

A possible way to monetise the project would be to offer research services to a smaller/ medium sized company that doesn't have their own machine learning department. The kinds of companies that could be interested are for example email providers. A minimum price of 5040€ would have to be charged for the project.

Appendix B

Requirements Specification

B.1 Introduction

This chapter contains requirement specifications for the developed software. It provides an overview over the software's required functionality. Since the main goal was not to develop an application, but to do more of a scientific study, the requirements are few and concern only the developed demo web application.

B.2 General Objectives

The sole objective of the web-app is to demonstrate the process of named entity recognition using the developed deep learning classifier.

B.3 Requirements Catalogue

This section lists the apps functional requirements.

Actors The only actor of the software is it's user. There are no different roles an actor can have, as the only actor he has the whole feature-set of the program available. His goal is to get as accurate named entity predictions as possible.

B.4 Requirements specification

Table B.1: Requirement «Receive Input»

ID:	Name: Receive Input
Description	The user can provide input text
Process Description	The user has the ability to introduce text into the applications text box. The text is then passed into the application by pressing the "classify" button.

Table B.2: Requirement «Process input»

ID:	Name: Process input
Description	After text is received it is processed it into the data formats processable by the deep learning classifier
Process description	Whenever new Text is entered into the application it is read and converted into a format readable by the system in order to work with that data.
Data conversion	<ol style="list-style-type: none"> 1. The input string data is word tokenised 2. Each word that was returned by the tokenisation is replaced by it's index in the vocabulary if possible. If not the index of the unknown word is used. 3. For each word all characters are replaced by it's index in the character vocabulary if possible. If not the index of the unknown word is used. This results in a list of list filled by a representation of indices for each word. 4. Each element of the in the previous step created list is padded with zeros until each word list has the same length as the longest word. This is done because the neural network expects all inputs to have the same lengths. 5. The in step 2 and 4 created lists are converted into Tensors.

Table B.3: Requirement «Classification»

ID:	Name: Classification
Description	The application feeds the in B.2 created tensors into the model and gets the predictions returned.

Table B.4: Requirement «Output results»

ID:	Name: Output results
Description	The results received from the model (B.3) the system outputs the results.
Process description	For better visualisation the results are outputted as an image. The image contains the original text with found entities being highlighted in specific colours
Colours	<ol style="list-style-type: none"> 1. Person: green (#1f5a07) 2. Corporation: red (#9f0120) 3. Creative-work: pink (#ff2770) 4. Group: blue (#4e4e94) 5. Location: yellow (#eae11a) 6. Product: purple (#941aea)

B.5 Use Cases

The in figure B.1 shown use case diagram contains the the only use case available to the user. The user can request a classification for a given input text. To return the classification to the user several actions, including loading the model, tokenisation of the text, conversion into numerical representation and the actual classification have to be taken. This procedure is further shown in table B.5.

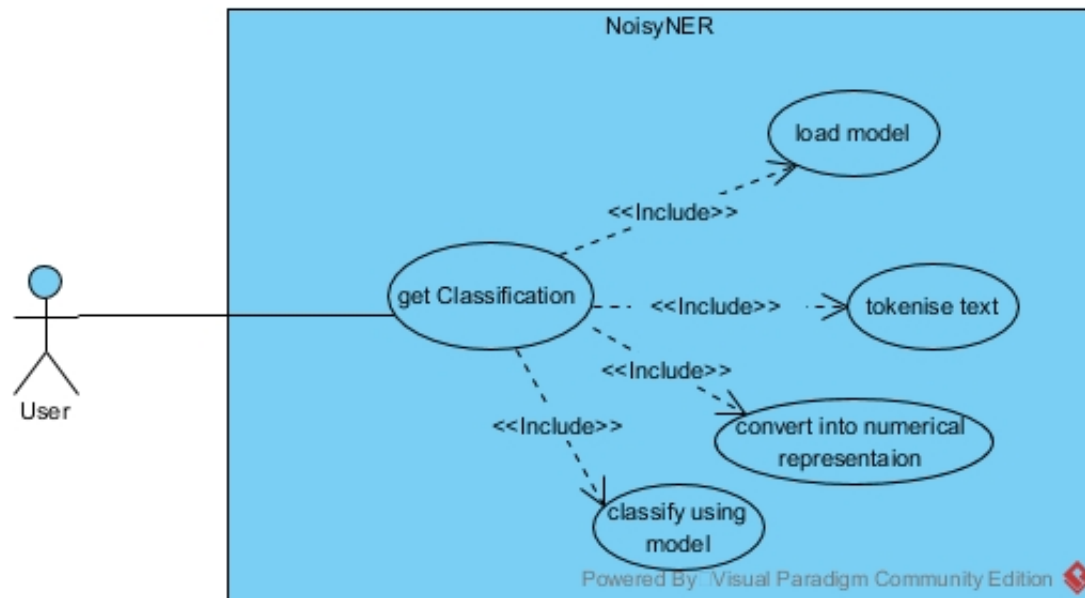


Figure B.1: System's use case diagram

Table B.5: Use Case - get Classification

Use Case	get Classification		
Description	The user enter requests the classification of named entities within a text.		
Preconditions	User has written text into the text field		
Trigger	User presses the "classify" Button		
Flow of events	Step	Action	
	1.	Data gets pre-processed	
	2.	pre-processed data is handed to the model	
	3.	classification by the model	
	3.	model returns classification	
	3.	output classification	

Appendix C

Design Specification

C.1 Introduction

This part of the annex serves as a description of the software's implementation, structure and way of functioning.

C.2 Data Model

Package Structure

The package structure is inspired by the PyTorch Template Project^[1] which attempts to provide a clear folder structure which is suitable for many deep learning projects.

Note: The structure of the pytorch package is not ideal. Basically there is the main pytorch package, which includes a heroku-app package which contains a lot of duplicate classes which are already contained within the pytorch package. Ideally the flask app would have been inside the main pytorch package, but due to the limitations of size on heroku this was not possible. Therefore the for the demo app necessary files have simply been copied into the heroku-app package to be hosted on heroku. This part of the structure will not be further examined as most of it is also contained within the pytorch package.

