ReAdvisor

June 26, 2021

Chapter 1

Sentiment Analysis and Aspect Recognition

1.1 Overview

The Sentiment Analysis component of the system detects the tonality in the reviews towards specific aspect. Namely, different aspects in the domain of restaurant reviews (RESTAURANT) and hotel reviews (HOTEL).

The system is built to accept textual input (a review) and a domain (RESTAURANT or HOTEL). This input is then analysed to produce predictions on the sentiment of different aspects. The sentiment prediction is given via a scoring between 1 (very negative) and 5 (very positive) Notably, predictions must be made in a fine-grained manner (sub-aspects) and must also allow us to pinpoint the exact location of the predicted signal in the text itself. The languages covered are German and English.

The output of the system is thus an annotation layer which depicts which parts of the text consider which aspects (and sub-aspects) and what kind of sentiment (POSITIVE/NEGATIVE and a rating between 1-5) is conveyed. This system is trained on preannotated data that consists of textual spans that have been manually labelled by re:spondelligent.[1]

After a first intermediate system has been implemented, we bootstrapped the output of that system to further enhance the annotation workflow and efficiently create sufficient annotated data to train the delivered system.

One of the most important points for this project is that the system is trainable for further improvement in the future. Additionally, a crucial design requirement is that system may be extended for other domains and/or languages in the future. Hence, we also describe the workflow for the annotation step briefly which is done with the software prodigy[2] which was acquired for this purpose. This is especially notable because this software is narrowly intertwined with spacy[3], an NLP framework that allows for robust and efficient

[1] Note that the currently available data is not in the form of annotated texts but only as a "collection of text spans"; hence it is not possible to train a sequence labeller, although it will be more feasible in the future to do so if the annotated data is available as fully annotated texts.

[2] https://prodi.gy/

[3] https://spacy.io/

data-processing on an industrial scale. However, there are no basic constraints that prohibit adopting other technical solutions in the future without losing any of the annotated data acquired in this project. This means that the close connection between the annotation framework and the processing machinery leads to a facilitated access to this kind of language processing software and techniques, while the flexibility for future improvement is preserved.

1.1.1 Requirements

The following requirements have been formulated for the project and are addressed accordingly:

- Language coverage: German, English
- Domain coverage: RESTAURANT, HOTEL
- Strategy for further improvement
- Strategy for further adaptations

1.2 Data Preparation

Since the system is designed in a modular fashion, there are no strict specifications for the single components. Following such a path avoids the lock-in effect for data collection as well as for the modelling—or the implemented solution—itself.

However, we will describe here the data organisation and the applied methods that are the foundations of the delivered system.

- Since the overall system requires an output from the aspect as well for the sentiment-based information on different levels (token-level, i.e., text spans as well as aggregated document vectors), the system is trained on the level of identifiable text spans which are predicted by the solution and further assembled to document-level information.
- This means that for each aspect and each sentiment category we collect annotated examples.
- Hence, the adaptation to other domains in the future is straight-forward and analogous to the given setting.
- The aspect classification is based on a hierarchical classification system which first predicts the top-level category and subsequently further distinguishes the sub-aspects.
- This is also due to the large variance of the available data (i.e., class imbalance) between a) the sub-aspect categories and b) even on the top-level aspects.
- For the sentiment system, we additionally implement a scorer component to rate the negativity/positivity of the single spans.

- One of the advantages of the system is that we use a cross-lingual (distilled) transformer-based embeddings, which allows us to train jointly on English and German data. Not only do we achieve a good level of robustness —which is needed for user-generated web content— through the form-agnostic transformer, we also leverage the data to make the same kind of predictions for unseen languages. This means that the predictors are capable of performing the same task on French, Spanish, and so on. Note: of course, the performance for unseen languages will improve if examples from this language are integrated in the training corpus.

1.3 Models

We have trained the following models:

- A sequence labeller (SLICER) that detects relevant information from given reviews (language-specific; cross-domain)
- Two top-level aspect classifiers (domain-specific; cross-lingual) that predict the top-level aspect
- Per top-level aspect, one classifier for the sub-level aspect (domain-aspect-specific; cross-lingual)
- A sentiment classifier per domain (cross-lingual) to predict the sentiment of the given text spans
- A scorer component that scores the sentiment of the given span

1.3.1 SLICER

In order to pool the available data, we frame the task of accurate recognition of the informative text spans as "slicing" (hence the name). This means, that we only discern important parts of the texts and non-informative parts. The advantage of such a perspective is that we can pool the data from all aspects and only train for a binary distinction. Of course, we will then need a component for the fine-grained classification and sentiment rating for the identified information bearing text spans.

1.3.2 Top-level Aspect Classifier

Since the classification scheme is hierarchically based, we reflect this fact in the architecture. This means that we classify in a first step on the top-level and, as a second step, do the fine-grained per-aspect classification according to the given top-level. This makes sure that we do not have a mismatch between the top-level aspect and the sub-aspect.

1.3.3 Sub-Aspects Classifier

As mentioned above, the classification of the sub-level aspects is a downstream task after the top-level classification. Hence, we only need to discern between the sub-aspects of the respective top-level aspect, which also helps to counteract the stark skew in distribution over the classes.

1.3.4 Sentiment

The component that predicts the sentiment is implemented in a way so that it produces a score between 1 and 5. It is trained on the given annotations for the text spans and is hence not a sequence labeller, but a classifier that sits on top of the sentence-transformer representation. Since the value 3 was given for textual spans that are slightly negative but also for such ones that were "ok-ish", we applied a two-phase classifier instead of a regressor.[4] The classifier first makes a three-partite decision: *negative, mid, positive*. If the prediction is *negative* or *positive*, a fine-grained distinctor is applied that decides if the final value is 1 or 2, or the final value is 4 or 5. If the *mid* class is predicted by the three-partite classifier, we counteract the mixed-data-in-one-class in the sense of a post-correction step that helps to better distinct between a *negative* rating and the *mid* class.

1.3.5 Sentiment Document Scorer

This component is trained against the ratings available in the data, i.e., the ratings that were scraped from the internet. As feature vector, we leverage the document vector (that is also used for the difficulty scoring component). We train the regression for each language-domain pair, to allow for a better level of adaptation. Additionally, the values are clipped at the values of 1 and 5—which means that higher scores will result in exactly 5 and lower results than 1 will be replaced with a scoring of 1. The according jupyter notebooks also provide further insights into the evaluation of the component.

1.4 Training

1.4.1 General Description

We use a cross-lingual (distilled) sentence transformer[5] for the featurization/embedding on top of which we trained small neural networks with a hidden layer (in specific cases with an adapted layer size). This leaves room for improvement to use another classifier or directly a transformer-based pretrained model (e.g. a BERT model), once sufficient amounts of data are available.

1.4.2 Training Details

SLICER Component

The SLICER component is meant to detect information in the reviews that is important for re:spondelligent (hence the name given internally to the identified text spans: "RESPO_INFO"). Since it must detect the information as precisely as possible and also allow for multiple different slices in the same sentence, we have implemented it as a sequence labeller.

For this, we use prodigy's/spacy's NER component (which is also a sequence labeller) - although we are not looking for named entities in the text. After bootstrapping the intermediary system, we assembled sufficient

[4] We evaluated also a regressor for the task but especially for values between 2 and 4 (including 3), the performance was not convincing. Accordingly, we applied a tweak to counteract the mixed-data-in-one-class problem. Since the classifier was not able to properly model the value-3 class, we trained an extra component, that is able to discern between "negative-value-3" cases and "ok-ish-value-3" cases. If negativity is detected, we lower the final score to value 2.

[5] https://www.sbert.net/ and https://github.com/UKPLab/sentence-transformers

training material for an initial replacement of the intermediary system and are able to use the deployed system to a) make predictions on the incoming new data and b) replace it with an improved version. This iterative process is also called bootstrapping which allows to systematically improve and adapt the system.

Furthermore, since the available training data (roughly 1,500 reviews for each language) is still rather small, we leverage the data that is not yet annotated and has been assembled over the years at re:spondelligent. We do this with *pretraining*[6] which trains one of the basic layers of the system "to adapt" to the given domain of texts. This kind of transfer learning leads to an improvement of approximately 0.05 to 0.1 F-Score.

To pretrain the tok2vec layer for the spacy-based system, we can simply call the relevant command, e.g.:

```
python -m spacy pretrain -uv --n-iter 200  de_texts.jsonl de_core_news_md
↪  SPACY2_PRETRAIN_MD_DE
```

where

- `de_texts.jsonl` is a JSONL file that contains the texts to pretrain on
- `de_core_news_md` is a pretrained model (downloadable for spacy)
- `SPACY2_PRETRAIN_MD_DE` is a target folder for the pretrained layer (weights)

We can then use the pretrained tok2vec layer to train the slicer with the prodigy command, e.g.:

```
prodigy train -t2v SPACY2_PRETRAIN_MD_DE/model125.bin -o SLICERMODEL -n 40 ner
↪  de_slicer_r1,de_slicer_r2  de_core_news_md
```

where

- `SPACY2_PRETRAIN_MD_DE/model125.bin` is the pretrained layer
- `SLICERMODEL` is the target folder where the trained model will be saved
- `ner` specifies the training procedure and is aligned with the way the data was annotated with prodigy, hence specifying that
- `de_slicer_r1,de_slicer_r2` are the two data sets from the prodigy database that are used for training
- `de_core_news_md` is the base model from spacy which we add (also for the static vectors). NB: this must be in accordance with the model that was used during pretraining.

