Cooking recipes generator utilizing a deep learning-based language model

Thesis · February 2020

DOI: 10.13140/RG.2.2.23904.51200

4 authors:

Michał Bień
NVIDIA
7 PUBLICATIONS **44** CITATIONS
SEE PROFILE

Michał Gilski
Poznan University of Technology
2 PUBLICATIONS **37** CITATIONS
SEE PROFILE

Martyna Maciejewska
Poznan University of Technology
2 PUBLICATIONS **37** CITATIONS
SEE PROFILE

Wojciech Taisner
Poznan University of Technology
2 PUBLICATIONS **37** CITATIONS
SEE PROFILE

Poznan University of Technology
Institute of Computing Science

Bachelor Thesis

Cooking recipes generator utilizing a deep learning-based language model

Authors:
Michał Bień, Michał Gilski, Martyna Maciejewska, Wojciech Taisner

Supervisor:
dr hab. inż. Agnieszka Ławrynowicz

Advisor:
mgr inż. Dawid Wiśniewski

Poznań, 2020

Contents

1 Introduction

1.1 Project summary

Cooking recipes are a very specific type of text, that allows to share culinary ideas between people by providing an algorithm for their realization. Creating a recipe requires a certain dose of creativity and often some of the best ones are crafting unlikely ingredient combinations. By providing an automatic recipe generator we can allow the creation of truly unique dishes with combinations no one has ever thought of. With current technology it would be very time consuming and difficult to create a generator that can distinguish a good recipe from a bad one in terms of taste, but having a model that creates them with a viable text format and sensible instructions is a step forward to a new generation of machine conceived dishes. Therefore, this work will focus on the creation of viable and original recipes that will be able to pass as real human made recipes when presented to a person.

The method that has been chosen for recipe generation is a deep learning model that will process real life recipes for training. The first order of business was the acquisition of training data that will be used by the model. In addition to using existing datasets, more data was gathered by scrapping cooking websites, to increase the variety of training data. Next the gathered data has been analyzed to understand how recipes are constructed. The result of such an analysis has been used to clean the data, so texts that were ungrammatical, irrelevant or lacking crucial features (like list of ingredients) were removed. We used this data to train a deep learning language model that is capable of generating a recipe based on some input ingredients. We then served the model in the form of a recipe generation website. A crucial part was evaluation of the generated text, that has been done by using NLG *(Natural Language Generation)* metrics as well as human based study.

1.2 Detailed project objective

The project objective is a merge of an exploratory research work and software engineering. It aims to provide the team with **shared knowledge**, **shareable outcome** and a **possible research outcome**. The detailed list of objectives can be presented as follows:

- **Knowledge:** web scraping methods, exploratory data analysis, classical NLP *(Natural Language Processing*), deep learning, state of the art statistical NLP, survey methods

- **Shareable outcomes:** **the largest public dataset** of cooking recipes, a **NER *(Named Entity Recognizer)* classifier** for detecting food, a **language model** for **recipes generation**, and a web runtime platform serving a model with modern frontend and backend stack.

- **Research outcomes:** a set of ideas on how language models can be controlled during text generation, comparison of different **automatic and manual validation metrics** for text generated by language models.

1.3 Work structure

The project was composed of 5 major steps. While the work was not strictly parallelizable, an effort was made to have each project member scope on one of the project parts. Each of the members was also in charge of one chapter of this thesis. In most cases the next step was dependent on the previous one, that's why many solutions were worked on together.

1. **Data acquisition** step was prepared by **Wojciech Taisner**. It consisted of data acquisition, filtering, deduplication, and resulted in a **general purpose recipes dataset**.

2. **Data preprocessing** step was prepared by **Martyna Maciejewska**. It consisted of exploratory data analysis, advanced data preprocessing and features preparation. It resulted in a **feature set ready for model training** and **spaCy NER model for food recognition**.

3. **Language modelling** step was prepared by **Michał Bień**. It consisted of neural architecture selection, hyperparameter tuning and model training. It resulted in a **neural language model for recipes generation**

4. **Evaluation** was prepared by **Michał Gilski**. It consisted of extensive research in the area of both automatic and manual evaluation of text generated by a language model. It resulted in **a set of methods for automatic model evaluation, compared for our use case**, and **an electronic survey for model evaluation using users judgements**.

5. **Productionalization** led to creation of proof of concept frontend and backend services for the language model. The work was done by **Wojciech Taisner** and **Michał Bień** respectively.

2 Related work

State of the art in language modelling changes very quickly. Several years ago, recurrent neural networks, like LSTM (Long-Short Term Memory Network) [16] served as the state of the art for modelling sequence data. These models had some fundamental flaws when applied to natural language. Lengthy trainings, that were not parallelizable, and limited possibility of transfer learning were among the problems, that affected every statistical language model for a very long time. [59]

Recent research in the area changed this situation. In the series of breakthroughs referenced to as ”NLP’s ImageNet moment” [50], new models such as ULMFiT (Universal Language Model Fine-tuning for Text Classification) [20], ELMo (Embeddings from Language Models) [43] and OpenAI GPT (Generative Pre-Trained Transformer) [47] emerged, each of them solving some of the previously stated problems. In this situation, it was reasonable to merge these solutions into a universal language model which offers deep context understanding, can be pretrained, and relatively easily fine-tuned.

Such solutions as Google BERT (Bidirectional Encoder Representations from Transformers) [12] and OpenAI GPT-2 [48] represent the latest generation of statistical language models based on Transformer [19] architecture. The key differences between these two models are highlighted in Section 5. These were the first models that made efficient use of large datasets gathered by Internet companies and researchers, to produce meaningful and human-understandable, logical discourse. Multiple rich language understanding utilities are based on, or currently being rewritten to use these models.

This project is inspired by two scientific resources that were created on top of these breakthroughs. It aims to provide new solutions on top of the results that were presented by these projects:

Recipe1M+ [27, 26] is a MIT Computer Science & Artificial Intelligence Lab project, which besides other resources, resulted in public dataset of over one million cooking recipes and 13 million food images. In the original work [52], the dataset was used for retrieving the recipes, given the food images. **Our work made use of published dataset and web resources to generate new, larger dataset, while dropping out the images, which were irrelevant for the research.**

Inverse Cooking [51] is a Facebook AI Research project which was an effort to generate a food recipe, given only an image of the result of cooking. As a dataset, the project made use of Recipe1M+ work. This work was particularly helpful and interesting, because it was a two-step machine learning pipeline. On the first step, multi-label ingredient classification was performed on input image, and then both images and ingredients embeddings were used as an input to transformer model generating new recipe. The second step of the pipeline was very similar to the process used in our work as it generates the recipe, given the set of ingredients. **However, our work proposes a different approach to model constraining and recipe generation. Furthermore, our work is more focused on measuring the plausability of the generated result.**

Currently there is a number of ongoing efforts to generate contextualized text for many different types of semi-structured tasks. The emerging promising solutions such as Salesforce’s CTRL [23] model were acknowledged. However they were published too late for this project to do further research on their usability or to consider them as an implementation option.

3 Data acquisition

3.1 Introduction

Model performance highly relies on the quality of acquired dataset. This is the main reason, why extending initial **Recipes1M+** [27] dataset is important part of our work. The data acquisition process involves obtaining as big as possible volume of data, restructuring into a uniform format and merging into a single dataset. Further actions include simple preprocessing, basic cleansing and duplicates removal. In the presented case, data extraction process may be split into two major parts: retrieving *available datasets* and *web scraping*.