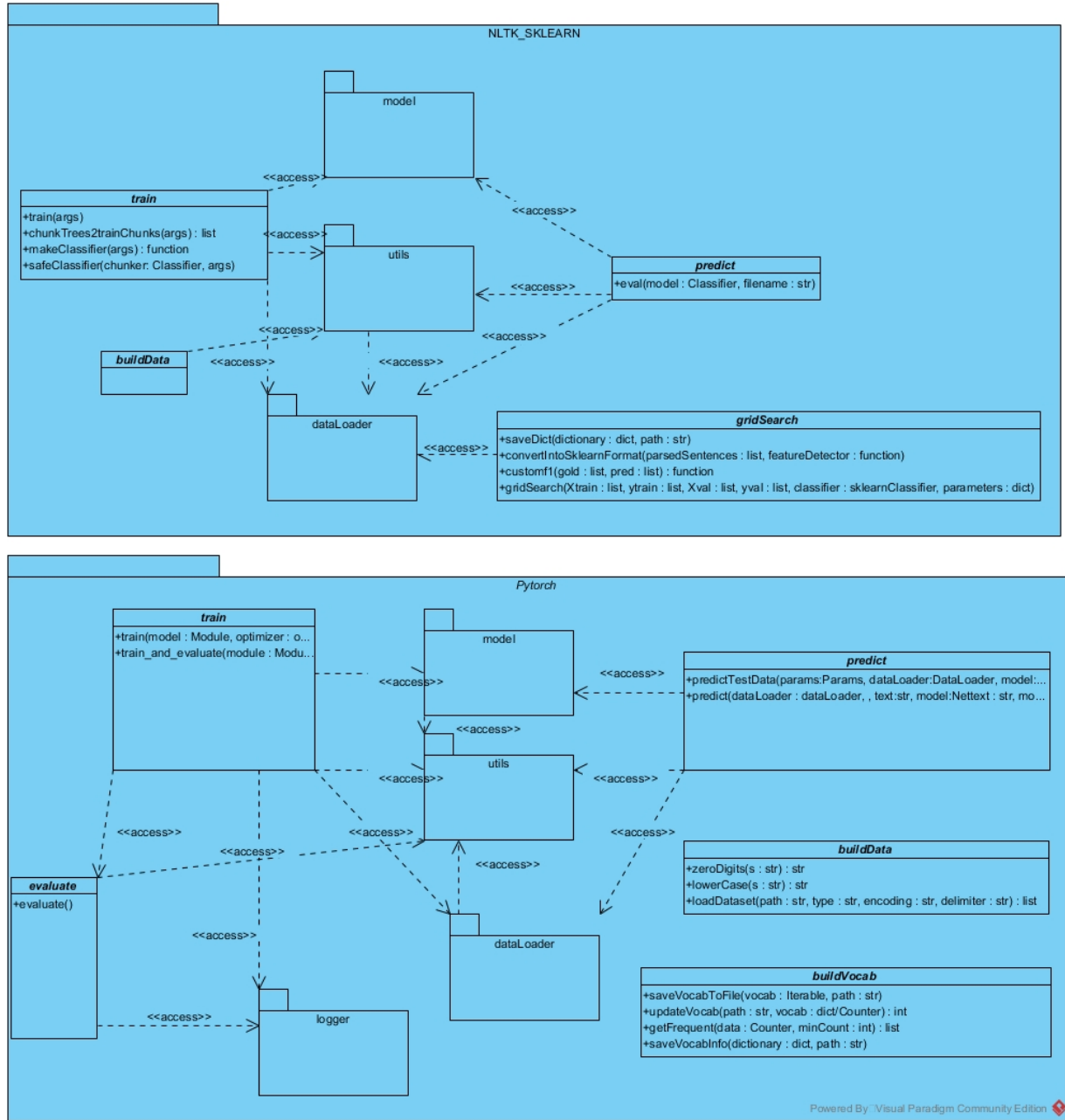


Figure C.1: Overall Package Structure

The general code structure and its dependencies are visualised in diagram C.1. The repositories code is divided into two main packages: The

NLTK_SKLEARN package and the pytorch package. Both packages have a very similar structure. The only big difference is that the pytorch package also contains a logger package to log the training process. Due to their similarity and the pytorch path being the more interesting one, the next design sections will concentrate on the bigger and more interesting part of the project, the pytorch part.

Python doesn't force the programmer to use a pure object orientated approach. The repositories code uses classes, but only where it makes sense, other code is written inside python modules, which in this case either act as scripts or modules offering functionality to classes or scripts.

Most of the portrayed classes are in reality not classes but python modules and are included to get a better overview of the project. They are marked as abstract classes for differentiation.

The classes are not shown in a single diagram but are split into several class diagrams based on their respective package for reasons of clarity and comprehensibility. A description of the packages containing the softwares classes is provided in the following sections.

pytorch package

In the pytorch package several python scripts can be found. It includes the scripts to load the data (buildVocab and buildData), as well as the training, evaluation and prediction scripts. These are the executable parts of the program and will be explained in further detail in the manual. The package also contains several subpackages which will be further looked at in the following sections.

dataLoader

The dataLoader package shown in [C.2](#) contains the DataLoader class. It is responsible for creating the word, character and tag mappings. It also includes functionality to generate batches of data and translates mapped data into Data processable by the neural network.

logger

The logger [C.3](#) is utilised by the training script to document the training process and results.

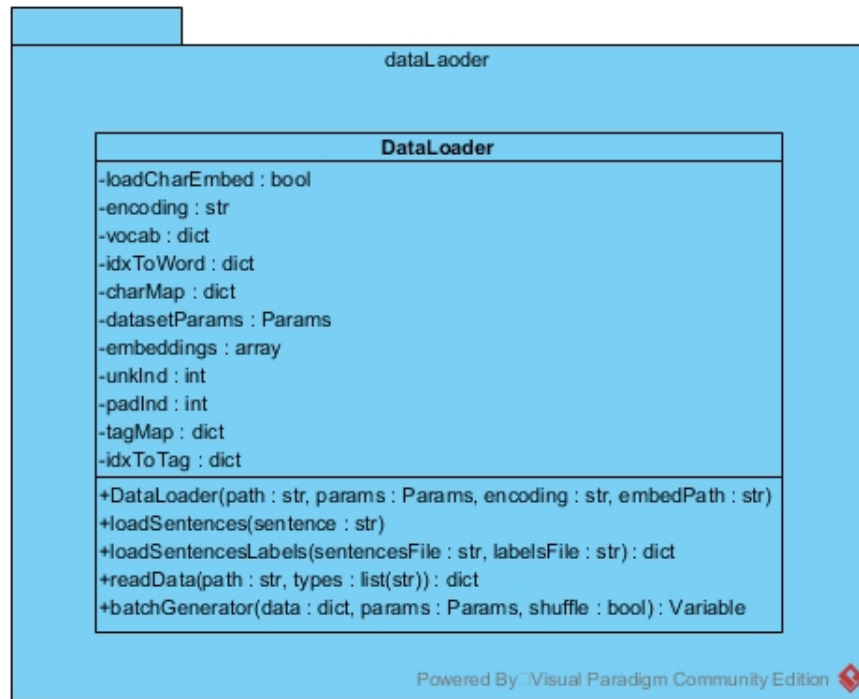


Figure C.2: dataLoader Package

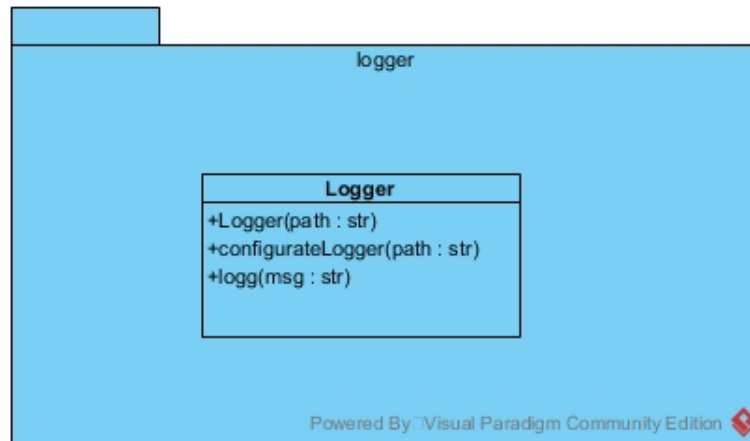


Figure C.3: logger Package

utils

The utils package C.4 contains various utility function used by nearly all other scripts. For instance there are functions to load and save the current state of the model. It also contains two utility classes. The first one is the RunningAverage class which computes a average of the loss functions and is used in the training and evaluation script. The second class Params is used to store the models parameters. This includes dataset parameters and also model architecture parameters.