This training procedure results in an iterative output which looks like the following:

```
✓ Loaded model 'de_core_news_md'
Created and merged data for 1467 total examples
Using 1174 train / 293 eval (split 20%)
Component: ner | Batch size: compounding | Dropout: 0.2 | Iterations: 40
✓ Initializing with tok2vec weights
SPACY2_PRETRAIN_MD_DE/model125.bin
```

[6] see https://spacy.io/usage/embeddings-transformers#pretraining

```
 Baseline accuracy: 0.000

============================   Training the model ============================

#    Loss       Precision   Recall     F-Score
--   --------   ---------   --------   --------
 1   28283.53      63.195     49.446     55.482
 2   26320.71      65.595     54.905     59.776
 3   25684.05      66.996     58.940     62.710
 ...
12   20572.88      67.456     63.133     65.223
13   20229.03      67.481     63.370     65.361
14   19954.90      67.311     63.370     65.281
15   19927.83      67.734     63.608     65.606
16   19516.26      68.094     64.161     66.069
17   19161.09      68.487     64.478     66.422
18   19066.67      68.425     64.636     66.477
19   18613.12      67.977     64.320     66.098
20   18386.78      67.419     63.845     65.583
....
38   16135.23      65.620     62.816     64.188
39   15945.93      65.267     62.737     63.977
40   15746.04      65.021     62.500     63.735

============================   Results summary ============================

Label        Precision   Recall   F-Score
----------   ---------   ------   -------
RESPO_INFO      68.425   64.636    66.477


Best F-Score   66.477
Baseline       0.000

✓ Saved model: /home/ma/prodigy_work/FINAL_DE/SLICERMODEL
```

We can easily see that the the component keeps improving across several epochs (internally, spacy uses a neural network architecture for its components) until it starts overfitting and hence the reported scores on the held-out test set start to decrease again. Training via prodigy has the additional benefit that we receive the characteristics of the the learning process of the component (which can further be inspected with `train-curve`[7]).

[7] see https://prodi.gy/docs/recipes#train-curve

Aspect Component

The training of the aspect component is delivered in the form of `jupyter-notebooks`. There are several reasons for this:

- The process of the data preprocessing and the involved software can be described in the notebook itself
- This helps to avoid divergence between documentation and code/components/data
- The notebook allows to inspect the computed models either on a quantitative basis with an evaluation matrix (different scores; for all classes, etc.) and offer substantially important details in the confusion matrix
- Also the interactive testing of newly derived models is possible

However, if the process is to be further automated and maybe processed as an overnight batch-job, the notebooks can either serve as a basis from which the respective code can be copy-pasted—or even more simply, the whole notebook can just be turned into a python script using `nbconvert` (e.g. `jupyter nbconvert -to script my_train_notebook.ipynb`)

Sentiment Component

The training of the aspect component is also delivered in the form of `jupyter-notebooks`. See above for reasoning.

1.5 Annotation Workflow

As mentioned above, prodigy is the annotation tool that we use for various purposes in this project. For the SLICER component, we can start up an annotation environment on a server on which prodigy was installed like this:

```
prodigy ner.correct ner_correct_de SLICERMODEL texts_to_annotate.jsonl --label RESPO_INFO
```

where

- `ner.correct` is the prodigy recipe that is used[8]
- `SLICERMODEL` is the folder where the already trained model is located (which then pre-annotates the reviews)
- `texts_to_annotate.jsonl` is a JSONL file that contains the texts to annotate
- `RESPO_INFO` is the label (can also be a list of labels) which we want to annotate for

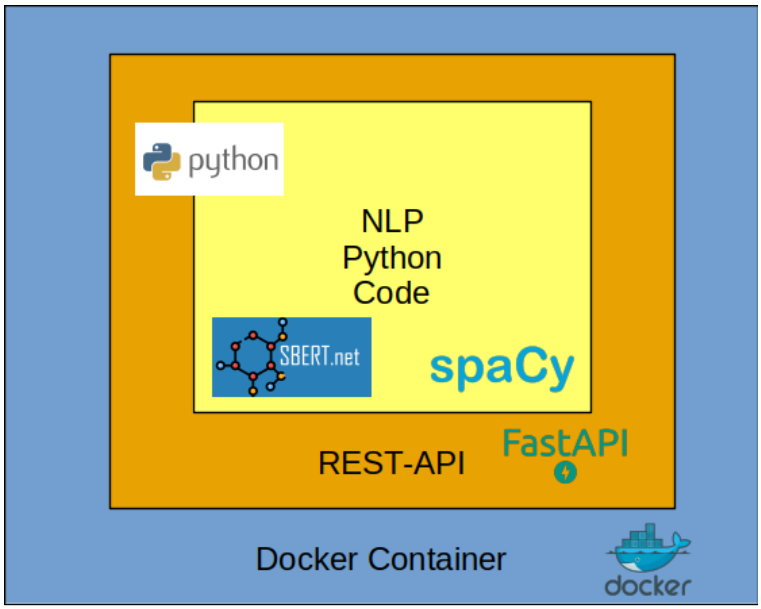

Figure 1.1: The basic architectural integration of the component: the functional NLP code is run as an REST-API which is included in a docker container to which requests from the outside are sent. Note that the models (SLICER, ASPECT, SENTIMENT) for the components are not visible in this schematic overview; if to include, they would sit in the yellow box

1.6 Inference

For inference, we need to package the components so that they can be used via a REST-API. For this project, we will stick to the following architectural blueprint

The code is self-contained and basically loads the classifiers and the slicer model. It contains a wrapper function which can be conveniently called from the API layer to abstract the underlying complexity.

The API layer itself is based on the elegant and efficient FAST-API package for python. One of the most beneficial features of this API software component is also the self-documentation (e.g. for the used example below, the documentation is available on http://docker.respondelligent.com:9500/docs). This documentation offers a full formal and exemplified description of the interface itself (i.e. which parameters are to specify and how) and hence should also be used as the point of reference for further documentation.

The required code for the API layer is embedded in the functional code.

The API may be started through the call of an apt server like `uvicorn`. However, in order to automatically start up the component, we decided to include this step in the Docker-layer. That is, in the `Dockerfile` provided for the full component, we define the relevant entry point, e.g.:

```
ENTRYPOINT uvicorn de_alpha_with_doc_vec:app --workers 1 --host 0.0.0.0 --port 8000
```

The Docker image can be built with the following command:

```
docker build . -t sentimentsaspects-de
```

Once built, the a docker container can be launched like this:

[8] see https://prodi.gy/docs/recipes#ner-correct

```
docker run --name de-sa -p 9500:8000 sentimentsaspects-de:latest
```

Note: this includes a port mapping from within the docker (8000) to the outer system (which routes 9500 to the docker)

1.7 Evaluation

1.7.1 Sentiment

Since the sentiment component is expected to produce discrete values between 1 and 5, naturally a regression component would be the suitable choice. However, the demand for an adapted solution turned up: on the one hand, to counteract inconsistencies within the levels of sentiment, and on the other hand, to tackle the challenge of the heavily skewed data distributions.

In order to do so, we apply an adapted approach as follows. First, we frame the task as a 3-partite classification (POSITIVE, MID, NEGATIVE). For this task, we trained a component that reaches an accuracy of 0.91.[9]

To receive a fine-grained score (4 or 5) for the POSITIVE predictions, we apply for each label a different classifier that decides between the scores 4 and 5 (for POSITIVE predictions) and for the scores 1 and 2 (for the NEGATIVE predictions.)

In the case where we have a MID prediction (for category 3), we apply a last tweak which helps to tackle the misclassifications between the NEGATIVE examples and the MID ones. Since the MID category contains examples that are justifiably in the middle of the range (e.g., "food was ok"), there are also lots of examples that contain rather negative evaluations of certain aspects. Hence we allow a *justified stealing* (from the perspective of the NEGATIVE examples) when we get a MID prediction, but if an additional component (that only distinct between MID and NEGATIVE cases) yields a NEGATIVE prediction. This means, that we correct the original decision MID towards NEGATIVE (score 2) if we have the appropriate pattern of classifications from the respective components.

This customized setup allows to counteract the challenges that arise from problems which we mentioned above.

1.7.2 Aspects

In this section, we give some evaluation numbers for the aspect classifier and sub-aspect classifiers.

RESTAURANT Domain

We can clearly see that the sentence embeddings (sentenceBERT) based classifiers outperform the simple baseline (DummyClassifier—which relies on the majority vote) and also quite clearly the TF-IDF based classifiers that we use as a measurement to estimate how much the gain is of the more sophisticated, transformer-based classifier.