In (mighty) quest of data acquisition, retrieving available datasets seems to be a natural choice. The great advantage of this approach is the fact, that the datasets are expected to be reliable, and by this, it is meant, that they were tested in action, or in simpler words, there was a work conducted with the help of this dataset. Moreover, this method shall be easier one, since the data should had been preprocessed by the author of the dataset, which leaves little post-acquisition work for our team.

On the other hand there is *web scraping*. This term refers to data extraction from websites, with the help of specialized programs called *web spiders* or *crawlers*. These programs follow a schedule and visit certain web pages, where they retrieve data. The process itself is likely to cause multiple challenges to overcome, but in the end it may be more effective than retrieving available datasets.

It is expected, that combination of these two approaches should result in a dataset, that will be at least the same size as the initial one. To the best of our knowledge, the dataset should become largest, publicly available dataset of this type.

Recipe format

To ease development, the uniform structure for storing recipes was defined. It enables maintaining recipes structure and includes all required source information for the reference:

- **url** - string - a reference to source of the recipe
- **title** - string - string title of the recipe
- **ingredients** - string[] - list of ingredients including units and quantities, each line as a separate string
- **directions** - string[] - list of directions necessary to prepare the recipe, each instruction in a separate string

3.2 Retrieving available datasets

When it comes to data acquisition, a natural approach is to gather already available datasets. The search of potential sources of recipes was conducted, but the results were unsatisfactory. There were plenty of ready datasets on *Kaggle* [22] and few more found on the Internet, but none of them was big enough in terms of size, comparing to the initial dataset.

Of course, there were bigger datasets regarding cooking recipes on the Internet such as *OpenRecipes* [40], but they were not suited for natural language processing, but focused on determining nutrition or type of kitchen, mostly involving only ingredients and their amounts. The main issue was that these datasets did not contain any directions, which are essential in our task of recipe generation.

The biggest dataset found during researches was *OpenRecipes* [40] - around 173 thousand distinct recipes, but it did not contain any instructions. The biggest one containing instructions was *Kaggle-Epicurious* dataset - around 20 thousand recipes. Considering every dataset containing instructions, total number of recipes gain would hardly exceed 50 thousand. It is worth mentioning, that most of datasets origin from cooking-themed websites, so it is highly likely, that these recipes will be gained anyway during the web scraping process. The *Kaggle-Epicurious* dataset may be example of such a situation, where during the web scraping process, the amount of recipes retrieved, highly exceeded the amount from the dataset.

Finally it was decided to abandon retrieving available datasets due to too low number of potential recipes and focus on web scraping.

3.3 Web scraping

Web scraping is a process of retrieving data directly from the web pages. It makes us face several challenges, to speed it up, which may be crucial for further steps.

In our case, the process started with searching for web pages containing cooking recipes. Later on, to keep process effective, the estimated number of recipes on the page was determined, usually with help of information on website or extracted from sitemap. Web pages with recipes number under certain threshold were not taken under consideration. Next step was to retrieve URLs where recipes were found, usually by extracting them from the sitemap, or generating with given pattern. After the URLs were collected, the spider script was developed, tested and deployed.

After the abandonment of the existing datasets acquisition, the web scraping became the way of extending data volume to the required size. In this project, the web scraping process was divided into two approaches, described in subsequent paragraphs.

3.4 Basic approach

This stage serves to better understand the process of web scraping, so basic solutions will be used. To deliver the dataset, firstly a custom web scarping framework was developed. Following subsections describe it in details.

3.4.1 Architecture overview

The framework was developed entirely in Python 3 programming language. It utilizes `urllib` library for *http requests*, `multiprocessing` library for parallelism and synchronization methods, along with `Beautiful Soup` [7] library for *HTML* parsing. It consists of several separated components. The most important one is a core function. It distributes URLs, given by the user provided generator across worker threads, taking care of proper threads management and request rate. Relations between these components are visualized at figure 3.1.

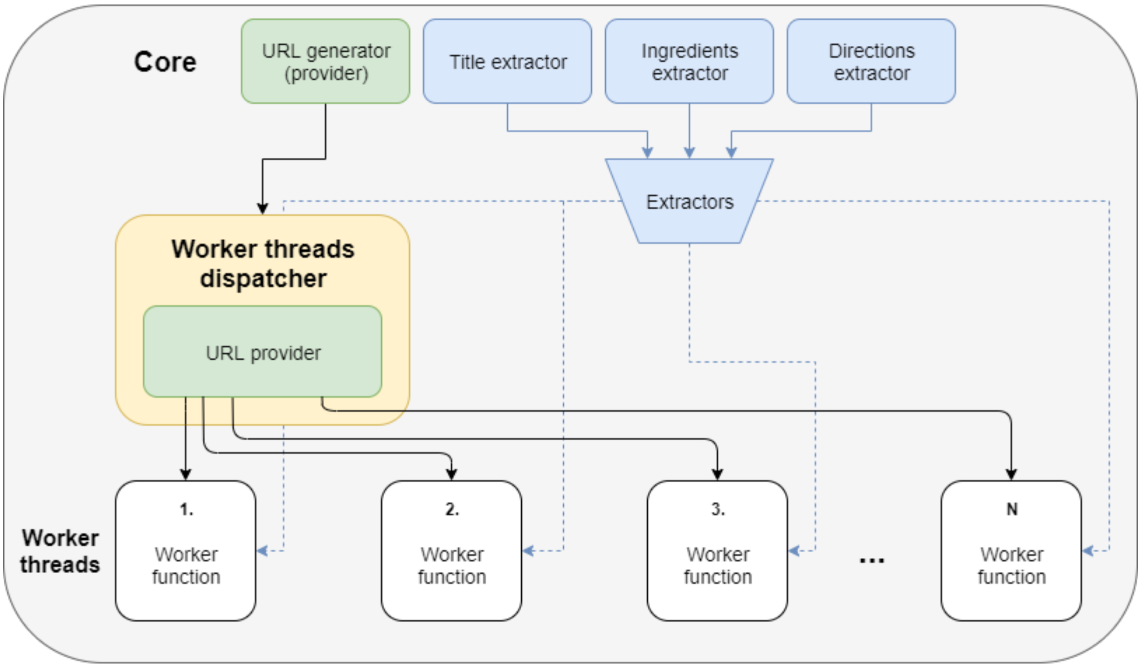


Figure 3.1: The core function relations

Second base component is a worker function, executed by each worker thread. It downloads a given website, parses its content with `Beautiful Soup` and utilizes user provided functions to retrieve data - title extractor, ingredients extractor and directions extractor. Finally it saves recipe in a single file. Flow chart of the worker function is presented on figure 3.2.

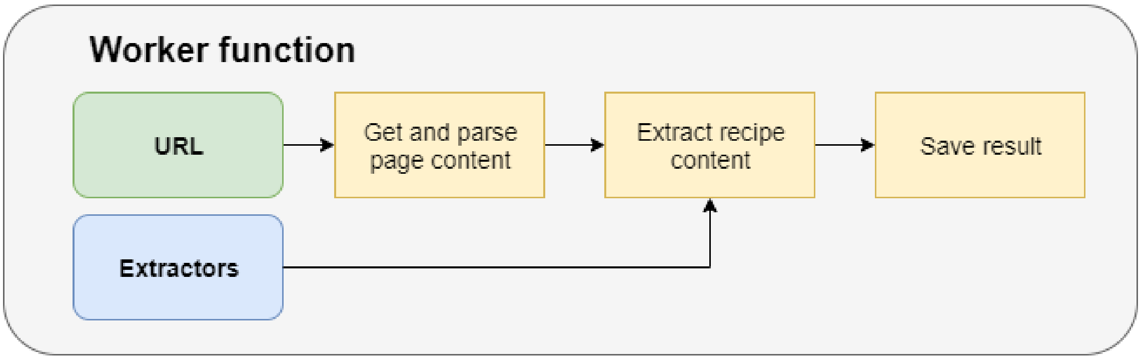


Figure 3.2: Worker function flow

The user must provide four functions for framework to work properly. First is URLs generator functions, which provides URLs for worker threads. Second is title extractor function - it parses given `Beautiful Soup` object seeking for title and returns one. Third and fourth functions retrieve ingredients and directions and work in the same way as the second function. These functions are dependent from certain web page structure, so they are almost impossible to reuse.