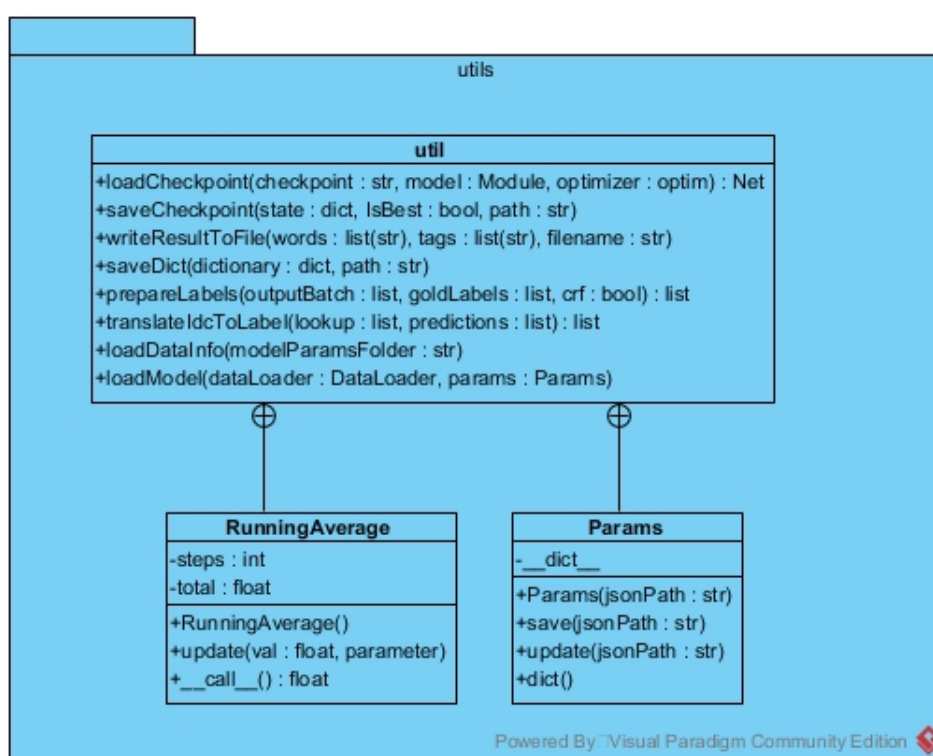


Figure C.4: utils Package

model

This package (C.5) contains classes related to the inner workings of the deep learning model. The class Net represents the model and contains, depending on the models architecture, the layers (classes) inside the layers package. The package also includes the modules eval and loss defining the models loss and evaluation functions.

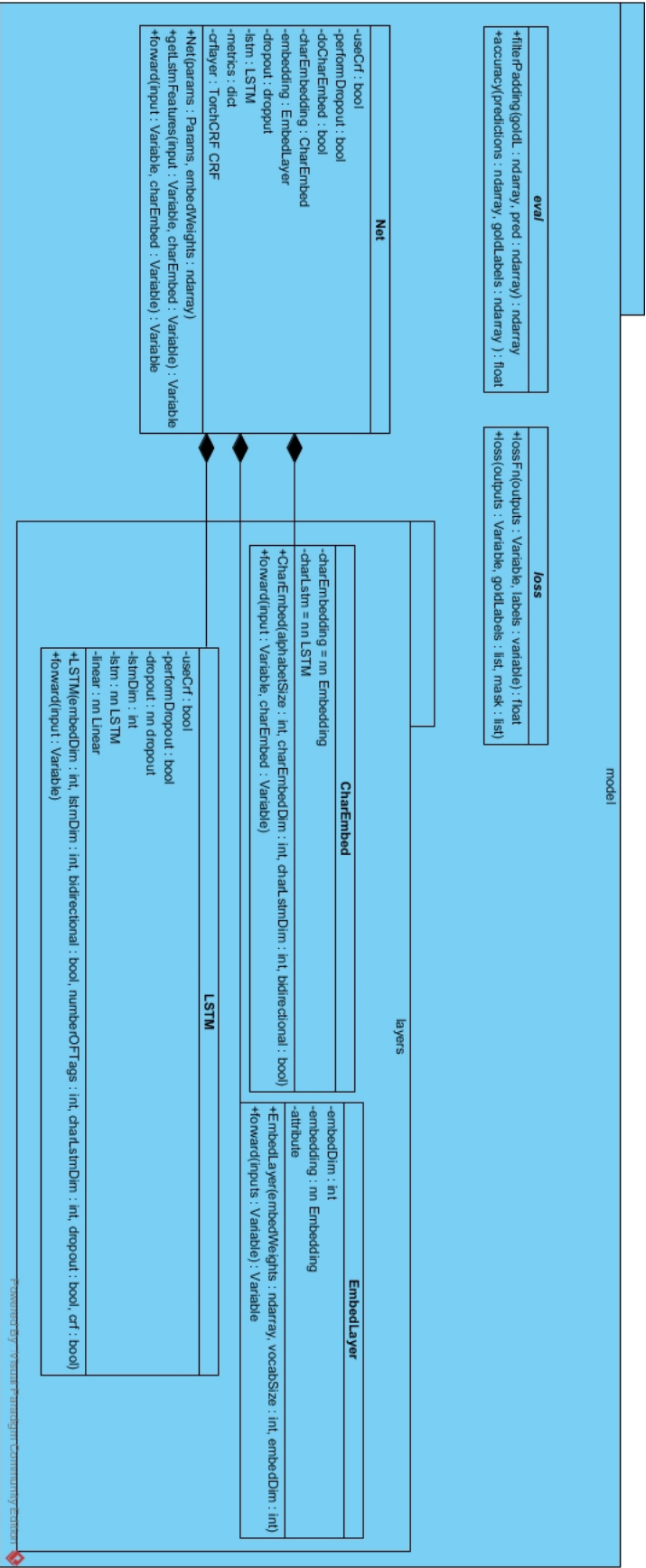


Figure C.5: model Package

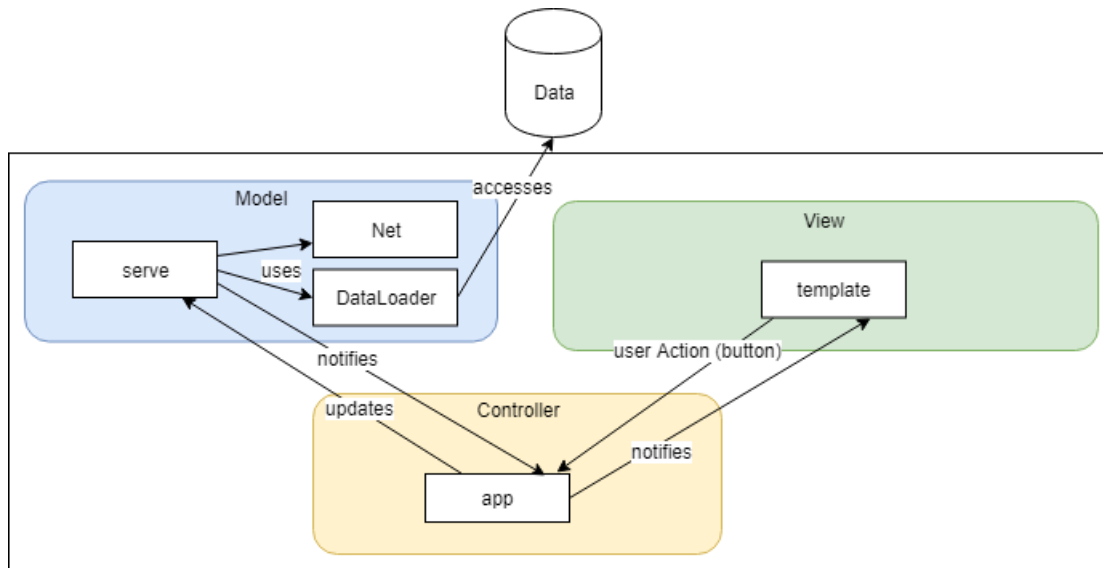


Figure C.6: Web-app Design

C.3 Architecture of the web app

Apart from the deep learning aspects of the code the project contains a demo web application. This section provides an overview of the design of said application. As figure C.6 indicates the application itself uses the model-view-controller design pattern. The diagram is not an official UML diagram but rather provides an overview about the design idea. Once a user inputs text to the (HTML)view it is passed to the controller which contains the app module. This module is used for routing. It updates the model with the text information. The model contains the logic of the application. Here the serve module receives the data and initialises the deep learning model and reloads it's weights with the help of the DataLoader. The deep learning model is accessed through the serve module whichs task it is to convert the data into data processable by the model and also acts as a wrapper to the deep learning model. To wrap the model it contains a getModelApi function that returns a lambda function able to process the request. Once the request is processed the controller is notified with the classifications and updates the HTML view.