The lower scores in the FOOD category are related to the 12-partite classification scheme and the available data that displays a heavy data distribution skew. Additionally, the annotation guidelines may not provide

[9]Note that a binary classification into POSITIVE and NEGATIVE categories, which are based on the data available for the scores 1 and 2, as well as 4 and 5, respectively, resulted in an even higher accuracy above 0.95.

Aspect Classification Top-Level for Restaurant	
Classifier	Accuracy
DummyClassifier	0.274
TF-IDF (+LR)	0.885
SentTransformer (+ LR tuned)	0.944
SentTransformer + MLP	**0.959**

Figure 1.2: The evaluation of the **top-level** aspect classification for the RESTAURANT domain. We give the Accuracy of the classifiers; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

Sub-Aspect Classification for Restaurant					
Classifier	AMBIANCE	FACILITIES	FOOD	SERVICE	VALUE
DummyClassifier	0.403	0.258	0.201	0.248	0.948
TF-IDF (+LR)	0.833	0.821	0.667	0.671	0.974
SentTransformer + MLP	**0.939**	**0.973**	**0.769**	**0.804**	**1.000**

Figure 1.3: The evaluation of the **sub-aspect** classification for the RESTAURANT domain. We give the Accuracy of the classifiers; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

rigorously exclusive distinctions of the categories which may also contribute to the misclassifications (e.g. between FOOD-GENERAL and FOOD-QUALITY_TASTE).

We also provide the confusion matrices in order to get a quick glimpse at the per class performance. Note that a more in-depth version of the evaluation is available in the jupyter notebooks and can be updated automatically when more training data is available, or a new language or domain should be added.

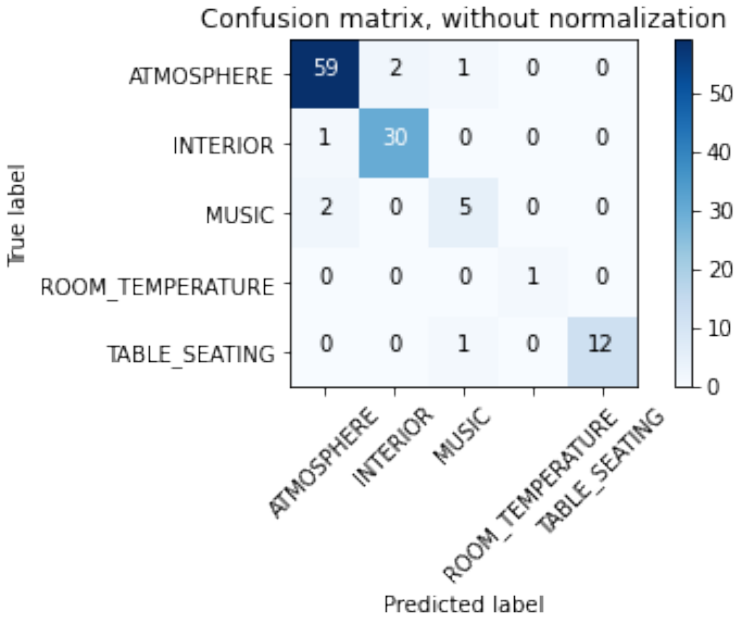

Figure 1.4: Confusion matrix for the evaluation of the **sub-aspect** AMBIANCE in the domain RESTAURANT; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

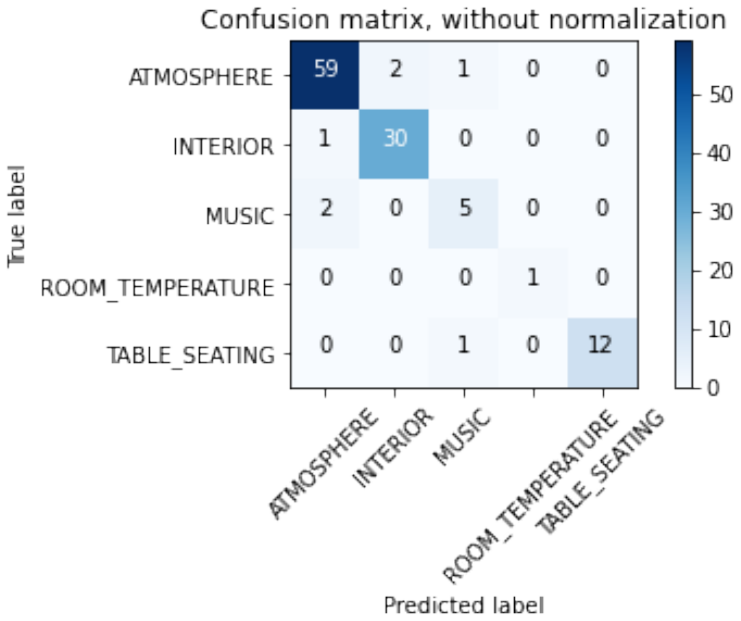

Figure 1.5: Confusion matrix for the evaluation of the **sub-aspect** FACILITIES in the domain RESTAURANT; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

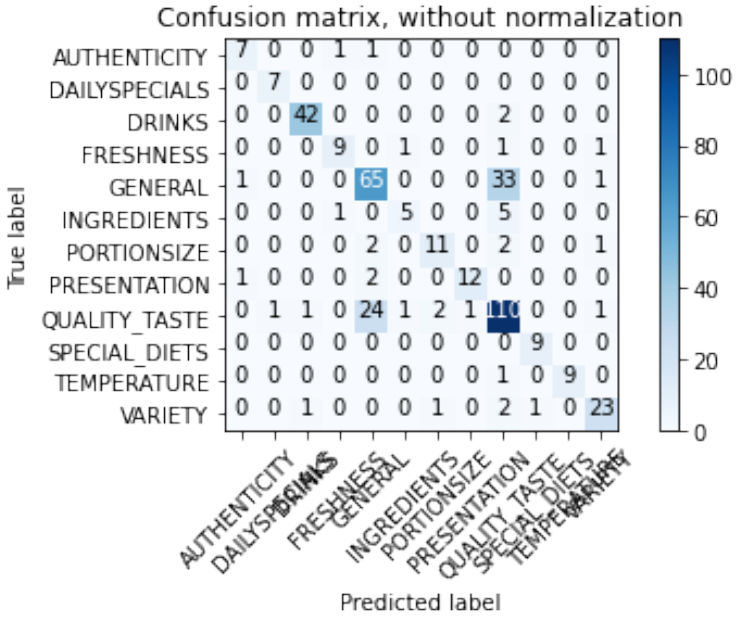

Figure 1.6: Confusion matrix for the evaluation of the **sub-aspect** FOOD in the domain RESTAURANT; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

HOTEL Domain

In the HOTEL domain, we see clearly how the sentenceBERT based classifier outperform the other baselines, often by a large margin. But also here, the performance is dependent on the cardinality of the sub-aspects

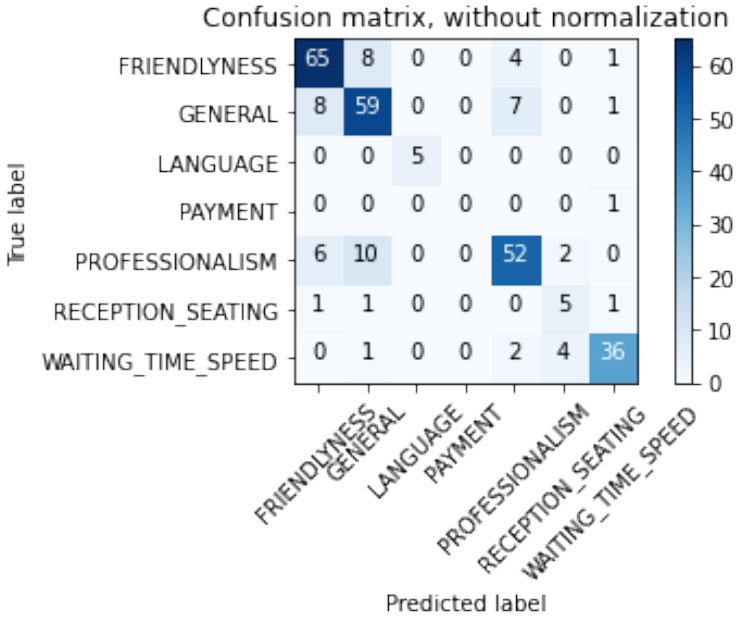

Figure 1.7: Confusion matrix for the evaluation of the **sub-aspect** SERVICE in the domain RESTAURANT; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

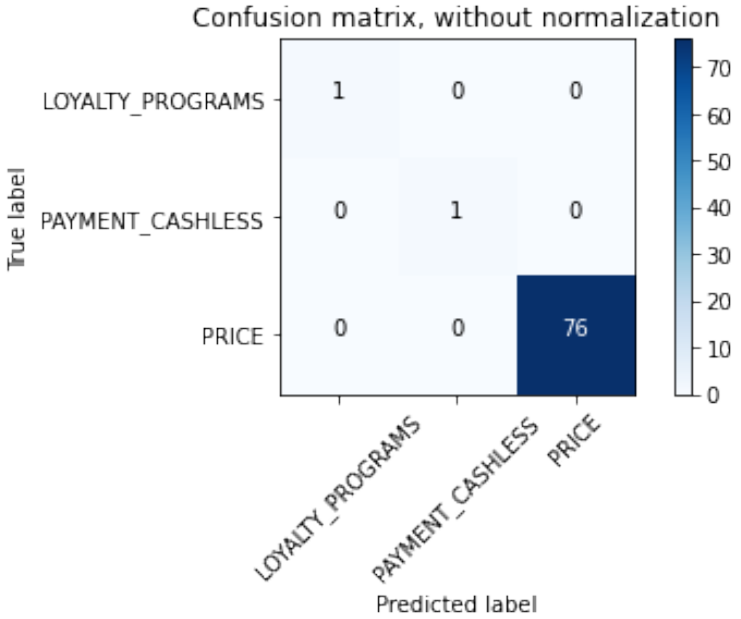

Figure 1.8: Confusion matrix for the evaluation of the **sub-aspect** VALUE in the domain RESTAURANT; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

and of the distinctiveness of the categorization scheme.