This framework utilizes local disk space to store retrieved recipes. The script itself follows certain convention:

- Directory must be created for each website and all script related data, e.g. list of URLs to retrieve, should be stored there.
- The retrieved data are stored in website directory, in defined sub directory.
- Recipes are stored in separate files.
- After recipes are downloaded, the `packer` script is run, to group recipes from sub directory to larger JSON files, because processing of numerous single files is ineffective.

Probably the biggest limitation of this script is that it allows only guided scraping - spider can only follow links from given list. It lacks crawling feature - ability to follow links retrieved from web page on its own, managing visited pages, avoiding revisiting, loops and so on. Due to limited time, this functionality is considered as a potential future work.

3.4.2 Recipes extraction process

The process consists of the following parts:

1. **Searching for websites where recipes are available** - since there is no aggregated information regarding web pages with recipes, the manual search must be done in order to find potential sources of data.
2. **Estimating the number of and verifying quality of recipes** - after the web page was found, it was essential to determine if it would be beneficial to develop and run spider on this website. Firstly, few recipes are chosen and manually reviewed in terms of quality. Two main factors were number of ingredients and total length of

directions. Next, the recipes number is estimated, usually based on sitemap content, so basically the absence of sitemap disqualifies from scraping in the basic approach. Number of recipes is known from number of links, which are expected to point directly to recipes, retrieved from sitemap.

3. **Developing functions** to be used by the framework during web scraping. In this step functions are developed to be used within the framework, the script is manually tested on the small subset of webpages.

4. **Running and monitoring script** when script is run on full subset of data, it is expected to run for longer period of time. During the runtime, there is little information presented to the user, because of limited log capabilities of the framework. After the process is completed, the archiving `packer` script is run to group the retrieved recipes so they can be read easier.

3.4.3 Summary and results

Table 3.1 contains summarized results of basic approach to web scraping. Note that value of total time is estimated, due to limited monitoring capabilities.

Table 3.1: Scraping results - basic approach

N.	Web page	Items scraped count	Estimated total time [hours]
1.	cookbooks.com	954675	300
2.	allrecipes.com	62022	18
3.	food.com	295256	86
4.	recipes-plus.com	20568	6
	Summary	1332521	410

3.5 Issues

During the basic approach to web scraping the following issues were faced:

1. **Crawling rules** - defined by website administrator, collected in *robots.txt* file. They mainly define URLs unavailable for web crawlers and define minimum request rate (frequency of request per second) or download delay which limits the developed script speed of data extraction. Respecting these rules is essential during the process, because violating them would probably cause denial of access to the web portal. The problem is that developed script does not adjust to rules automatically, so the limits must be set manually before each run.
Possible solution: use ready made scraping framework implementing this feature.

2. **Time consumption** - scraping process is time consuming due to certain factors: internet connection speed and certain crawling rules, mentioned above. For instance, it takes approximately a week to gather 1 million web pages, with crawl delay 0.8 second. As stated above, we aim to gather significantly more than 1 million of recipes and since the dataset is critical for the progress of work, so it must be acquired in advance, or multiple spiders must be run in parallel.

Possible solution: use environment supporting easy monitoring for parallel spider runs.

3. **Scale** - since the data extracting programs are assumed to be running for weeks, probably in parallel, more issues must be faced: failure detection and tolerance - so that minor failure will not stop the process, health checks - the process status might be checked in any time, for instance to determine if job was finished, also remotely, state recovery - the job should be able to be paused and resumed later, for instance in case where some upgrades might be required, without restarting the entire process.
   Possible solution: use environment implementing monitoring features, also remotely. It should also be failure resistant.

4. **Page structure discovery** - there are certain websites which do not allow an access to the sitemap. In these cases crawling all pages and following links is necessary, but developed script requires input list of URLs of pages to visit. In this situation, it would be best to find and use ready made solution, because development of such a feature might be tedious task and unnecessary workload.
   Possible solution: use web scarping framework

5. **Quality vs Quantity vs Ease of extraction** - according to one of fundamental machine learning rules *garbage in - garbage out*, it is essential for dataset to contain recipes as high quality as possible, otherwise final results are likely to be unsatisfactory. Unfortunately most of high quality recipes are distributed across numerous small websites and blogs. It is hard and demanding task to target and extract big amount of these. Moreover, for such a minor websites, it is often impossible to calculate total amount of recipes to extract and determine structure, so it hardens the task even more. On the other hand, there are quite big web portals, with numerous recipes, but at least some of them are rather of low quality. They often consist of low number of ingredients, and no more than one instruction, which hardly makes it good and precise recipe.
   Possible solution: gather as much recipes as possible, eventually drop the ones considered as low quality recipes.

These issues and possible solutions build quite a clear picture of what should be used for web scraping:

- Cloud environment as it should have high availability, so no failures, followed by accessibility from almost everywhere.
- Ready web scraping framework supporting URLs following.
- Some kind of daemon or other monitor tool to oversee spiders runs.

3.6 Advanced approach

Advanced approach to web scraping is based on conclusions drawn from issues faced in the basic approach.

3.6.1 Web scraping framework

Scrapy[54] is a web crawling and web scraping framework. It is written in Python 3 and has many features, e.g. it comes with predefined spiders templates including crawling template, has support for `css` and `xpath` selectors which ease data extraction, has support for external data sources like databases and many more.
Among Scrapy related projects there are two applications we can benefit from:

1. **Scrapyd**[55] - a daemon, allows deploying and running Scrapy spiders, exposes JSON API to enable remote management and minimal web interface. It runs on single machine.

2. **Scrapy Cluster**[53] - distributed crawling environment for Scrapy Spiders. It allows even faster crawling but also requires multiple nodes and is relatively hard to deploy since it requires external services and advanced configuration.

We decided to use Scrapyd since it is easier to deploy and maintain.

3.6.2 Architecture overview

Our architecture for web scraping consists of the following elements:

1. Scrapyd[55] - a daemon for Scrapy[54] spiders.

2. Mongo DB[32] - NoSQL document store database server used to store retrieved recipes.

3. Mongo-Express[34] - web client for Mongo DB - used for manual data checkup and data retrieval.

4. Nginx[37] - acts as a reverse proxy for Scrapyd[55] and Mongo-Express[34]. Enables http basic authentication and secure https connection to the server.

All elements are shipped as Docker[14] containers, orchestrated with Docker Compose[13]. For Scrapyd custom Docker container was developed, for all the others, solutions provided in Docker Hub [33] [31] [36] were used. For the client side, the minimal shell script was developed to utilize Scrapyd JSON API.

The solution was deployed on a single instance in the Google Cloud Platform [17]. Cloud virtual machines are used to minimize the risk of failures and to assure remote access to monitoring services from any place. Moreover, when necessary more instances could be deployed and used.

Figure 3.3 presents our architecture.