C.4 Procedural Design

The sequence diagram shown in figure C.7 shows the sequence of actions happening once a user inputs data and request a classification. It has to be noted that since not using a pure object oriented approach the UML sequence diagram is mainly used for visualisation of how modules/objects work together. Serve and app are modules, Net and DataLoader classes and the view is a HTML template.

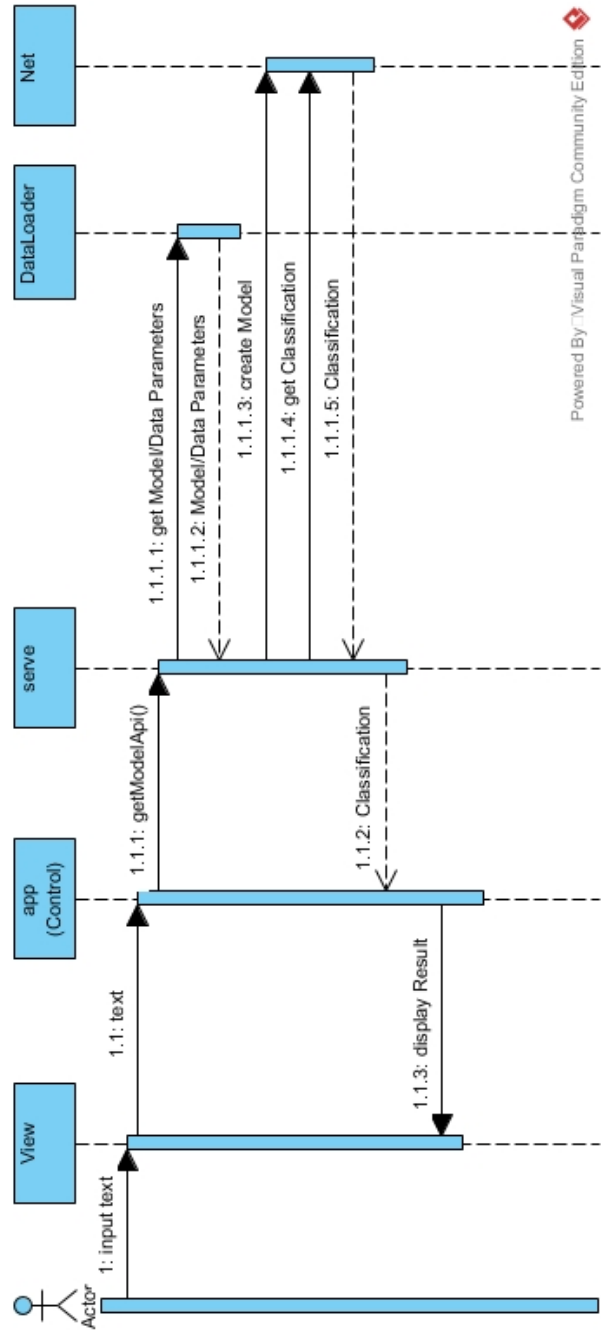


Figure C.7: get classification sequence

Appendix D

Study of applied classifiers

D.1 Introduction

This chapter contains several studies and comparisons of the in the memoria mentioned classifiers. They are done to find the best configuration of each classifier as well as to find the most appropriate classifier for the task.

D.2 Gridsearch of scikit-learn classifiers

As discussed in the memoria an extensive gridsearch was carried out to find each classifier. The following subsections contain the results of the gridsearch. F1 scores were rounded. Furthermore, it also provides a small description of each hyperparameter.

The results are merely summaries of the results, as to include complete tables would take up too much space with little to no information gain. For the complete results look at the following location: `\NLTK_SKLEARN\results\gridtest`

The portrayed F1 scores are computed by the default scikit-learn f1 score and do not keep the sequential nature of the data in mind. This is done to find the parameters which encourage the classifier to find as many as possible of the entities, even if only partially.

classifier	F1 score
MultinomialNB	0.0
BernoulliNB	0.0
SVC	0.0
LogisticRegression	0.358
DecisionTree	0.352
GradientBoosting	0.292

Table D.1: Baseline Results

Baseline Results

Before doing the gridsearch the F1 score for a baseline configuration of each model was calculated. Table D.1 shows the results. Three of the tested classifiers failed to capture any entity. They classified every single token as "O" (outside of chunk). The best performing baseline configuration is the Logistic Regression classifier with a token-level f1 score of 0.358 on the validation data.

SVM

Tuned parameters:

- **Kernel** kernel type
- **C** Penalty parameter of the error term - how much classification errors are penalised. This influences the smoothness of the decision boundary (low c: smooth decision boundary, high C: unsmooth decision boundary (more samples as support vectors)). Bias Variance Tradeoff: large C: low bias, high variance; small C: high bias, low variance
- **Gamma** Gamma defines how far the influence of single training example reaches. If the value of Gamma is high, then the decision boundary will depend on points close to the decision boundary and nearer points carry more weights than far away points.
- **degree** only applies to polynomial kernel function. Defines the flexibility of decision boundary (1 = linear).

Table D.2 shows that rbf and linear kernels yield the best results. The best result is yielded by a rbf support vector machine with a C value of 10

Kernel	C	Gamma	F1-Score	Kernel	C	Gamma	F1-Score
linear	1.0	-	0.393	sigmoid	1	0.1	0.314
linear	10.0	-	0.391	sigmoid	1	1.0	0.163
linear	100.0	-	0.384	sigmoid	1	0.01	0.085
rbf	1	0.1	0.344	sigmoid	10.0	0.1	0.351
rbf	1	0.01	0.170	sigmoid	100.0	0.1	0.326
rbf	1	1.0	0.076	sigmoid	1.0	0.1	0.314
rbf	10.0	0.1	0.399	poly	1	10.0	0.273
rbf	1.0	0.1	0.344	poly	1	1.0	0.273
rbf	0.1	0.1	0.084	poly	1	0.1	0.138

Table D.2: SVM general search

Kernel	C	Gamma	degree	F1 Score
poly	1.0	10.0	1	0.391
poly	1.0	10.0	2	0.359
poly	1.0	10.0	3	0.273
poly	1.0	10.0	5	0.145

Table D.3: SVM - degree search

and a gamma of 0.1. These values result in the best bias-variance tradeoff. As expected too high C values and too low gamma values (lower than 0.1) resulted in worse results, as they lead to high variance. The same can be said for low C values (smaller than 1) and high gamma values (usually bigger than 0.1) in relation to the bias.

The poly Kernel was also tested with different degree values as can be seen in table D.3. A lower degree resulted in better results, but didn't increase the previously found best score of 0.399.

By doing a parameter search it was possible to make the SVM, that failed to capture any entities with the baseline configuration, work for the given problem. The performance was increased by nearly 40%.

Best set of Parameters:

C: 10

Gamma: 0.1

Kernel: rbf

criterion	max_depth	min_samples_leaf	F1 score
entropy	100	1	0.375
entropy	100	5	0.369
entropy	50	1	0.373
entropy	50	5	0.316
gini	100	1	0.370
gini	100	5	0.370
gini	50	1	0.356
gini	50	5	0.370

Table D.4: Decision Tree - general search

Decision Tree

Tuned parameters:

- **criterion** The function to measure the quality of a split.
- **max_depth** maximum depth of the tree
- **min_samples_leaf** minimum number of samples required to split an internal node
- **min_samples_split** minimum number of samples required to be at a leaf node. `min_samples_leaf` and `min_samples_split` decide if a split is made.