Aspect Classification Top-Level for Hotel	
Classifier	Accuracy
DummyClassifier	0.223
TF-IDF (+LR)	0.861
SentTransformer (+ LR tuned)	0.925
SentTransformer + MLP	**0.939**

Figure 1.9: The evaluation of the **top-level** aspect classification for the HOTEL domain. We give the Accuracy of the classifiers; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

Sub-Aspect Classification for Hotel						
Classifier	FACILITIES	FOOD	LOCATION	ROOM	SERVICE	VALUE
DummyClassifier	0.197	0.529	0.331	0.097	0.359	0.874
TF-IDF (+LR)	0.700	0.915	0.782	0.671	0.828	0.929
SentTransformer + MLP	**1.000**	**0.983**	**0.936**	**0.924**	**0.845**	**1.000**

Figure 1.10: The evaluation of the **sub-aspect** classification for the HOTEL domain. We give the Accuracy of the classifiers; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

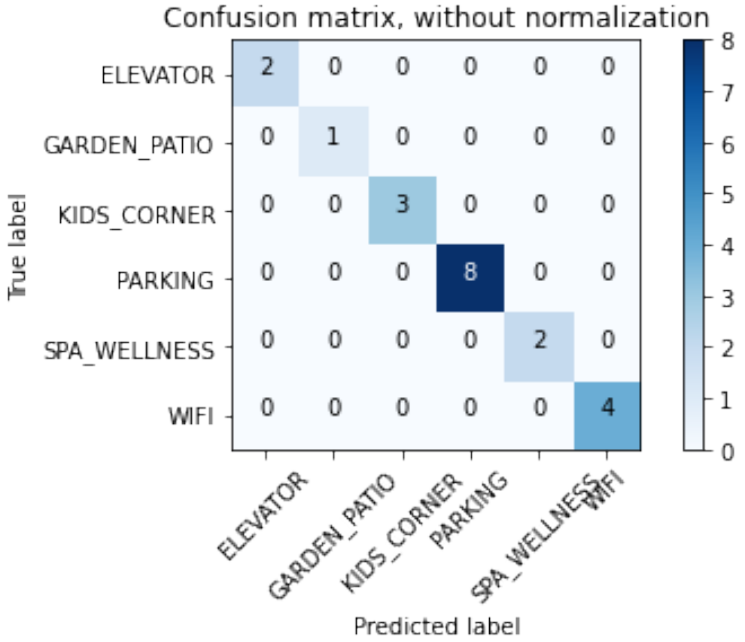

Figure 1.11: Confusion matrix for the evaluation of the **sub-aspect** FACILITIES in the domain HOTEL; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

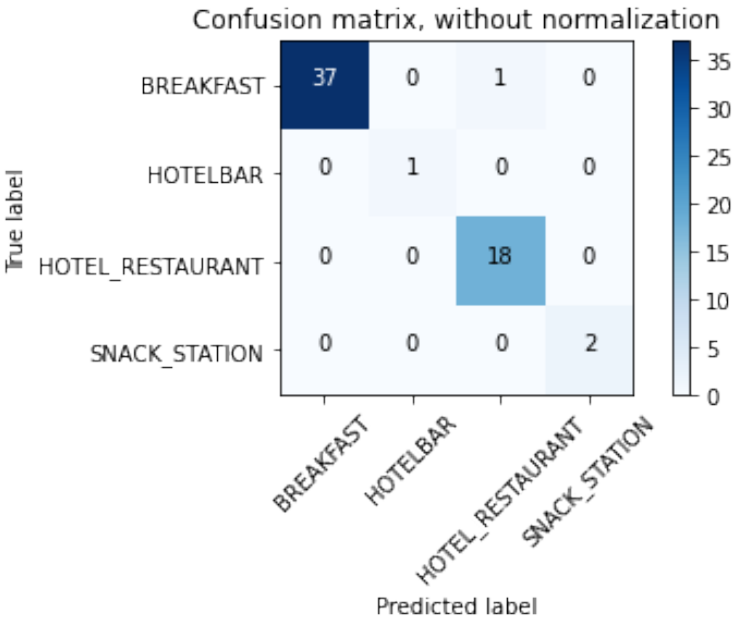

Figure 1.12: Confusion matrix for the evaluation of the **sub-aspect** FOOD in the domain HOTEL; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

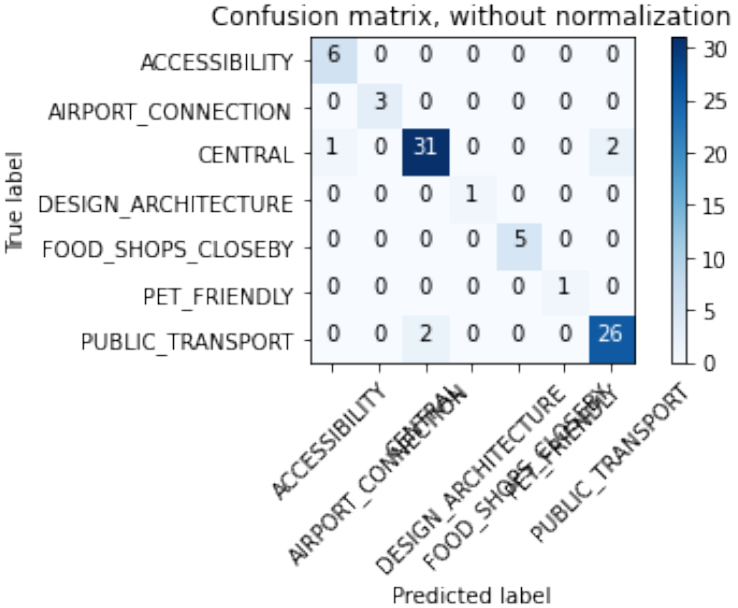

Figure 1.13: Confusion matrix for the evaluation of the **sub-aspect** LOCATION in the domain HOTEL; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

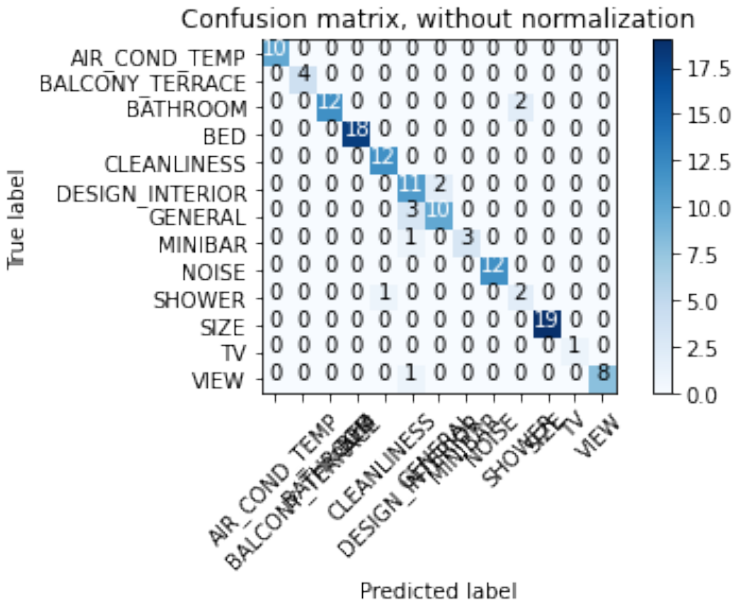

Figure 1.14: Confusion matrix for the evaluation of the **sub-aspect** ROOM in the domain HOTEL; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