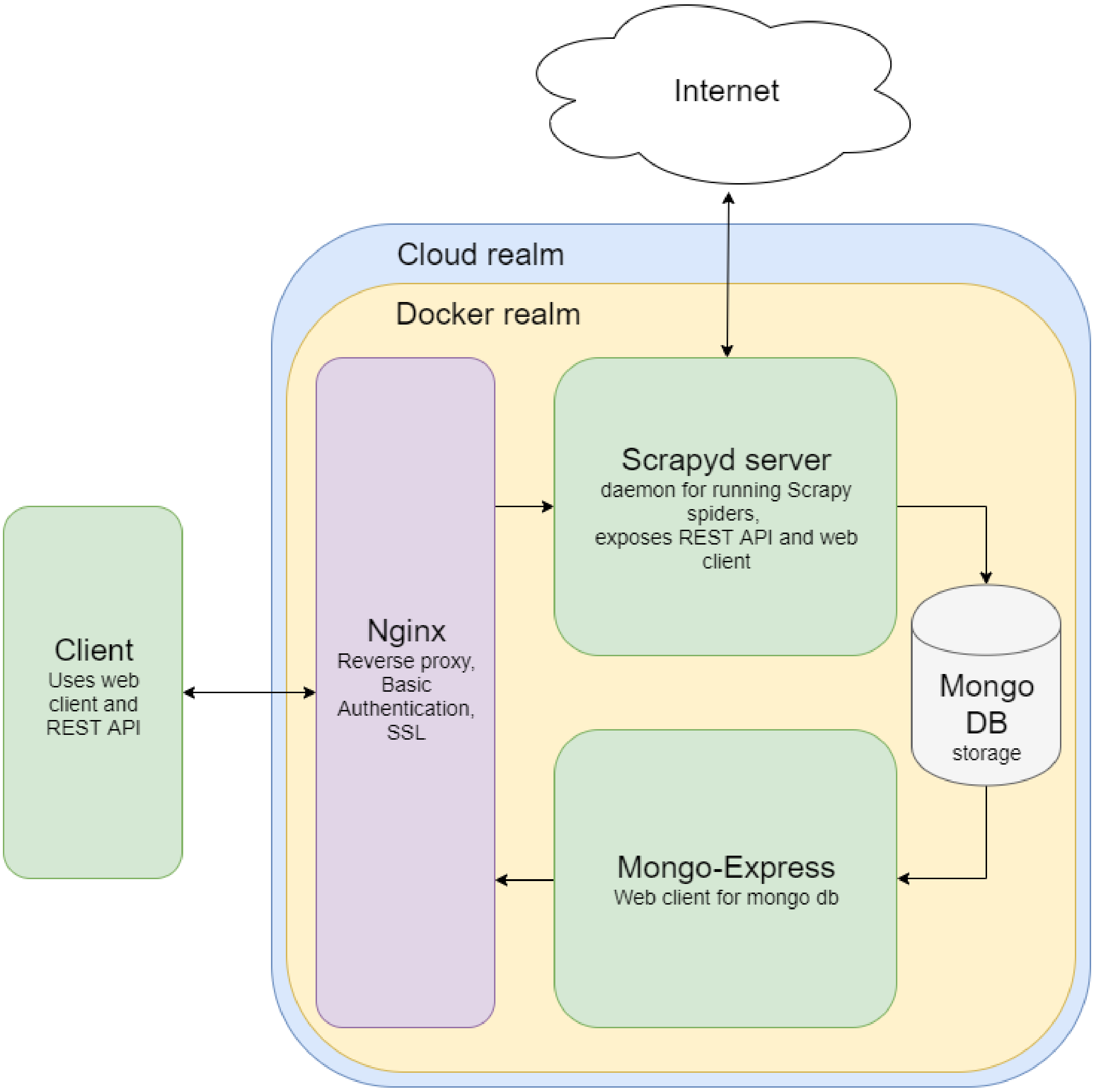


Figure 3.3: Advanced approach architecture schema

3.6.3 Recipes extraction process

There are several differences in recipes extraction process between basic and advanced approach to web scraping. This paragraph will describe only relevant differences.

When it comes to searching for websites with cooking recipes, we no longer rely on manual searching, but on data retrieved from `yummly.com` website. These data were not available during basic approach, because they could not be acquired in guided scraping, since the sitemap did not point to the most of pages. According to the information, it contains approximately 2.5 million recipes, although the sitemap does not confirm this statement. It seems like this page is perfect place to retrieve the data from, but most of the recipes are incomplete and lack list of directions. However, they usually contain the origin of the recipe, and link to the source, so it was decided to crawl this website and look for most popular sources, so they can be chosen for being crawled.

It is important to mention, that now when using Scrapy for scraping, it is irrelevant if site contains sitemap or not, because the crawling spider can be developed easily, and number of recipes can be estimated by number of references from yummly.com. The function development process did not change as well, it only utilizes Scrapy interfaces and tools, instead of custom framework.

The running stage is also simplified and more convenient, since spiders are easy to deploy on running Scrapyd instance with developed command line interface. Monitoring can be easily done as well, with minimal web user interface provided by Scrapyd. It displays current state, running time and allows logs access. Moreover the retrieved data may be viewed using database client.

3.6.4 Summary and results

The table 3.2 presents results of the advanced approach to scraping. It is worth to notice that the count of scraped items is not equal to number of recipes, because during the scraping of yummly there were 65938 actual, valid recipes and 1461480 valid references to the recipes from external providers.

Table 3.2: Scraping results - advanced approach

N.	Web page	Method	Items count	Total requests	Total time [h]
1.	cdkitchen.com	crawl	Failure: Spider permanently blocked		
2.	epicurious.com	guided	104401	106038	28.70
3.	food52.com	guided	49439	49873	13.51
4.	myrecipes.com	guided	70252	70406	19.08
5.	seriouseats.com	crawl	26685	97439	26.55
6.	tasteofhome.com	guided	54437	54564	14.78
7.	tastykitchen.com	crawl	51340	645968	175.13
8.	yummly.com	crawl	1527440	1715826	465.40
Summary			1883994	2740114	743.15

3.7 Data extraction summary

Table 3.3 presents the results of the scraping process. Although more pages could be scrapped it was decided, that the current amount of recipes should be enough for our purpose. As presented in table 3.3, 12 pages were crawled (or crawling attempt was made) and more than 1.7 million recipes were gathered. The process took approximately 1150 hours. However the point should be made, that this value is a crude approximation since especially during the advanced approach, multiple programs were run in parallel.

Table 3.3: Scraping results - summary

Approach	Number of websites	Number of recipes	Time [hours]
Basic	4	1332521	410
Advanced	8	422492	743
Summary	12	1755013	1153

Summarizing, the data set was gathered and its own size significantly exceeds the size of Recipes1M+ data set.

3.8 Post extraction data processing

Before any further actions, the data must be transformed into a usage ready form. This section describes actions involved in preparing and cleansing the dataset. There are following technologies utilized during this phase:

- Jupyter Notebook [21]
- Python 3 standard library
- Python Dask library [11]
- Python Pandas library [28]
- Python Scikit library [42]

3.8.1 Preparing and cleansing the dataset

First step in the process requires joining data from all the sources into a single, structured dataset. Currently data are stored in separate files, in JSON format. Moreover, the data from the basic approach are in standard JSON format and data from the advanced approach are MongoDB dumps in JSON-line format. In this situation, the Dask Bag API will be used. It is meant to handle semi-structured data and allows to transform them to Dask Dataframe in an easy way, so we can have the structured dataframe as we wanted. After all Dask Dataframe can be easily transformed to Pandas Dataframe.

As the structured dataset is prepared, the exploratory analysis is conducted and the following issues concerning particular recipes are revealed:

1. The very first issue encountered was that some recipes had directions stored in a single line, separated by endline characters, rather than in multiple lines. The solution for this issue involved applying a function, which trimmed whitespace characters first, split direction lines by endline character, flattened the list and again trimmed whitespace characters from every single line of directions. As the result number of recipes with single line of direction significantly decreased. It can be observed when studying the plots on figure 3.4.