The first search (D.4) showed that using information gain (entropy) to measure the quality of a split is slightly better than using gini impurity, although it took a little longer to compute (presumably because of calculating a logarithm).

Furthermore it can be observed that allowing splitting nodes even if it results in a node with just one example was ideal (`min_samples_leaf = 1`). Having this parameter small is important in this very unbalanced dataset since in most regions the minority classes (entities) will be very few.

Next the maximum depth of a tree was looked at. The results are showed in table D.5. High values did not lead to overfitting (at least the ones tested up to 500). Values lower than 250 did not capture all useful pattern. A depth of 250 and 500 yielded the best results. Since the f1 score of both is the same 250 was chosen as the ideal value due to potentially lower time complexity.

criterion	max_depth	min_samples_leaf	F1 score
entropy	500	1	0.382
entropy	250	1	FFCCC90.382
entropy	100	1	0.375
entropy	50	1	0.373

Table D.5: Decision Tree - maximum depth search

criterion	max_depth	min_samples_leaf	min_samples_split	F1 score
entropy	500	1	2	0.382
entropy	250	1	3	0.382
entropy	100	1	5	0.375
entropy	50	1	10	0.373

Table D.6: Decision Tree - min_samples_split search

The minimum samples to do a split was also looked at as can be seen in table D.6. Using the minimum amount of 2 yielded the best results (0.382 F1 score). The F1 score was increased from 0.352 with default parameters to 0.382 with the best set of found parameters.

Best set of Parameters:

criterion: entropy
max_depth: 250
min_samples_leaf: 1
min_samples_split: 2

Gradient Boosting

The criterion parameter was not tuned to save time and because the scikit-learn documentation says: "The default value of "friedman_mse" is generally the best as it can provide a better approximation in some cases." [2]

tuned parameters: For parameters min_samples_leaf, min_samples_split, max_depth see Decision Trees.

- **subsample** fraction of samples to be used for fitting the individual base learners
- **learning_rate** learning rate shrinks the contribution of each tree
- **loss** loss function to be optimised

max_depth	min_samples_leaf	min_samples_split	F1-score
20	5	8	0.383
20	5	2	0.383
20	5	4	0.383
10	5	8	0.375
10	5	2	0.375
10	5	4	0.375
8	5	8	0.371
8	5	2	0.371
8	5	4	0.371
20	3	8	0.361
10	3	8	0.368
8	3	8	0.354

Table D.7: Gradient Boosting - Tree parameter search

max_depth	min_samples_leaf	min_samples_split	F1-score
20	5	8	0.387
20	10	8	0.362
20	20	8	0.388
0	30	8	0.391
20	50	8	0.386
20	100	8	0.389

Table D.8: Gradient Boosting - further optimisation of tree parameters

First a smaller number of estimators (80) was used to tune the tree parameters. This allows for more tests, since training takes less time. In table D.7 it can be observed that increasing the maximum depth of trees from the default of one had a major effect on performance. Furthermore it can be observed that higher numbers of minimum leaf samples (`min_samples_leaf`) did better than lower numbers. The parameter `min_samples_split` barely had an effect though.

After the first general tests the tree parameters were further tuned as can be seen in table D.8. Even further increasing the `min_samples_leaf` parameter to 30 increased performance a little.

Next the learning rate was decreased and the amount of estimators increased. By decreasing the learning rate to 0.03 and at the same time

learning_rate	n_estimators	max_feats	F1-score
0.1	80	None	0.391
0.08	100	None	0.383
0.08	500	None	0.384
0.08	250	None	0.393
0.05	500	None	0.390
0.05	250	None	0.391
0.03	250	None	0.391
0.03	500	None	0.394
0.03	500	log2	0.291

Table D.9: Gradient Boosting - learning rate + amount of estimator search

increasing the amount of estimators the performance was slightly increased as can be seen in table D.9. Increasing the amount of estimators makes the model more powerful and vulnerable to overfitting. To counteract this the importance of each estimator has to be decreased by decreasing the learning rate.

Best set of Parameters:

learning_rate: 0.03
n_estimators: 500
max_feats: None
min_samples_split: 2
min_samples_split: 8

Bernoulli Naive Bayes

tuned parameters:

- **alpha** Additive (Laplace) smoothing parameter. This is basically a 'fail-safe' probability in case a word is not seen before.

Before doing a parameter search both Naive Bayes classifiers failed to predict any entities and only predicted the majority class "O". Decreasing the alpha value increased results in both cases. Tables D.10 and D.11 suggested that doing close to no smoothing at all resulted in the best performance.

alpha	F1-score
1	0.0
0.8	0.0
0.6	0.0
0.4	0.0
0.2	0.0
0.1	0.086
0.000001	0.297

Table D.10: Bernoulli Naive Bayes

alpha	F1-score
1	0.0
0.8	0.0
0.6	0.001
0.4	0.024
0.2	0.099
0.1	0.189
0.000001	0.285

Table D.11: Multinomial Naive Bayes

Logistic Regression

tuned parameters:

- **penalty** norm used in the penalization.
- **C** penalty term - see SVM parameter description
- **solver** optimisation method
- **max_iter** maximum number of iterations

The results, shown in table [D.12](#) show that most solvers performed similarly. Whenever the C value was smaller than 1 the classifier failed to produce decent results (see complete results). The best results were produced by a classifier with the saga solver, a C value of 1000, and the l1 penalty function. With these parameters the baseline performance was improved from by almost 5% from 0.358 to 0.410 F1 score.

solver	C	penalty	F1 score	solver	C	penalty	F1 score
saga	10.0	l2	0.389	liblinear	100.0	l2	0.400
saga	100.0	l2	0.401	liblinear	1000.0	l2	0.4002
saga	1000.0	l2	0.403	liblinear	1000000.0	l2	0.397
saga	1000.0	l1	0.410	liblinear	1000.0	l1	0.365
sag	10.0	l2	0.389	lbfgs	1000.0	l2	0.407
sag	100.0	l2	0.402	lbfgs	100000.0	l2	0.408
sag	1000.0	l2	0.401	lbfgs	1000000.0	l2	0.404
newton cg	100.0	l2	0.399	newton cg	1000.0	l2	0.400

Table D.12: Logistic regression - general search

Best set of Parameters:

solver: saga

C: 1000

penalty: l1

D.3 Feature extraction

As discussed in the memoria a search of the optimal set features was done.
The tested features were:

- **word(w)** word
- **POS-tag(p)** POS-tag of the token
- **class(c)** entity(class) of the token
- **shape(s)** shape of the word (eg.: Upper-casE -> Xxxxx-xxxX)
- **is digit(d)** whether the is a digit
- **is uppercase(u)** whether the word is uppercased
- **is title(t)** whether the word starts with a uppercased letter and all other letters are lowercase
- **length(l)** length of the word

Feature-set	F1 Score
w, w-1, w+1	2.4
w, w-1, w+1, p, p-1, p+1	7.9
w, w-1, w+1, p, p-1, p+1, c-1	9.7
w, w-1, w+1, p, p-1, p+1, c-1, s	11.5
w, w-1, w+1, p, p-1, p+1, c-1, s, s+1	9.4
w, w-1, w+1, p, p-1, p+1, c-1, s, s-1, s+1	8.2
w, w-1, w+1, p, p-1, p+1, c-1, s, s-1	9.1
w, w-1, p, p-1, p+1, c-1, s	13.1
w, p, p-1, p+1, c-1, s	15.2
p, p-1, p+1, c-1, s	10.0
w, p, p-1, c-1, s	10.3
w, p, p-1, p+1, c-1, s, d	15.4
w, p, p-1, p+1, c-1, s, d, u	15.5
w, p, p-1, p+1, c-1, s, d, u, t	17.8
w, p, p-1, p+1, c-1, s, d, u, t, l	17.2
w, p, p-1, p+1, c-1, d, u, t, l	17.7

Table D.13: Results of feature-set tests

A -/+1 describes the previous/next token.