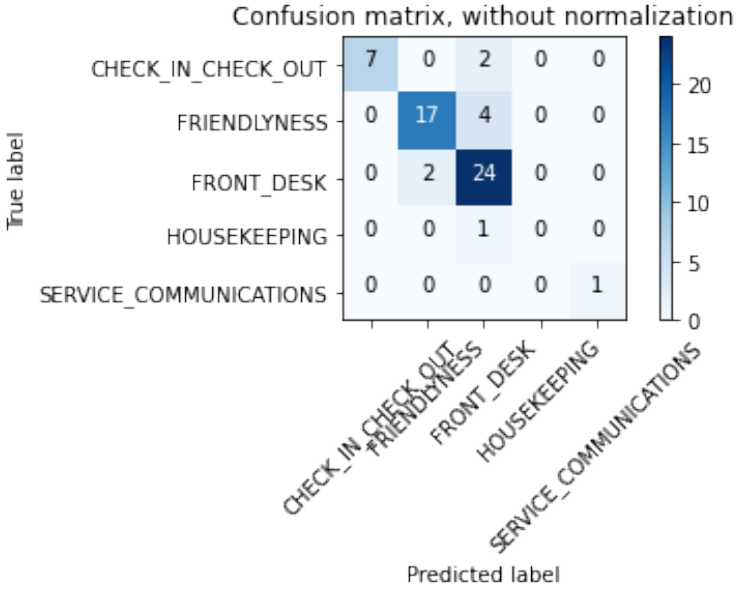

Figure 1.15: Confusion matrix for the evaluation of the **sub-aspect** SERVICE in the domain HOTEL; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

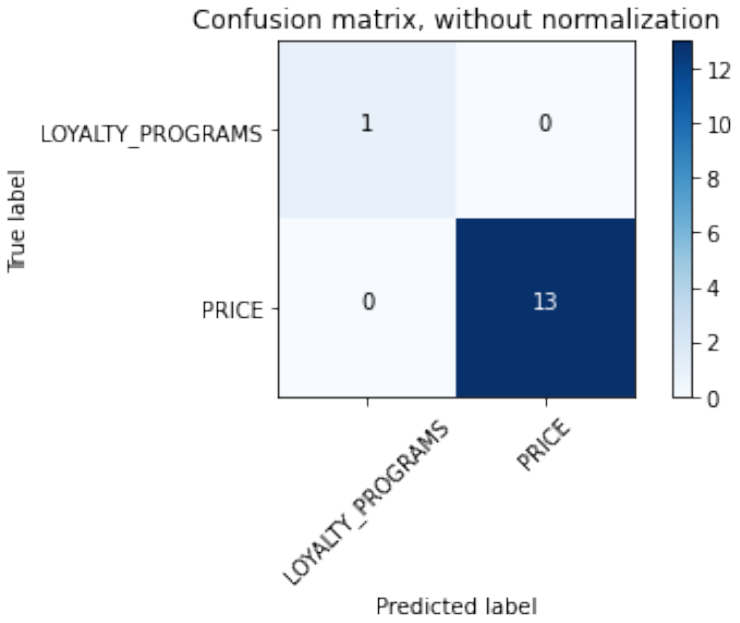

Figure 1.16: Confusion matrix for the evaluation of the **sub-aspect** VALUE in the domain HOTEL; more fine-grained and class-specific evaluation can be found in the relevant jupyter notebook.

Chapter 2

Response Generation

2.1 Overview

Our response generation models are based on the sequence-to-sequence (seq2seq) encoder-decoder paradigm commonly used in machine translation (MT). This approach provides an effective method of generating a suitable-length target text conditioned on a given input source text. We consider a customer review as the source text and a response as the target text. Therefore, model training relies on a parallel corpus of review-response pairs.

Here, we describe the response generation models developed as part of the ReAdvisor project and the steps involved in preparing the data, training and evaluating these models.

2.1.1 Seq2seq models

Seq2seq models allow for mapping arbitrarily long source sequences to arbitrarily long target sequences and are commonly used in NLP tasks like MT, abstractive summarization and dialog systems. The basic idea is as follows: a source text is first *encoded* into a dense vector representation using a neural network (e.g. Recurrent Neural Network (RNN), Transformer encoder, etc.). The resulting representation is then fed to another neural network known as the decoder. The decoder's job is to *decode* this dense vector into a sequence of words. The decoder is trained to predict output tokens auto-regressively over a series of timesteps. In this way, each prediction is conditioned on the encoded input sequence and the previous output tokens. The decoder stops generating once the model predicts a special end-of-sequence token or reaches the maximum allowable sequence length that the model is trained on.

2.1.2 Response generation model

The final model exploits recent developments in large-scale pretrained seq2seq models (e.g. BART[1]).

In order to be able to generate responses for both language pairs we use the multilingual version, mBART[2]. The off-the-shelf mBART model requires a GPU with at least 32GB of RAM, which makes fine-tuning challenging. However, most of the memory is consumed simply by the large embedding matrix that gets

[1] https://arxiv.org/abs/1910.13461
[2] https://arxiv.org/abs/2001.08210

loaded for the 250K-item vocabulary. Trimming this embedding matrix before fine-tuning reduces the memory demands considerably and allows us to fine-tune mBART on a regular 12GB GPU[3].

We define task-specific tokens and provide these to the model to explicitly incorporate review meta information. Specifically, we prefix review texts with a special token indicating the domain of the review and another to indicate the user's rating. Both of the labels are intended to aid the model when encoding ambiguous review texts and ensure that the generated output text is conditioned appropriately.

In this chapter we provide details about how to prepare the data, fine-tune and run inference with a customized mBART model. Finally, we provide details on how to evaluate model performance provide ideas for future developments.

2.1.3 Requirements

As with the parallel task of sentiment analysis, the following requirements have been formulated for the response generation task:

- Language coverage: German, English
- Domain coverage: RESTAURANT, HOTEL
- Evaluation of overall performance
- Strategies for further improvement
- Strategies for further adaptations

In order to run the scripts described in this chapter, it is best to setup a clean python environment (e.g. using conda). The script repository contains a `requirements.txt` file which lists all the packages and versions required.

```
conda create --name respondelligent python=3.8.5

conda activate respondelligent

conda install cudatoolkit=your-cuda-version # only required if training a new model
↪  on a GPU instance

pip install -r requirements.txt
```

2.2 Data Preparation

Since our response generation models rely on subword-segmented inputs, data preprocessing is minimal. Below we describe the main steps involved.

[3]For our experiments, we use a single NVIDIA GeForce GTX TITAN X with 12GB memory

2.2.1 Text cleaning

The data from re:spondelligent was provided as a SQL database dump. We apply regular cleaning and normalisation techniques. These include:

- normalize all whitespace characters by removing line breaks and tab space characters
- remove any html markup
- remove any automatic translations from web-based content (e.g. 'Translated by Google')
- de-duplicate based on lowercased string matches with no whitespace characters
- identify the language of the review and response texts with `langdetect`[4]

Assuming well-formed CSV files[5], the first step involves applying a single cleaning script that wraps the necessary functions. For example:

```
python clean_respondelligent_data_from_extracted_csv.py \
    directory-containing-db-csv-files \
    output-pickled-dataframe
```

The cleaned outputs are then saved to a `Pandas` Dataframe.

2.2.2 Mask greetings and salutations

Response texts typically adhere to a formulaic structure, e.g. greeting, body, salutation. Since we want the generation models to focus solely on producing the body of a response, we replace greetings and salutations with masking tokens (e.g. `<GREETING>`, `<SALUTATION>`). Customer-specific greetings and salutations can be inserted in a generated response text using simple gap-filling techniques.

We trained a custom sequence-labelling model from `Flair`[6] to reliably identify a wide range of expressions used for greetings and salutations in both English and German response texts. The model was trained on approximately 1000 labeled texts (500 en / 500 de).[7]

For processing a small amount of data (e.g. only re:spondelligent data), this step is integrated in the Ipython notebook `generate_model_files_for_mBART.ipynb` (see the description provided in Section 2.2.3).

2.2.3 Generate Model Training Input Files

Input files required for model training are line-aligned source and target files. Each line in the source file contains a review text and the corresponding line in the target file contains its response. To generate appropriate training files from a `Pandas` dataframe, we simply extract the columns of interest and write them

[4] https://pypi.org/project/langdetect/

[5] We had issues exporting well-formed CSV files directly from the DB dump. The easiest solution was to export as JSON and convert the JSON exports to CSV with Pandas. Use the script `merge_rechannel_and_sf_guard_group.py` to convert DB tables `rechannel` and `sf_guard_group` to a single CSV file and `convert_json_exports_to_well_formed_csv.py` to convert `reviews` and `reviewanswers`.

[6] https://github.com/flairNLP/flair

[7] Note: the model was originally trained on tokenized texts that had been processed by `spaCy`, however, we observed that the model still works well on non-tokenized text that is naïvely split on whitespace.

to their own regular files separated into training, test and validation splits. In this step, we also make any final modifications to the input sequences which aids reproducibility. For example:

- remove greetings and salutations from response texts
- convert categorical information to 'labels', for example;
  - replace ISO language tags with mBART language tags;
  - wrap domain and rating values with '<' and '>'.
- check for duplicates between train, test and validation splits.
- shuffle training set

We provide this script in the form of an Ipython Notebook `generate_model_files_for_mBART.ipynb`.