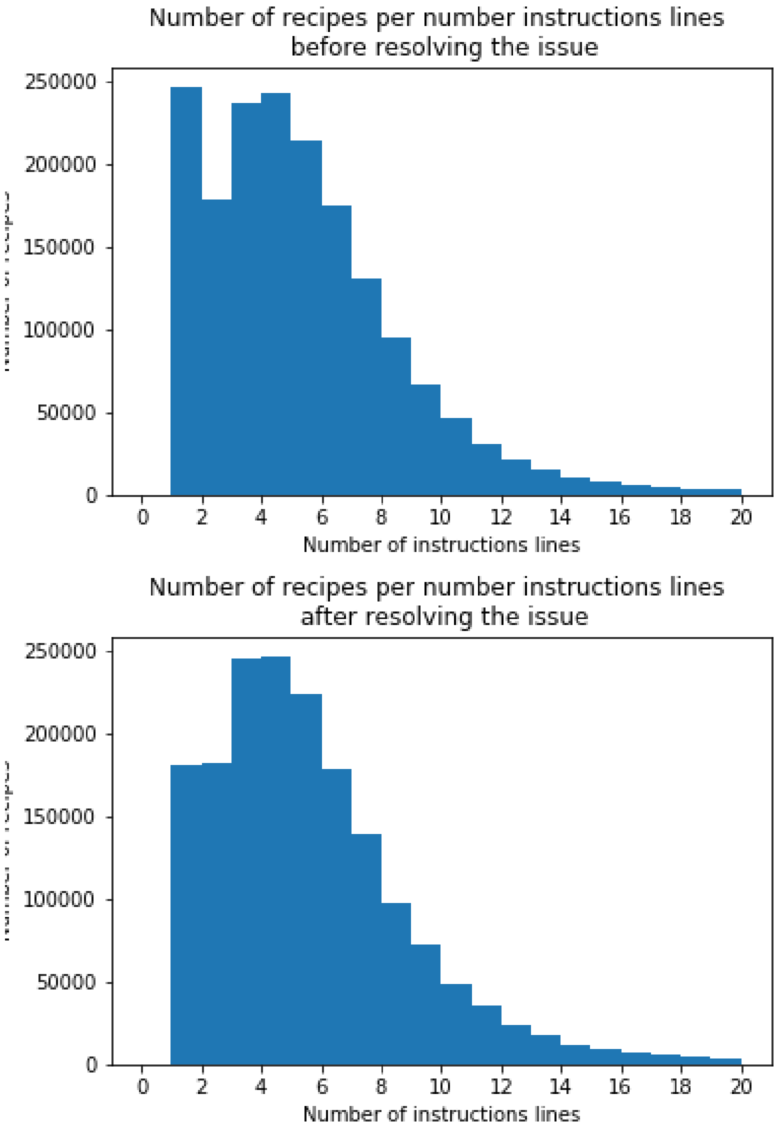


Figure 3.4: Number of recipes before and after resolving issue

2. The plots on figure 3.4 also reveled recipes with no instructions (or no ingredients respectively). Obviously, these are considered useless, so they are removed with a simple filtering function.

3. During the web scraping process, several recipes were classified as malformed during parsing due to ambiguous user input. The symptoms of malformations are instructions consisting of single words, as presented in the figure 3.5. This issue was resolved by removing such recipes from the dataset. To do so, the filter function was developed, to find recipes where single directions were probably split into multiple lines. It was decided to remove these recipes instead of fixing them for two reasons. The first one is that number of recipes categorized as invalid was around 27 thousand, so it is small number when comparing to the dataset size. The second one was the fact that we were unable to determine, where should one direction end and where should the next one start.

4. There is a group of recipes containing significantly more than single space between words, these recipes were cleansed with applying proper function.

5. When checking if all titles are written in title case, several recipes with non-latin letters and numerical, ambiguous titles was found. All of them were removed.

The next step is to merge the current version of the dataset with the Recipe1M [27] dataset. This operation is performed utilizing Pandas Dataframe features. Afterwards, the removal of recipes, that are not cooking recipes is performed. The examples of such situations are listed below.

- Test recipes - figure 3.6
- Serviette folding directions - figure 3.7
- Recipes ”For good life” - figure 3.8

These actions are directly preceding duplicates removal.
At this point there is around 2.75 million recipes in the dataset.

```
Title:          Saucy Shrimp Casserole
Ingredients:    ['1/4  c. margarine',
    '1/4  c. flour', '2  c. milk', '1  tsp. salt', ...
Directions:     ['Make', 'cream', 'sauce', 'with',
    'margarine, flour, milk, salt, Worcestershire', ...
Link:  http://www.cookbooks.com/Recipe-Details.aspx?id=619430
```

Figure 3.5: Example recipe with single word directions

```
Title:        test
Ingredients: ['138 grams test']
Directions:  ['canceled']
Link:        https://food52.com/recipes/73656-test
```

Figure 3.6: Example test recipe

```
Title:        Serviette/Napkin Folding, a Neck Tie for Dad...
Ingredients: ['1 paper, serviette (large)',
    '1 cloth, serviette (large)']
Directions:  ['Lay your serviette out flat
    before you in a diamond shape.', ...
Link:        http://www.food.com/recipe/serviette-napkin-
    folding-a-neck-tie-for-dad-244798
```

Figure 3.7: Example serviette folding recipe

```
Title:       Recipe For Happiness
Ingredients: ["2 heaping  c. patience",
    "1  heartful of love", "2  hands generosity", ...
Directions:  ['Sprinkle generously with kindness.',
    'Add plenty of faith and mix well.', ...
Link:        http://www.cookbooks.com/
    Recipe-Details.aspx?id=577815
```

Figure 3.8: Example recipe ”For a good life”

3.8.2 Duplicates removal

The first attempt to remove duplicates is to erase recipes where link duplicates or where ingredients and directions duplicates. These actions resulted in dropping around 40 thousand duplicated recipes. However this method removes only recipes being equal comparing char-by-char, but the very similar should be removed as well. That is why the further actions are conducted.

To measure similarity between two recipes, the cosine similarity measure is used. It is calculated upon the TF-IDF vector representation of recipe ingredients and directions. Term frequency $tf_{t,r}$ is number of times term t is present in recipe r. Document frequency df_t is number of documents where term t is present. With N equal to number of recipes in collection, inverse document frequency can be calculated using following formula:

$$idf_t = log\frac{N}{df_t} \tag{3.1}$$

For each term in recipe, $tf\text{-}idf_{t,r}$ can be calculated using following formula:

$$tf\text{-}idf_{t,r} = tf_{t,r} \times idf_t \tag{3.2}$$

Between each two recipes represented as $tf\text{-}idf$ vector (rv), cosine similarity is calculated, using following formula:

$$sim(rv_1, rv_2) = \frac{rv_1 \cdot rv_2}{|rv_1||rv_2|} \tag{3.3}$$

During the process, to compute vector representation and similairty, the ready-made utilities from Python Scikit library are used: `TfidfVectorizer` from `sklearn feature extraction text` and `cosine similarity` from `sklearn metrics pairwise`.

First, the TF-IDF matrix is computed from which similarity matrix is obtained. The challenge in this approach is to perform computations on the data of this size. Considering the data size, the similarity matrix would contain approximately 2.7^2 million floating point numbers, which is almost impossible to fit into computers memory, not to mention the time of computations. That’s why the process must had been paralleled.