The results of the tests are visible in table D.13. They showed that the word itself is important, but that the surrounding words do not increase performance. In contrast the POS-tags of surrounding words did contribute to better results. The reasons for this are kind of obvious because languages are structured, so the type of word of surrounding words contains valuable information. Despite of the nature of data, which has in contrast to formal text less structure, some structure does remain. The entity of the prior word also had an influence. This is due to that an entity I-Person is more likely to follow B-Person than for example I-corporation. Furthermore the shape of a word had an influence on performance. This can likely be contributed to very short words not likely being an entity. The shape also captures whether a word contains special characters and information about lower/upper casing. Despite that information already partially represented by the shape the feature t (is title) had a big influence, increasing the F1 score by over 2%. Factors slightly increasing the score were whether a word is a digit and uppercased which can be explained by knowing that a word is a digit, makes it very unlikely being an entity. The only proposed feature not included in the final feature selection function was the length of a word, which does

not mean that this feature does not contain any valuable information. On the contrary it added a few percent (visible in the last row), but only if the shape feature was not included. This can be explained by the shape feature already covering the length of a word. Including the shape feature instead resulted in a slight increase though (0.1%) because it also contained additional information as explained earlier. The shapes of the surrounding word did not increase the results.

n-gram Taggers - Backoff

The n-gram backoff functionality was tested. These are classifiers which have a backoff tagger. A Bigram tagger for instance would if it failed to predict an entity for a word try its backoff, the Unigram tagger, as well. The resulting F1 scores have the sequential nature of sentences in mind. Here no parameter had to be tuned. Table D.14 suggests that unigram taggers are far superior to bi- or tri-gram taggers, but that they were able to classify some entities that unigram taggers couldn't, meaning the backoff functionality slightly increased the F1 score.

Classifier	F1-score
Unigram	0.051
Bigram	0.005
Trigram	0.002
Bigram - backoff	0.0530
Trigram - backoff	0.0531

Table D.14: n-gram backoff functionality

D.4 Model Architectures

Different model setups were tested to observe the influence of the different techniques used. Portrayed F1-scores are validation scores. The results shown in table D.15 show that the best performance is reached with the BI-LSTM CRF model utilising pre-trained embeddings and a char embedding layer. The biggest factors in terms of F1 score were using pre-trained word embeddings and character embeddings. The pre-trained word embeddings contributed additional information which couldn't have been captured by looking at the training data alone, while the character embeddings improved the model because the data contains a lot of misspellings and rare words. Leveraging not only past, but also future features, by using a bidirectional LSTM also had a small effect. The BI-LSTM with pre-trained embedding performed 2.6% better than the unidirectional one. The dropout functionality generally did not increase the F1 score. In one case (BI-LSTM pretrainedEmbed + charEmbed + Dropout) it increased the score by 0.2%, but usually it decreased the score. It results in a network that converges slower and therefore needs more iterations to overfit on the training data, but that didn't increase overall performance. The by Hinton et al. suggested dropout of 0.5.[3] was used in these tests. Testing different values did not improve the model. Using a CRF layer only slightly increased F1 score (1%), suggesting that the surrounding tag information is not that informative but nevertheless adds to the model.

Model	F1-score
LSTM	9.0
LSTM pretrainedEmbed	21.8
BI-LSTM pretrainedEmbed	24.4
Bi-LSTM pretrainedEmbed + Dropout	16.0
BI-LSTM pretrainedEmbed + charEmbed	35.4
BI-LSTM pretrainedEmbed + charEmbed + Dropout	35.6
BI-LSTM CRF pretrainedEmbed + charEmbed + Dropout	34.7
BI-LSTM CRF pretrainedEmbed + charEmbed	36.4

Table D.15: Comparison of model architectures

D.5 Parameter Search

The model was tuned testing different learning rates and LSTM dimensions. Further tests were not possible because of limited resources. Portrayed F1 scores are validation scores. The model used for the tests was the best performing model from the prior section (BI-LSTM CRF pretrainedEmbed + charEmbed). The results of testing different learning rates, showed in table D.1, show that the initially picked learning rate of $1e-3$ was by far the best learning rate. A lower learning rate took too long to converge anywhere near a local optima while the higher learning rate (blue line) was too high and got stuck at just under a F1 score of 20.

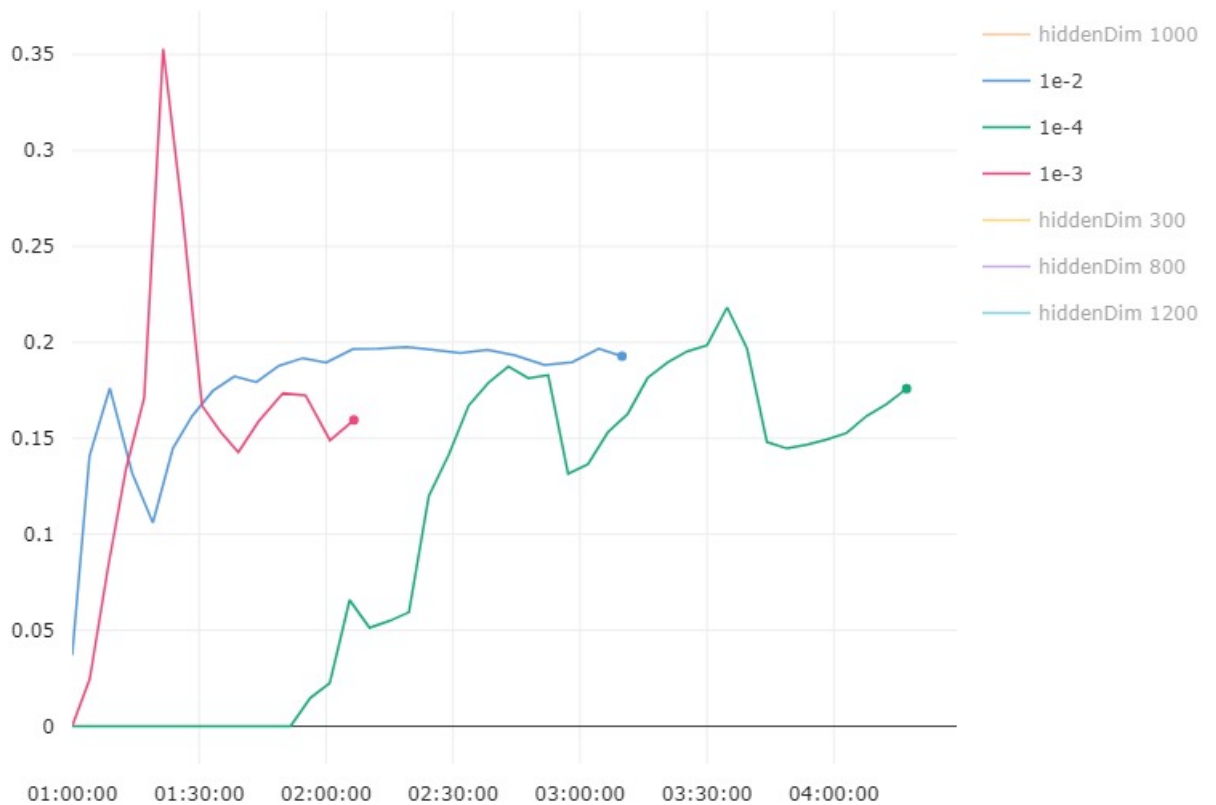


Figure D.1: Comparison of learning rates
 xAxis: F1 yAxis: Training time

The hidden dimension of the LSTM layer appeared to not have a huge influence(D.2. Usually a higher dimension leads to a more powerful network that is at the same time more prone to overfitting. The poorest results were produced by a model with a dimension of 1200, suggesting that a model with a dimension over 1000 is too powerful. Although this has to be taken with caution because of random aspects of neural networks another run could have led to different results. If more resources would have been available a averaged test could have produced more reliable results. The best score was produced by a model with a hidden dimension of 1000 (36.4), closely followed by dimensions 300, 600 and 800. The score of 36.4 was previously also achieved by a network with a LSTM dimension of 600 as can be seen in D.15.