2.3 Model Setup and Training

To train the response generation model, we require our prepared corpus of review-response pairs, and the pretrained mBART model and the relevant SentencePiece model from Hugging Face[8]. Training is then comprised of two main steps: (i) applying the customizations to the pretrained mBART model and (ii) fine-tuning the customized model on review-response pairs

2.3.1 Step 1: Customizing mBART

Collect Vocabulary

The off-the-shelf mBART model requires high-resource hardware. However, the main cause for such high memory requirements is the large multilingual embedding matrix. Since our response generation model is designed to cover only English and German, we can discard majority of the embeddings for non-relevant tokens (e.g. tokens from Arabic, Romanian, Chinese, etc.). To trim mBART's embedding matrix to the task at hand, we need to provide a list of relevant subword tokens. The script `collect_list_of_spm_pieces.py` can be used to generate this list directly from the training corpus and save it to a text file.

```
python collect_list_of_spm_pieces.py \
    path-to-train-source-file path-to-train-target-file \
    --spm path-to-mbart-sentencepiece-model \
    --outfile path-to-output-file
```

Define additional special tokens for vocabulary

Since we rely on discrete labels to encode additional information (e.g. review rating, domain) with the source text, we provide a script to collect these from the corpus in order to ensure that none are missed in a particular experiment. The script `collect_list_of_special_tokens.py` extracts any tokens that appear in the corpus surrounded by alligator brackets ('<', '>'). Note, we also define the following three special tokens to demarcate distinct parts of the review text: `<GREETING>`, `<SALUTATION>`, `<endtitle>`.

[8]The model and supporting files can be downloaded from https://huggingface.co/facebook/mbart-large-cc25/tree/main.

```
python collect_list_of_special_tokens.py \
    paths-to-train-rating/domain/source/target-files
    --outfile path-to-output-file
```

Trim mBART model

Finally, the script `trim_mbart.py` applies the relevant conversions and saves the customized tokenizer, config file and model, which is now ready to be fine-tuned, to the specified output directory.

```
python trim_mbart.py \
    --base_model facebook/mbart-large-cc25 \
    --save_model_to path-to-save-new-model \
    --reduce-to-vocab list-of-spm-pieces \
    --cache_dir path-to-huggingface-mbart \
    --add_special_tokens list-of-special-tokens
```

Note: We also provide a bash script `run_convert_mbart.sh` that wraps the calls to above mentioned python scripts. To use this script, simply ensure that the path variables at the top of the script are correct for your operating system.

2.3.2 Step 2: Fine-tuning

Model fine-tuning takes place in the python script `train.py`. The model is implemented with `pytorch lightning`[9], which provides support for a wide range of state-of-the-art training techniques. For a full list of training parameters and setups, see the command-line arguments in the script or the example call in the code's README file. Below we provide a minimal example of how to use the script:

```
python train.py \
    --from_pretrained path-to-trimmed-model \
    --tokenizer path-to-trimmed-model \
    --save_dir path-to-finetuned-model \
    --save_prefix name-of-model \
    --train_source path-to-source-train --train_target path-to-target-train \
    --val_source path-to-source-valid --val_target path-to-target-valid \
    --test_source path-to-source-test --test_target path-to-target-test \
    --tags_included \
    --batch_size 8 --grad_accum 5
```

The model is used to run inference on the validation set provided at regular intervals (e.g. each epoch). These outputs are automatically saved in the 'path-to-finetuned-model' and are available for inspection during model training.

Early stopping is available for any of these metrics: vloss, rouge1, rouge2, rougeL, rougeLsum, bleu. In order to use rouge or BLEU-based metrics, you must have rouge_score and sacrebleu installed.

[9] https://www.pytorchlightning.ai/

To effectively train on a single 12GB GPU, we need to set a rather small batch size (e.g. 8). We use the `grad_accum` argument to collect gradient over multiple batch before updating the model parameters. This helps generalisation and speeds up training time. In our experiments we aimed for an effective batch size of approximately 40.

Note: Again, we provide a bash script with the relevant call for single GPU training.

2.4 Inference

Once trained, the model an be used to perform inference on unseen data. In the case of wanting to evaluate the model with automatic metrics, a reference file of target tets can also be provided. Rouge and BLEU scores will be computed automatically. For more detailed evaluation metrics (see Section 2.5), the output file of this command can be used in a later step. If reference target texts are not available, simply leave this argument out. In this case, no evaluation will be performed and the output texts will simply be written to the specified file for inspection.

```
python inference.py \
    --model_path path-to-fine-tuned-model \
    --checkpoint "checkpointepoch=name-of-checkpoint" \
    --tokenizer path-to-fine-tuned-model \
    --translation output-file --output_to_json \
    --test_source path-to-source \
    --test_target path-to-reference \
    --tags_included \
    --max_output_len max_target_length \
    --max_input_len max_source_length \
    --batch_size 2 --beam_size 6
```

Model outputs can be written to a simple one-line-per-text output format or a JSONL format which keeps the source texts, reference texts (if provided), model output scores and texts for each data point together. This is recommended as it simplifies input parameters for the evaluation scripts described below (see Section 2.5).

Advanced decoding strategies

Since we use the underlying model provided by Hugging Face, we can access to a wide range of deterministic and stochastic decoding strategies[10]. These can easily be experimented with by providing the appropriate command-line arguments (e.g. `do_sampling` with `top_p=0.9`). For a full overview of available strategies, see the Hugging Face documentation and the parameters in the underlying generation function[11].

[10] https://huggingface.co/blog/how-to-generate

[11] https://github.com/ZurichNLP/transformers/blob/4c5ff0e8497e9dcabd9fd304756bc2b31a76b3e2/src/transformers/generation_utils.py#L594

2.5 Evaluation

2.5.1 Automatic Evaluation

Automatic evaluation of review response generation is especially tricky since the range of potentially 'good' responses to a given review is endless. Nevertheless, given an evaluation set of held-out review-response pairs, we can use a range of automatic metrics as well as strong performing text classifiers in an attempt to quickly gauge the quality of a trained system. Still, it should be noted that these automatic metrics are only rough proxies and do not reliably indicate performance. Metrics used include:

Reference-based metrics Following common practice we use the word-overlap metrics BLEU and ROUGE-L. Despite the fact that these metrics are not ideal for the task and have been shown to really only 'work well' when multiple references are available, they may provide a rough indication as to the precision and recall for model predictions. We use the implementations from `Vizseq`[12].

Diversity metrics Instead of comparing model outputs to a set of references, it is useful to assess the diversity of the generated outputs. Generative language models can often get stuck in repetition loops or generate a sentence that is very similar to the previous one. Therefore, it helps to measure the intra-textual diversity. On the other hand, a model may simply generate the same output text for a variety of input texts. Thus we also want to measure inter-textual diversity of the outputs.

- Inter-textual diversity:
  - **DISTINCT-1** and **DISTINCT-2**[13] compute the number of distinct unigrams and bigrams in generated responses and scales this by the total number of generated tokens to avoid favoring long sentences.
  - **Self-BLEU**[14] computes BLEU-4 scores for each output text using all other generated outputs as references. Our implementation has been integrated into `Vizseq`.
- Intra-textual diversity:
  - **rep-w** computes the fraction of tokens that occur in the previous l tokens (i.e. single word repetition). Note, this implementation extends Welleck et al. (2019)'s **rep/l**[15] to use variable length histories.
  - **seq-rep-n**[16] computes the portion of duplicate n-grams in a generated sequence (i.e. sequence-level repetition).
  - **rep-r**[17] computes the ratio of a repeated sequence to the total sentence length (i.e. how much of the output is repeated).

[12] https://github.com/facebookresearch/vizseq
[13] Li et al. (2016) http://arxiv.org/abs/1510.03055
[14] Zhu et al. (2018) https://arxiv.org/pdf/1802.01886.pdf
[15] https://arxiv.org/pdf/1908.04319.pdf
[16] Welleck et al. (2019) https://arxiv.org/pdf/1908.04319.pdf
[17] Fu et al. (2021) https://arxiv.org/pdf/2012.14660.pdf

Trained classification metrics In order to assess whether the model outputs are appropriate for a given domain (hotel vs. restaurant) or a review rating ([1-5]), we use pretrained text classifiers to predict the domain label and review rating value for a given response text. Our classifiers are implemented with `fastText`[18].

We trained classifiers to predict the following labels:

- **domain** predicts the review domain based on the response text. (Approx. accuracy 98%)
- **review rating** predicts the review rating based on the response text. Note, this is the least accurate classifier since it is a multivariate classifier (5 possible labels) and it is trying to predict characteristics of the review text based only on the response text. Additionally, due to the low counts of negative reviews, it is also trained on the minimal amounts of response texts for each class. However, we argue that it's not so important to know the *exact* review rating, but rather to gauge whether the review is more positive or more negative. In this case, classifier accuracy is much better if you consider the Hit@2 score and aggregate neighbouring ratings.
- **source** predicts whether the style of the response (re:spondelligent vs. TripAdvisor). Note, since the final models are trained only using re:spondelligent data, model outputs always match re:spondelligent's general style. (Approx. accuracy 99%)

All classifiers were trained on as much data as possible, using TripAdvisor and re:spondelligent data combined. For each classifier, classes were evenly balanced by downsampling the majority classes.