The goal was achieved by dividing the result similarity matrix into submatrices of size 10 thousand over 10 thousand. It is important to mention, that to avoid having duplicated values, only half of similarity matrix must be computed. With this approach, the process could be paralleled and distributed across the working threads. The master thread is responsible for work synchronization and result completion. It is providing a distinct set of submatrices for each worker thread to compute. Each working thread computes values in each of given sumbatrices and emits pairs of recipes identifiers, where similarity between them exceeds certain threshold. This technique significantly reduces the size of output data. The final result comes as a list of pairs of similar recipes identifiers. It's also worth mentioning that the TF-IDF matrix serves as a shared, read only resource, so the access to its elements does not have to be additionally synchronized.

The final step is to remove duplicated recipes, which is a trivial operation when utilizing Pandas API. During the process more than 200 thousand recipes were removed, so the final size of the dataset is almost 2.5 million distinct recipes.

4 Data analysis and data preprocessing

Before the data analysis and the data preprocessing is performed, the full dataset counted 2,496,548 recipes. Over 1,000,000 recipes come from MIT Computer Science & Artificial Intelligence Lab[27] and almost 1,500,000 from web scrapping. The preparation of the dataset for training the model was a multi-stage approach (defined later in this section).

4.1 Determining the language of the recipes

Firstly, for all recipes the language of directions was checked to determine the language of the whole recipe. To achieve it, two different functions were compared: one from spiCy[56] library and another one from TextBlob [58] library. The table below presents the results for an example list of directions:

Table 4.1: spaCy vs TextBlob for language detection - a function selection

directions	spaCy	TextBlob
’Bake for 30 minutes.’	[]	[’en’]
’Melt butter in a 9 x 13-inch casserole dish.’	[]	[’en’]
’In un tegame di medie dimensioni e su fuoco...’	[]	[]
’Place chipped beef on bottom of baking dish...’	[’en’]	[’en’]

The function from spiCy library classified only the last direction as English text. The function from TextBlow library works better in terms of classification score when dealing with short directions or directions containing numbers or special characters. Based on the results of the comparison, the function from TexBlow library was selected to verify the language of the recipes.

4.2 The reasons for deleting some of the recipes

Some of the recipes were removed from the dataset. Below there are all the reasons, why the recipe will not be used in the learning process.

- The recipes with at most one ingredient. An example recipe:

```
Title:       Onion Powder
Ingredients: ['bunch Onions']
Directions:  ['Peel onions.', 'Bring water to a boil.',...]
```

- The recipes with a title shorter than four characters. An example recipe:

```
Title:       Dip
Ingredients: ['1 lb. hamburger meat', '2 lb. Velveeta cheese',...]
Directions:  ['Add other ingredients and stir until smooth.']
```

- The recipes without any directions or with a direction shorter than ten characters. An example recipe:

```
Title:       Vegan Cheesecake
Ingredients: ['8 ounces soy cream cheese', '12 ounces tofu',...]
Directions:  ['Mix']
```

- The recipes whose language were not detected as English. An example recipe:

```
Title:       Muffin au a l'ananas
Ingredients: ['3/4 cup sucre', '1/3 cup huile vegetale',...]
Directions:  ['Melanger les 4 premiers ingredients.',...]
```

- The recipes with word ’step’ in directions. An example recipe:

```
Title:       Cucumber Sandwiches
Ingredients: ['1 cucumber, lightly peeled', '1/2 tsp salt',...]
Directions:  ['Prepare ahead.', 'Step 1 can be
                completed 1 hour in advance',...]
```

4.2.1 Summary

In the classic neural network based on language model, at the input of the network there is vocabulary consisting of words, characters, or some fragments between words and characters. The presence of data from other languages unnecessarily extends the vocabulary and makes the network has more parameters to learn. Even a few recipes in language other than English can teach a model to generate directions in that language.

It is crucial to create the model that generate ”rich”, extensive recipes. Therefore, we removed recipes that do not provide the model with sufficiently comprehensive information, such as one-ingredient recipes or recipes with short instructions. Part of generating a recipe is the title generation. We intend to generating the title strictly related to the content of the recipe.

It is also impossible to check if the model has learned to refer to previous steps correctly. The incorrect use of the word ’step’ causes losing the meaning of the entire instruction.

In summary, 39,000 recipes have been removed at this stage. It is 1.5% of the original dataset. The preprocessing step has reduced the noise in the data and should increase the quality of the generator prepared.

4.3 Ingredients preprocessing

The list of inputs for the model consisted of ingredients, so for each recipe one had to prepare a list of food items without quantity and qualities.

The first idea was to prepare a list of all units and on that basis removing the quantity and unit. However, many ingredients contained not only quantity but also qualities. For

example in ingredient ’2 lb russet potatoes, peeled and cubed’ qualities ’peeled and cubed ’are not necessary information and should result from directions.
So there was a second idea based on spaCy’s Named Entity Recognizer model[56]. Named Entity Recognizer (NER) is a standard NLP problem which involves detecting and extracting named entities (people, places, time expressions, etc.) from the text and classifying them based on a predefined set of categories.

4.3.1 Ingredient annotation

To use Named Entity Recognizer for the described problem, it was necessary to teach NER what the ingredients are. In order to do this, the ingredients have been annotated thanks to the ”spacy-ner-annotator” created by Manivannan Murugavel[35]. The ”spacy-ner-annotator” allows to indicate in the text which strings of words constitute food items. Based on training data, NER is able to learn to detect potentially unobserved ingredients.
To determine the collection of ingredients for annotation, 500 recipes were randomly selected. In total, chosen recipes contained about 2,400 individual ingredients. We have extracted food items from those ingredients and classified them into a predefined category - food. Sometimes several entities were classified from one ingredient to teach the model to identify as many food items as possible.

Table 4.2: An example of ingredient annotation

ingredient from recipes	classified ingredient

1 tbsp. parsley leaves	`{entities: [['parsley','food'],`
	`['parsley leaves','food']]}`

2 tbsp. butter or margarine	`{entities: [['butter','food'],`
	`['margarine','food']]}`

1/4 tsp. freshly ground cinnamon	`{entities: [['cinnamon','food'],`
	`['ground cinnamon','food'],`
	`['freshly ground cinnamon',`
	`'food']]}`

The classified set of ingredients has been divided into the training set - 80% and testing set - 20%.

4.3.2 SpaCy NER training

Before training neural networks, metric values were measured for various model parameters.
Metrics that we selected for the purpose of the model quality assessment:

- losses – training loss for named entity recognizer; the metric should decrease during learning process

- penalty – a metric used to evaluate how precisely the model extracts food item from ingredients based on a test set; the classification of each ingredient was assessed according to the following formula:

$$penalty = \begin{cases} 0 & \text{if the result is identical to the classified ingredient} \\ 0.5 & \text{if the result coincides with part of the classified ingredient} \\ 1 & \text{if the result is not present in the list of the classified ingredients} \end{cases} \tag{4.1}$$

For example:

Table 4.3: Value of 'penalty' metric for the selected ingredient

classified ingredient	result by NER	penalty
`{entities: [['vegetable oil']]}`	"vegetables oil"	0
`{entities: [['vegetable oil']]}`	"vegetables"	0.5
`{entities: [['vegetable oil']]}`	"tablespoon"	1

The metric value was the sum of the penalties calculated for each ingredient from test set. It allowed determining when the network is overfitting.

Model's parameters:

- drop – the value serving as a dropout. Makes it harder for the model to memorize the data. The network needs to build alternative pathways to counteract the dropout. The drop was adjusted from 0.1 to 0.6 in steps of 0.1.

- number of epochs – the number of iterations was adjusted from 1 to 15 in steps of 1.