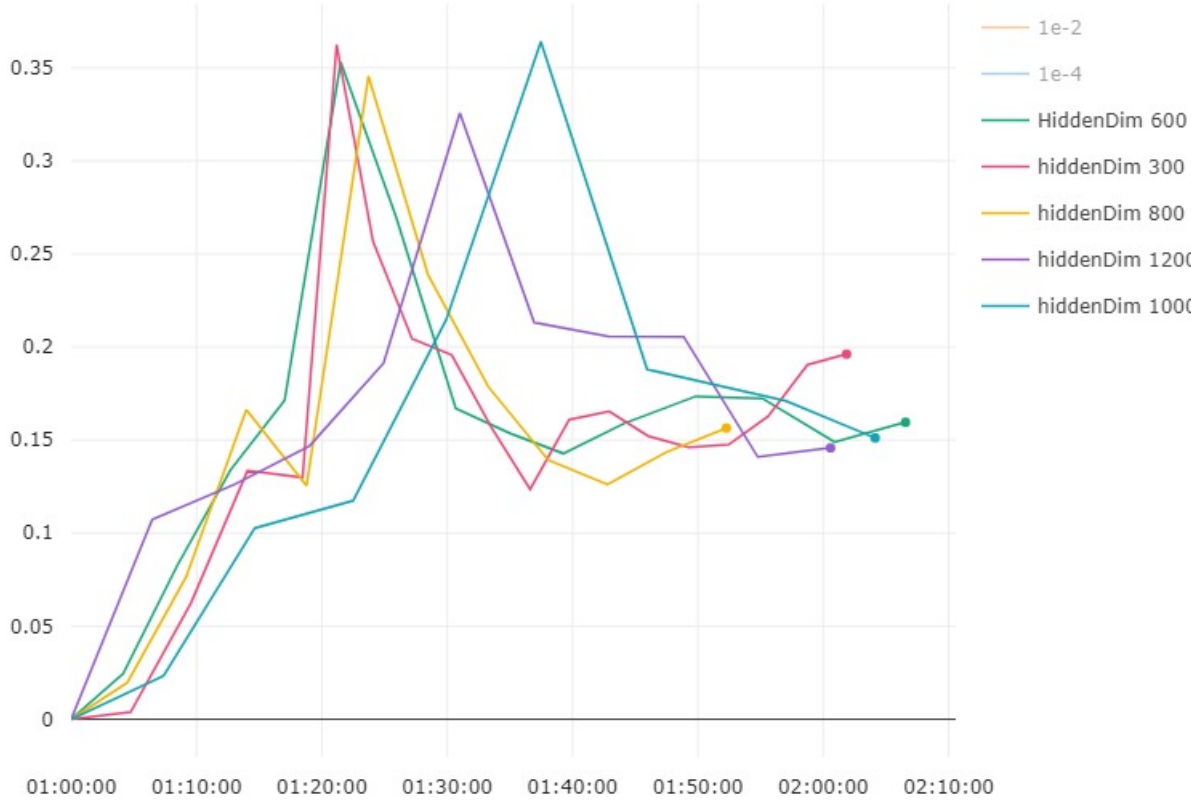


Figure D.2: Comparison of LSTM hidden layer dimensions
 xAxis: F1 yAxis: Training time

Classifier	Accuracy	Precision	Recall	F1-score
Unigram	92.54	18.18	2.97	5.10
Bigram - backoff	92.57	19.64	3.06	5.29
Trigram - backoff	92.56	19.53	3.06	5.29
Decision Tree	91.66	14.00	5.75	8.15
Bernoulli NB	91.91	21.11	15.11	17.61
Multinomial NB	92.00	19.40	10.84	13.91
SVM	92.71	35.27	7.88	12.88
GradientBoosting	92.63	32.11	6.49	10.79
LogisticRegression	92.53	32.72	6.42	10.79
BI-LSTM CRF pretrainedEmbed + charEmbed	90.13	13.23	13.13	13.18

Table D.16: Final Results

D.6 Results

The scikit-learn classifiers, the NLTK n-gram classifiers and the deep learning classifier were trained with the best found setups and the official wnut-17 evaluation script was executed on the results. Table D.16 shows the result of the evaluation. The Bernoulli Naive Bayes classifier did the best with a F1 score of 17.61. The proposed deep learning classifier only came in third with a F1 score of 13.18.

It turned out that one of the more simple classifiers performed best. This could be an indication as to the training dataset is too small for the more powerful models. A more detailed discussion of the results and conclusions can be found in chapter 7 of the memoria.

Appendix E

Technical Programming Documentation

E.1 Introduction

This section contains the technical programming documentation, describing the directory structure of the presented DVD and an installation guide.

E.2 Directory structure

E.3 Programmer's Manual

The presented DVD contains the following directories (the directory structure is split up into it's main directories since showing them all in one tree would get messy because of new pages):

```
| Documentation
|   |__ annex.pdf
|   |__ memoria.pdf
|__ requirements.txt
```

NLTK_SKLEARN

- `Classifiers` contains the trained classifiers
- `Data` contains the Data used for training
 - `wnut` contains the wnut challenge data
 - `emerging.dev.conll` validation data
 - `emerging.test.annotated` test data
 - `wnut17train.conll` train data
- `dataLoader` contains data loading modules
 - `reader` .. very slightly adapted ConllCorpusReader from NLTK (original does not support minus in entity name)
 - `api.py`
 - `reader.py`
 - `dataLoader` uses adapted ConllCorpusReader to read data
- `model` classifier related modules
 - `classifier.py` .implements ClassifierChunker used to build all classifiers
 - `features.py` contains feature extraction functions
- `results` contains gridtest/prediction results
 - `gridtest` contains gridsearch results
 - `prediction` . contains test data predictions of trained classifiers
- `utils` contains util fuctions
 - `readWrite.py` implements functions regarding reading/writing of data
 - `util.py` implements util functions most of them to transform data
- `buildData.py` .. script that reads, transforms and writes data into correct format
- `gridSearch.py`...script used for gridsearching with validation data
- `predict.py`script to predict input or test data
- `train.py`script to train classifier

```

pytorch
├── Data ..... contains the Data used for training
│   ├── embed ..... contains pre-trained embedding data
│   │   ├── glove.twitter.27B .. pre-trained twitter gloVe embeddings
│   │   └── glove.twitter.27B.200d.txt
│   ├── test ..... contains test sentences and labels
│   │   ├── labels.txt
│   │   └── sentences.txt
│   ├── train ..... contains train sentences and labels
│   │   ├── labels.txt
│   │   └── sentences.txt
│   ├── validation ..... contains validation sentences and labels
│   │   ├── labels.txt
│   │   └── sentences.txt
│   └── wnut ..... contains the wnut challenge data
│       ├── tags.txt ..... contains tagset
│       └── words.txt ..... contains all training words
├── dataLoader ..... contains data loading modules
│   └── dataLoader.py .. module that loads vocab/ char/ tag mappings
│       and generates batch data
├── experiments ..... contains logs of ran experiments
├── heroku-app ..... contains all modules relevant to web-app
├── logger
│   └── logger.py ..... implements logger class
├── model ..... contains all model specific modules
│   ├── layers ..... contains layers usable by Net
│   │   ├── charEmbed ..... implements character embedding layer
│   │   ├── embedLayer ..... implements word embedding layer
│   │   └── LSTM ..... implements LSTM layer
│   ├── eval.py ..... implements metric functions
│   ├── loss.py ..... implements loss functions
│   └── net.py ..... implements main neural network class
├── results ..... contains results of prediction
│   └── prediction
├── utils ..... contains utility modules
│   └── util.py ..... implements utility functions
├── buildData.py .script used to build the Data (create sentences and
│   labels files
└── buildVocab.py ..script used to build the vocab (create words and
    tags files

```

- `evaluate.py` implements evaluation function used in training
- `hyperParameterSearch.py` script that starts training jobs used for parameter search
- `predict.py` script used for predicting sample text or test data
- `train.py` script to train neural net

E.4 Installation

The project requires Python 3.6.1 or later. To install all dependencies simply install the `requirements.txt` file.