Note, for each of the metrics, we compute corpus-level results. In other words, we average sentence-level results over the entire test set. To run an evaluation, use the script `evaluate_jsonl_format.py`. For example,

```
python evaluate_jsonl_format.py \
    fairseq-generation-log \
    --domain_ref [review-domain-reference-file] \
    --rating_ref [review-rating-reference-file] \
    --source_ref [review-source-reference-file]
```

where

- `review-*-reference-file` are **optional** files containing the line-aligned rating, domain, etc. labels for each item. If specified, output texts will be classified by the relevant `Fasttext` classifier. If not, no classification will be done.

2.5.2 Human Evaluation

Throughout the project, we used `Prodigy` to run human evaluations. We provide the relevant Prodigy recipes to reproduce an evaluation round given the JSONL output file generated by `inference.py`.

In a first step, we convert the model outputs to the JSONL format expected by `Prodigy`.

[18] https://fasttext.cc/

```
python generate_data_for_prodigy_from_jsonl_outputs.py
    --model_outputs path(s)-to-jsonl-generation-output(s) \
    --reference_inputs path(s)-to-reference-information \
    --outfile output-jsonl-file
```

where

- `path(s)-to-jsonl-generation-output(s)` is one or more JSONL output files from `inference.py` (Note, the file must be line-aligned and contain the same number of entries.)
- `path(s)-to-reference-information` is one or more files containing, for example, rating labels, domain labels, etc. for the inputs. (Note, the items specified here are stored in the meta data field for each prodigy example.)

Once the data is in the correct format, on the instance where Prodigy is installed and activated, you can use the following command to launch an annotation session.

```
prodigy name-of-custom-recipe \
    name-of-database \
    path-to-jsonl-inputs \
    -F path-to-python-script-containing-custom-recipe
```

Note, in this command, `path-to-jsonl-inputs` is the same as `output-jsonl-file` in the command above.

2.6 Interface

The final response generation model is provided as a Docker container and uses FAST-API to provide an efficient and reliable REST-API.

Similar to the sentiment analysis component, the code for the response generation model is self-contained and essentially loads all relevant models required to perform inference and to do post-processing on the model outputs. This is then served as a FAST-API app which provides self-documentation and access to the interface for testing.

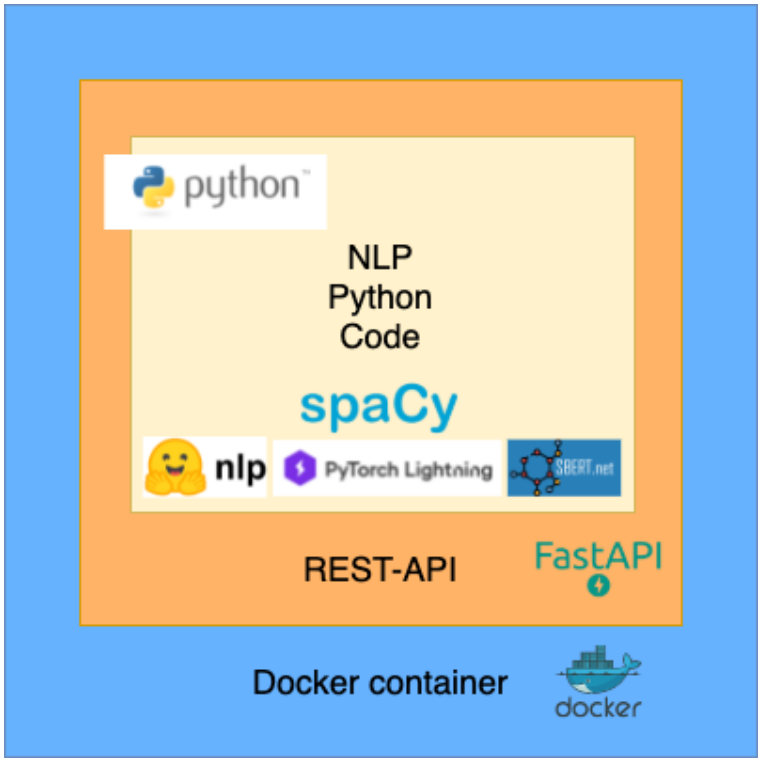

Figure 2.1: The basic architectural integration of the response generation component replicates that of the sentiment analysis system: the functional NLP code is run as a REST-API which is included in a docker container to which requests from the outside are sent.

2.6.1 Backend Details

The entry point of the response generation app is `main.py` and can be launched with the following command:

```
python main.py config.json
```

where `config.json` is a human-readable configuration file designed to aid customization and testing. This file specifies the relative paths to the underlying models and files and model parameters required for loading and generating responses. In Listing 2.6.1, we provide an example and describe the different sections.

- `models` contains the relative paths to spacy models and the names of sentence-transformer models used in post-processing.

```
{
    "models": {
        "spacy_en": "./app/models/en_core_web_md-2.3.1",
        "spacy_de": "./app/models/de_core_news_md-2.3.0",
        "sentence_transformers": {
            "xlm": "xx_paraphrase_xlm_r_multilingual_v1",
            "dist": "xx_distiluse_base_multilingual_cased"
        }
    },
    "data": {
        "company_gazetteer": "./app/models/company_names_de_en.txt",
        "test_inputs": "./app/egs/input_210512.json",
        "int_outputs": "./app/egs/intermediary_output_210512.json",
        "test_outputs": "./app/egs/output_210512.json",
        "company_label_mapping":
        ↪  "./app/models/response_generator/210527_longmBART_DER/re_establ_name_ids_ml_2020.txt"
    },
    "response_generator": {
        "type": "mBART_DER",
        "model_args": [
            "--model_path=./app/models/response_generator/210527_longmBART_DER/",
            "--checkpoint=model19.ckpt",
            "--tokenizer=./app/models/response_generator/210527_longmBART_DER/",
            "--tags_included",
            "--max_output_len=400",
            "--max_input_len=512",
            "--batch_size=1",
            "--beam_size=4",
            "--num_return_sequences=4",
            "--do_sample",
            "--top_k=10",
            "--temperature=1.2"
        ]
    }
}
```

Listing 1: Example of `config.json` file for loading and running model for inference.

- `data` defines the relative paths to files for testing and loading the mapping of company names to known establishment IDs. (Note, the latter is only relevant for `DER` models.)
- `response_generator` specifies the type of response generation model[19] and arguments required for loading the model and decoding parameters. (Note, if alternate decoding strategies are needed, simply replace the relevant arguments in `model_args`.)

As input, the model expects an input JSON object with mandatory fields: 'review', 'lang', 'domain' and 'rating'. Other fields are used for postprocessing and customizing the outputs towards a specific re:spondelligent customer. The full input to the API should aim to replicate the example provided below.

```
{
    "review": "Hatten ein Wochenende in Basel verbracht und da wir auch Karten für das
    ↪  Musical Theater hatten verbrachten wir die Nacht im Hotel du Commerce. Wunder
    ↪  schöne grosse Zimmer und ein reichhaltiges Frühstück versüssten uns das tolle
    ↪  Wochenende in Basel.",
    "meta": {
        "lang": "de",
        "title": "Schönes Erlebnis in Basel!",
        "domain": "hotel",
        "rating": 5,
        "author": "Hans Heissen",
        "gpe": "Basel",
        "loc": "",
        "email": "service@commercehotel.ch",
        "phone": "+41 XXXXXXXX",
        "url": "www.commercehotel.ch",
        "name": "hotel-du-commerce-basel",
        "company": "Hotel du Commerce",
        "greetings": [
            "Liebe*r NAMEPLACEHOLDER,",
            "Guten Tag NAMEPLACEHOLDER,"
        ],
        "salutations": [
            "Besten Dank, Hanspeter, Team Leader.",
            "Mit freundlichen Grüssen, Hotel du Commerce."
        ]
    }
}
```

Listing 2: Example input object for which we automatically generate responses.

Note: value for `greetings` can also be a string, either 'formally', 'informally'.

[19] This argument is mainly left in for legacy issues. As of the time of writing this documentation, it is not yet known if the final model includes establishment-specific labels. `mBART_DER` or `mBART_DR`.

Note: value for `salutations` can also be a string, e.g. 'DE: Liebe Grüsse Ihr Gifthüttli Team EN: Kind regards, Your Gifthüttli Team', 'Herzliche Grüsse (DE) / Kind regards (EN), Team Restaurant Gartäbeiz Eymatt62', 'Herzliche Grüsse Berggasthaus Mostelberg'.

Fields are internally preprocessed and combined before being passed to the model. The resulting output object will contain one or more responses generated by the model. The exact number of responses generated is controlled by the `beam_size` and `num_return_sequences` parameters in the config file.

2.6.2 Postprocessing

Model hypotheses are postprocessed in order to tailor the outputs towards the relevant re:spondelligent customer and to order the hypotheses from 'best' to 'worst'.