The following graph presents the value of training loss for different drops in subsequent epochs. According to the graph, the higher the dropout is, the slower convergence can be observed. If the dropout is higher than 0.5, the training loss drops very slowly.

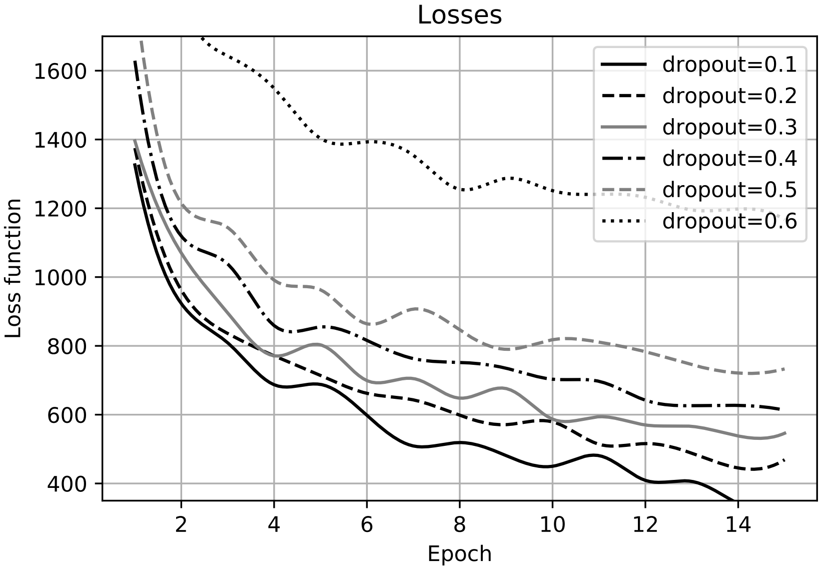


Figure 4.1: Training loss for different dropout values

The table presents the values of the 'penalty' metric for different dropout values in subsequent epochs. The higher the dropout is, the more slowly the penalty falls. According to the chart, the lowest values are observed for the dropout equal to 0.5.

Table 4.4: Value of 'penalty' metric

	number of epoch											
dropout	2	3	4	5	6	7	8	9	10	11	12	13
0.1	90.5	95	75.5	62	**60**	64.5	68	75.5	78	78.5	100	86.5
0.2	87	86	60.5	68	**58.5**	**58.5**	63	68.5	79	70.5	69.5	73.5
0.3	93	94	88.5	86.5	68.5	**52**	70	68.5	72.5	65.5	70.5	76.5
0.4	84.5	81	81	75	68	63	61.5	**53.5**	63	67.5	78.5	73.5
0.5	75	81.5	79	77	70	66.5	62.5	57.5	51	**49**	56.5	66
0.6	106.5	105	88.5	101.5	88	79.5	98	94.5	79.5	73.5	**72**	79

4.3.3 Summary

Considering the analysis provided above, it seems reasonable to choose dropout value equals to 0.5 for further computations.
The following graph presents the value of the 'penalty' metric when the dropout is equal to 0.5. The graph was helpful in choosing how many iterations should be done before stopping training the model. In the end, to avoid overfitting, it was decided to stop training the model after eleven iterations.

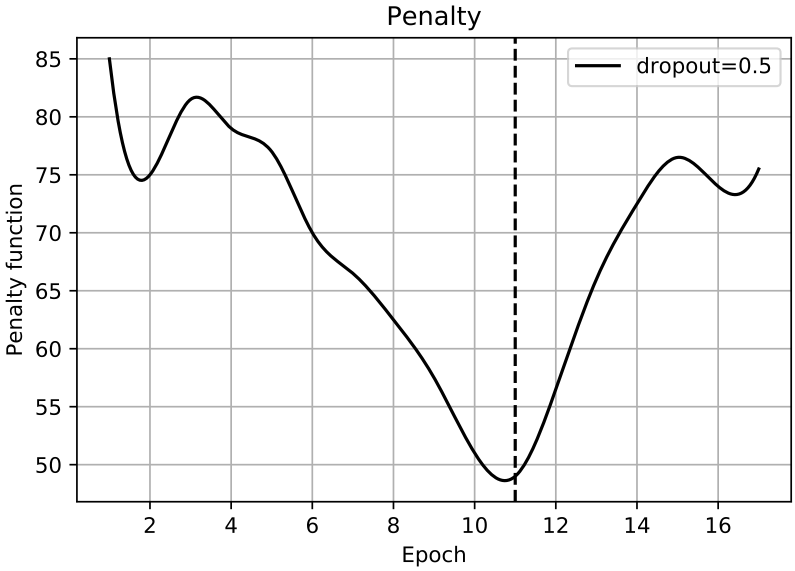


Figure 4.2: Penalty function for dropout = 0.5

4.4 Resulting Dataset

The resulting dataset is saved in a CSV file and consists of four columns with data:

- Title - an original recipe title
- Ingredient - a list of original ingredients from recipes
- Direction - a list of original directions from recipes
- NER - a list of ingredients extracted by SpaCy NER

Table 4.5: Sample row from dataset after preprocessing

title	ingredients	directions	NER
No-Bake Nut Cookies	[”1 c. firmly packed brown sugar”, ”1/2 c. evaporated milk”, ”1/2 tsp. vanilla”, ”1/2 c. broken nuts (pecans)”, ”2 Tbsp. butter or margarine”, ”3 1/2 c. bite size shredded rice biscuits”]	[”In a heavy 2-quart saucepan, mix brown sugar, nuts, evaporated milk and butter or margarine.”, ”Stir over medium heat until mixture bubbles all over top.”, ”Boil and stir 5 minutes more. Take off heat.”, ”Stir in vanilla and cereal; mix well.”, ”Using 2 teaspoons, drop and shape into 30 clusters on wax paper.”, ”Let stand until firm, about 30 minutes.”]	[”brown sugar”, ”milk”, ”vanilla”, ”nuts”, ”butter”, ”bite size shredded rice biscuits”]

4.5 List of ingredients

In order to make it easier for users to use the recipe generator and to avoid typos or an unknown ingredient, the users are able to select ingredients only from the predefined list. The list of predefined ingredients was prepared on the basis of the result extracted by SpaCy NER because food items chosen by users have to be consistent with the ingredients in the recipes. Preparation of the list required performing the following steps:

1. Collecting ingredients from the ”NER” column in the dataset and preparing a list of them.
2. Changing characters to lowercase.
3. Lemmatizing with NLTK[38]. Lemmatization is the process of changing different inflectional forms of the same word to their basic form. As a result all of these forms can be interpreted as the same thing. In many cases the correct grammatical form of the word is lost during lemmatization, but it becomes possible to compare whether two elements are identical. For example words ”apple”, ”apples” and ”apple's” will have the same form - ”apple” - after transformation.
4. Preparing a set of all ingredients with the number of occurrences and removing the duplicates.
5. Sorting the set from the most common to the least frequent phrases.

The final list only contains ingredients that have appeared in the recipes more than 1,000 times. The number was selected to ensure that the given ingredient was observed in many contexts, so nontrivial usages are expected to be generated. Selecting an ingredient from the list guarantees that it appears in the dataset and increases the chance that the model can generate the correct recipe containing selected food item.
The table below shows 5 the most popular ingredients in the dataset together with the number of occurrences:

Table 4.6: TOP 5 the most popular ingredients in the dataset

	ingredient	frequency
1	salt	1,137,720
2	sugar	741,062
3	egg	669,700
4	butter	615,537
5	flour	548,573

5 Language modelling

5.1 Introduction

The language generation is one of the tasks that are nowadays performed by statistical language models. These models predict the missing word in the text, given its surrounding. The word meaning is stored in a word embedding - dense vector of numbers, which represent the word's proximity to other words and relation with them[30]. Our work, makes use of a Transformer model family, which is a currently considered state-of-the art in the sequence generation tasks. However, there are multiple transformers models that have different usage scenarios and their architectural choices make them better or worse for some tasks.