Appendix F

User Manual

F.1 Introduction

This user manual serves as a guide on how to use the provided scripts.

F.2 User requirements

The project requires Python 3.6.1 or later.

F.3 Installation

To install all dependencies simply install the requirements.txt file.

F.4 User manual

This user manual describes at how the different script can be run. Since the project is divided into two parts their scrips will be described separately.

NLTK

Data Preparation

This paragraph describes how the Data is transformed into the input format of the NLTK classifiers. These steps only have to be followed by the user if he wishes to introduce new data. Transformations for the wnut dataset

have already been done and are saved, do not have to be repeated.

To understand these steps first the structure of the provided WNUT data has to be examined.

WNUT 2017 Format

The provided wnut 17 Data is structured the following way:
Each line of data describes one token (word) and contains the word and its named entity tag in IOB tagging scheme delimited by a tab.

Example:

```
@Suzie55 O
whispering O
cause O
I O
may O
have O
had O
1 O
too O
many O
vodka B-product
's O
last O
night O
```

Data transformation The NLTK classifiers expect a different input format. The input format of the file for the NLTK classifiers contain a token per line. A line is made up by the word, its POS tag and named entity tag in IOB tagging scheme.

Example:

```
@Suzie55 JJ O
whispering NN O
cause NN O
I PRP O
may MD O
have VB O
had VBD O
1 CD O
```

too RB O
many JJ O
vodka NN B-product
's POS O
last JJ O
night NN O

The following steps are necessary to transform the data into the NLTK classifiers input format:

- Remove entity Tags: All entity tags have to be removed for tokenisation
- Tokenisation: Tokenise Datasets into sentences and then words
- POS Tagging: POS tag resulting words from previous step
- Join POS tagged results with previously removed entity tags

These steps are done with the `buildData` script in the NLTK_SKLEARN package. The script has the optional `--corpus` argument which specifies the path of the corpus to be loaded. The path has to contain a `train.conll`, `val.conll` and `test.conll` file in the wnut data format.

train

The `train` script located in NLTK_SKLEARN is used to train a selected classifier. Several optional arguments can be passed to the script, most of them specify the parameter of the classifiers. All arguments are optional. Usage: `python train.py --classifier <classifier> --corpus<corpuspath>`
Classifier options:

- `decisionTree`
- `NaiveBayes`
- `sklearnGradientBoostingClassifier`
- `sklearnRandomForestClassifier`
- `sklearnLogisticRegression`
- `sklearnDecisionTreeClassifier`

- sklearnSVC
- sklearnMultinomialNB
- 1-gram
- 2-gram
- 3-gram

For a more detailed view of all arguments and their functions call the script with the -h option.

predict

The `predict` script can be ran in two modes. Either in evaluation or prediction mode. The `--eval` parameter determines the mode. Eval mode means that the test dataset is loaded and the results are saved in the results/prediction folder. The test data is loaded from Data/wnut/test.conll Prediction mode allows the passing of a sentence that is to be classified. Optionally a specific pickled model can be selected to do the prediction. For that the pickle files name has to be passed to the model parameter.
Usage: python --model <name> --eval --sentence <sentence>

Pytorch

Data Preparation

The provided data has the same structure as previously described [here](#). The pytorch package contains two relevant data pre-processing scripts. One to build the vocabulary which is later used to map words and tokens to their respective indices in the vocabulary (`buildVocabulary`) and the other to tokenise the wnut data into sentences (`buildData`). Again these only have to be executed if a new dataset is introduced.

buildData

The `buildData` script reads each datasplit (train, test, validation) and divides them into two different text files. One containing all it's sentences (sentences.txt), each line containing one sentence, the other containing their associated labels. The files are saved in /Data/<dataset-name>. A optional corpus path can be specified via the `--corpus` argument. The folder has to contain a train.conll, test.coll and val.conll file.

Usage: python buildData.py `--corpus` <path>

Example (Representation of a sentence in their respective files):
 sentences.txt (on one line): @Suzie55 whispering cause I may have had 1 too many vodka 's last night and am a lil fragile , hold me ?

labels.txt: O O O O O O O O O O B-product O O O O O O O O O O O

buildVocab

The `buildVocab` script creates a vocabulary file in the /Data directory. Running the script saves one file for the words and one file for the labels containing one token per line. The script assumes the data to be in the Data/train/sentences.txt, Data/train/tags.txt files, meaning that the `buildData` script has to be ran before.

train

The `train` script is used to train the deep learning model. It has a single argument specifying the path to the folder to the json file that defines the models hyperparameters. The file has to be named "params.json".

Usage: `python train.py --paramsDir <path>` Listing F.1 shows all possible parameters.

Listing F.1: params.json: defining hyperparameters of the model

```
{
    "learning_rate": 1e-3,
    "batch_size": 32,
    "num_epochs": 100,
    "lstm_hidden_dim": 300,
    "embedding_dim": 200,
    "char_lstm_dim": 200,
    "char_embedding_dim": 100,
    "save_summary_steps": 100,
    "dropout": True,
    "bidirectional": "True",
    "crf": "True"
}
```

predict

The `predict` script can again be ran in two modes. Either in evaluation or prediction mode. The `--eval` parameter determines the mode. Eval mode means that the test dataset is loaded and the results are saved in the results/prediction folder. The test data is loaded from Data/wnut/test.conll. Prediction mode allows the passing of a text that is to be classified. Optionally a specific model can be selected to do the prediction. For that the location folder that contains of the state dictionary (.pth.tar) and parameter (params.json) file has to be passed.

Usage: `python --paramsDir <path> --eval --text <text>`

Output Format

The evaluation output format of all classifiers is the same for easy use of results. Each line describes one token, containing it's word, gold-entity and predicted entity in IOB tagging scheme. (token gold-label predicted-label). Sentences are divided by an empty line. Example: @Suzie55 O O
whispering O O

cause O O
I O O
may O O
have O O
had O O
1 O O
too O O
many O O
vodka B-product B-product
's O O
last O O
night O O

wnuteval The official `wnuteval` script located in the base package can be ran with the previously discussed output file format. The script calculates accuracy, recall, precision and F1 score of the predictions.

Usage: `python wnuteval.py <filename>`

Demo web-app

The usage of the web- is straightforward. Simply introduce a text in the text field, choose an output option (text for only textual output, Image for a coloured graphic) and press the CLASSIFY button. The classification will then be created and outputted. How the HTML interface of the application looks like can be seen in figure [F.1](#).

Noisy-NER

☒ Text

☐ Image

CLEAR

CLASSIFY

Text: Facebook released statements

Entities: B-corporation O O

Facebook CORPORATION released statements

Named entity recognition using a BiLSTM CRF Model




Figure F.1: Web-application

Bibliography

- [1] Victor Huang. Pytorch template project. URL: <https://github.com/victoresque/pytorch-template>.
- [2] sklearn documentation - sklearn.ensemble.gradientboostingclassifier. Accessed: 2019-04-23. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html>.
- [3] Srivastava Hinton, Sutskever Krizhevsky, and Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. Technical report, University of Toronto. URL: <https://arxiv.org/pdf/1207.0580.pdf#>.