Replacing names We use Spacy to identify business names, locations, phone numbers and emails in the generated texts and replace these with the relevant fields included in the input's metadata. Since Spacy's NER module is not trained to detect restaurant names, we provide a gazetteer of approximately 30K business names extracted from the re:spondelligent database and the TripAdvisor datasets and extend Spacy's NER module using their `PhraseMatcher`[20].

Note, the gazetteer provided has not been manually cleaned and may contain problematic entries (e.g. EAT) that could lead to erroneous placement of business names. In order to guarantee reliable replacements, the gazetteer should be proofed and updated based on the names commonly generated by the model in practice. The gazetteer can be updated at anytime and read in when by when the system is initialized (i.e. executing `main.py`).[21] For details on how the gazetteer is loaded, see `spacy_utils.py`.

Custom greetings and salutations The current database structure does not allow to extract fully customized greetings and salutations simply. At present we provide two options:

1. Given current, unstructured values from the columns `greetingtype` and `goodbyetext` from customer information table `SF-guard-group`, we attempt to infer an appropriate greeting/salutation based on a predefined list of allowable values and the language code of the review. **Note:** in many cases, the field `goodbyetext` is empty, so we cannot infer a salutation.

2. Given a **list** of allowable greetings and salutations for a particular customer, randomly select 1 of each. Ideally these should come from the customer information table in the database so that they are easily manageable and can be updated if needed.

We suggest that method number 2 be adopted going forward and that the database structure be improved to facilitate automatic customization with less overhead and room for ambiguity. At present, many 'goodbyetext' values are empty or incomplete (e.g. missing the author's name) in the database. Note, if no customer-specific greetings and salutations are provided in the review input metadata, we do nothing and these values must be entered by the author.

[20] https://spacy.io/api/phrasematcher

[21] Note, functionality could be further extended to provide API access for live updates to the Spacy's phrasematcher, however, this is outside the scope of the project.

Reordering N-best hypotheses The order of model output hypotheses considers the total probability of the sequence, the semantic relatedness to the input review. Here we use `SBERT` to compute semantic textual similarity as well as simple word overlap. This rescoring technique can be easily modified according to the user's specific needs.

2.7 Evaluation of Delivered Model

The final model is trained on a database dump from June 2021. To train the delivered model, we follow the pipeline described above to extract review-response pairs from the database dump (provided as an sql file), prepare them for model fine-tuning and also split into train/test and validation sets. For future comparisons, we provide the prepared datasets along with the final models. This test set can be used to benchmark models trained in the future.

To assist with these comparisons, Table 2.1 presents the results from automatic evaluation metrics described in Section 2.5.

epoch	decoding	BLEU	ROUGE-L	DIST-1	DIST-2	Self-BLEU	rep-r	rep-w	seq-rep-n	domain acc	rating acc	source acc	hyp lens
21	bs=6	0.121	**0.287**	0.030	0.090	**0.037**	0.069	0.076	0.059	**0.975**	**0.929**	0.941	77.57
28	bs=6	**0.124**	0.285	**0.033**	**0.105**	0.044	0.066	0.081	0.060	0.971	0.923	0.943	**79.57**
21	top-k=10	0.094	0.239	0.026	0.076	0.056	0.062	**0.075**	**0.057**	0.892	0.853	0.936	75.78
28	top-k=10	0.097	0.238	0.029	0.087	0.061	**0.060**	0.078	**0.057**	0.901	0.846	**0.946**	77.71
21	top-p=0.95	0.093	0.235	0.026	0.075	0.051	0.062	0.076	**0.057**	0.895	0.844	**0.946**	75.78
28	top-p=0.95	0.096	0.238	0.029	0.088	0.076	0.062	0.080	**0.057**	0.898	0.845	0.941	77.57

Table 2.1: Automatic evaluation results of final model, trained and tested on database dump from June, 2021.

Here we evaluate two model checkpoints (epoch 21 and 28). These two checkpoints were the 'best' models according to the **ROUGE-2** metric (not shown in this table) as computed on the validation set after each training epoch. Note, we also provide evaluation scores using different decoding strategies. Sampling-based decoding strategies reduce performance according to reference-based automatic metrics but show slight improvements among the repetition-based metrics (for which lower is better).

One standout aspect of this evaluation is the measures Distinct-1 and Distinct-2. These clearly show that there is very little lexical variety in all of the generated responses. Since the lexical/textual diversity metrics are reference-free metrics, we can also compute these on reference responses from the test set to gain an idea of what we should be able to expect. Scores for reference responses are as follows: DIST-1=0.089, DIST-2=0.357, Self-BLEU=0.006. This shows that the generated responses are, in general, less lexically diverse and have significantly more lexical overlap between responses.

According to the automatic classifier-based metrics, for domain and review rating accuracy (i.e., appropriateness), beam-search decoding delivers best performance by far. Source accuracy (whether or not the response looks like a re:spondelligent response) is consistently high across all decoding strategies, since the training data contains only re:spondelligent review-response pairs.

2.8 Conclusion and Future Development

As part of the ReAdvisor project we have developed a viable and effective solution for supporting human authors in writing review responses in the hospitality domain. Our final model is a general-purpose multilin-

gual model capable of generating responses for both target languages (English and German) and both target domains (Restaurants and Hotels).

A variety of experiments with other architectures showed that generating responses to hospitality reviews is challenging due to the lack of explicit and consistent lexical alignments in source-target pairs. Inspecting results of the final mBART-based model, we see large improvements in grammatical accuracy and fluency in comparison to regular transformer and LSTM-based encoder-decoder models. For English, specifically, we also see large improvements in the model's ability to generate relevant lexical items that relate it strongly to the input review text. In contrast, inspections showed that this is not always the case for German examples, indicating that the task is harder for German reviews and would benefit from additional training examples.

2.8.1 Style and domain robustness

Despite considerable work involving training with additional data from web platforms, it became apparent that stylistic differences between re:spondelligent's data and the web-based data led to model outputs that were unacceptable for re:spondelligent's authors. Therefore, as supervised data for fine-tuning mBART, we use only review-response pairs that are written by re:spondelligent authors and accepted by their customers. This guarantees that generated responses will remain in a similar style and be of a certain quality.

As a consequence, the effective training data contains considerable bias in terms of distribution between the two domains. As shown in Table 2.2, restaurant review-response pairs significantly outweigh hotel review-response pairs. Subsequently, for ambiguous or challenging domain reviews, the model tends to generate responses that would be appropriate for the majority class (i.e. restaurants).

	Hotel	Restaurant
de	1,442	11,336
en	2,274	7,412

Table 2.2: Distribution of data between target languages and domains in data used to fine-tune mBART.

In order to combat this, we suggest increasing the amount of appropriate review-response pairs. Balancing out the domain classes should lead to improved model outputs for the hotel domain.

If sufficient review-response pairs become available for other domains, extending the model to cover these domains is trivial. The basic approach would be to simply define a new domain label token (e.g. `<camping>`) and then reconstruct the model from the Hugging Face checkpoint and by fine-tuning the model on updated data (see Section 2.3 for details).

2.8.2 Extending for other languages

Since the underlying model is multilingual, the delivered approach can easily be extended to other supported languages (e.g. French and Italian) provided sufficient training examples are available. Furthermore, extending the training data with different source-target language pairs (e.g. source language: English – target language: German) would also allow for generating a response in any given language for any given language input. These outputs could then also be used to further extend training data and improve generalisation performance.

2.8.3 Leveraging additional contextual information

Recently, there has been a number of works that aim to incorporate additional contextual information to reduce model hallucinations and ground the generated text with real-world 'knowledge'.[22] As part of the ReAdvisor project, experiments were initially conducted on using grounding information based on the approach by Zhao et al.[23], however, we found two major limitations of this approach. Firstly, since these models were trained from scratch on very limited data, the generation quality was rather poor. Secondly, grounding information did not exist for all re:spondelligent customers, which likely makes it challenging for the model to learn how to effectively utilize any grounding information. Since the generation quality from a fine-tuned mBART model is considerably better than training a model from scratch on limited data, we suggest that extending the BART architecture with grounding knowledge would be a useful technique to reduce hallucinations in the model outputs.

2.8.4 Alternative architectures

In this project, we have restricted our solutions to seq2seq architectures. Some commercial applications of computer-supported writing systems, such as Google's smart compose use a more lightweight language modelling approach which considers the input text, metadata and the previous tokens as context to predict n-gram continuations. While this works quite well, Google's system is trained on approximately 3 billion data points of user emails. For our task, we have limited data. Thus. leveraging the pretrained mBART checkpoint allows us to generate full response texts of decent quality with as little as 15K data points. Nevertheless, we encourage further exploration of architectures for review-response generation in order to improve upon the delivered system.

[22] For example, http://arxiv.org/abs/2104.12714 and BARTforknowledgegroundedconversations.

[23] http://dl.acm.org/citation.cfm?doid=3308558.3313581. See the master's thesis by Alla Stöckli (2020) for details on these experiments.