The Transformer model architecture makes use of modern NLP finding: attention mechanism. Thus key concepts is nowadays crucial to deliver the state of the art performance in most language tasks.

Attention mechanism was introduced by work of Bahdanau et al. in 2014 [6] and Luong et al. in 2015 [25]. The overall idea is to use only the most relevant words from the previous context for text generation. The effect is achieved by calculating the weighted sum of context given by each word, and feeding it to the fully connected layer that generates the new word. The unidirectional language models, described below, use masked self-attention. In that case, the attention weights of the words that follow the currently analyzed word are zeroed, so that only the context of preceding words is counted into the consideration, instead of all the words.

5.2 Model selection

Language models can be grouped into two categories by the position of generated word. Both types of models use word embeddings in their cores.

- **Unidirectional Language Models** predict the next word, given the previous words in the sentence. While training, they use masked self-attention, which allows the model to guess the next word based only on the previous part of the sentence. These models are especially useful for text generation tasks where there is no context at the right side of the predicted words. The major strength of the approach for our use case is the model's autoregression. In each step, the model generates next word based on all the previous words - including the ones generated in the previous steps. Language models such as GPT-2[48] are unidirectional models.

- **Masked/Bidirectional Language Models** predict the masked word in the middle of text, given its surrounding. This kind of model has an additional feature of taking into consideration both sentences on the left and right side of the predicted word. While the additional context gives an extra insight for the model, it highly increases model complexity. First MLM architectures (such as BERT [12]) didn't support autoregression by design, but some of the most recent research on the XLNet model architecture [62] managed to bring it back to the bidirectional models.

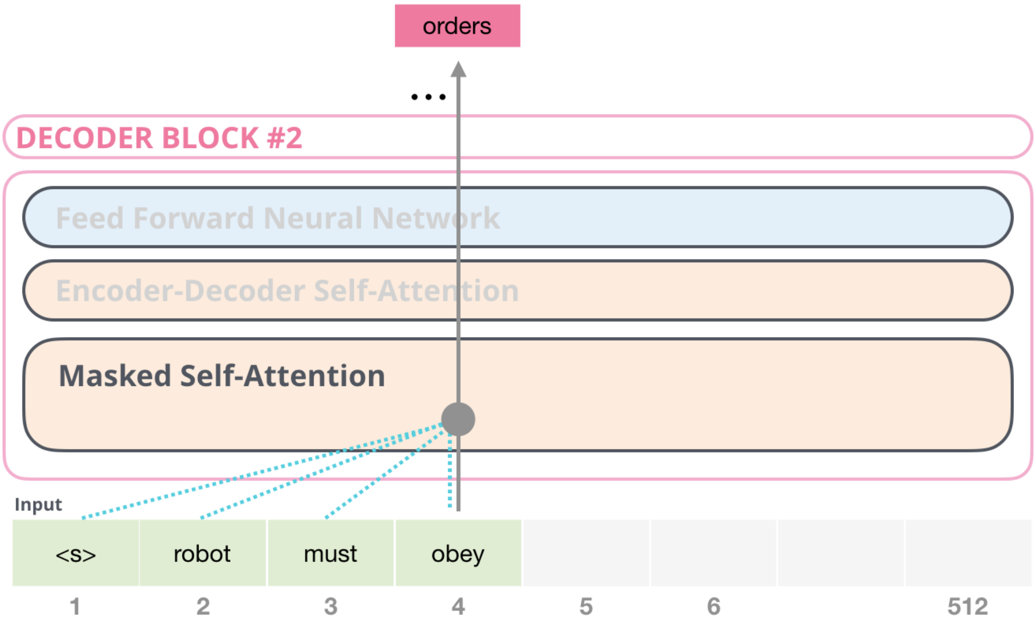


Figure 5.1: The GPT-2 next word generation step. *(by Alammar, Jay (2018). The Illustrated GPT-2 [2])*

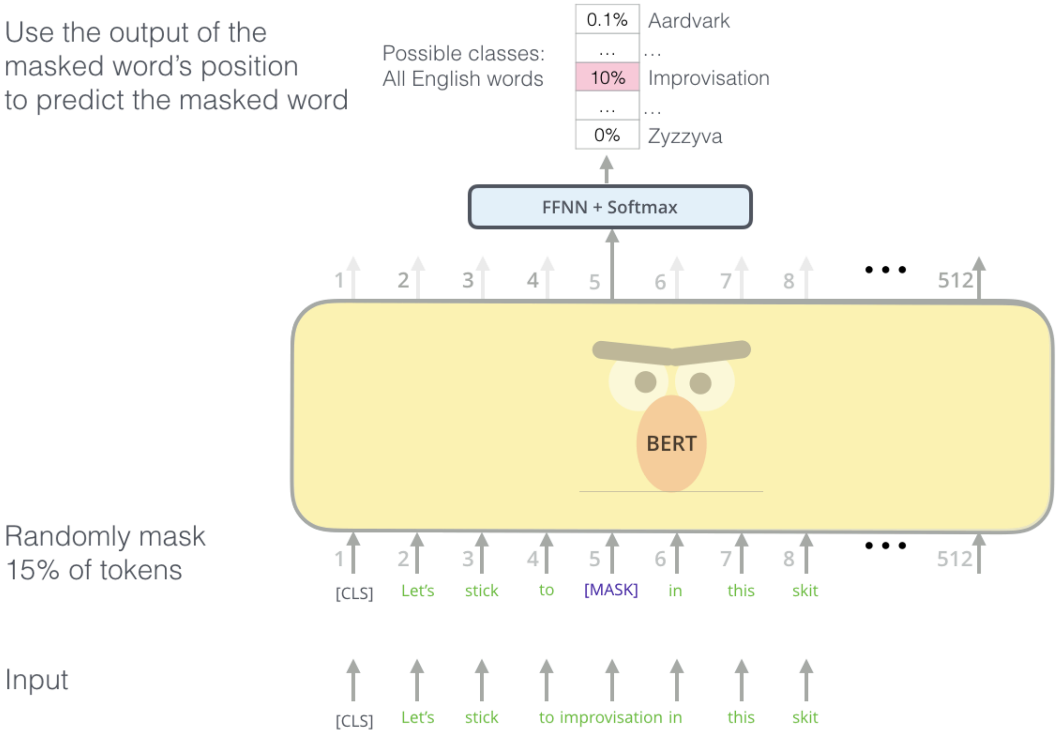


Figure 5.2: The BERT MLM next word generation step. *(by Alammar, Jay (2018). The Illustrated BERT, ELMo, and co. [2])*

As cooking recipe generation can be intuitively perceived as left-to-right generation technique, where the reason precedes the cause, and understanding of following words doesn't seem highly rewarding, the idea of using MLMs was dropped **and GPT-2 model architecture was selected for implementation**. The original, general purpose GPT-2 model trained by Huggingface (see Section 5.6) was fine-tuned for this specific task. The details of the fine-tuning process are outlined in the following sections. The only difference between the bare dataset and the information provided to the model, was a NER step, performed using supervised spaCy model. It was considered, whether MLMs could be used as a post-processing step, adjusting the ingredients amounts and fixing the possible flaws of recipes generated by GPT-2. However, after the first GPT-2 model was trained and presented surprisingly good performance, it was decided that the available computational capacity should be used to continue training on the small models rather than trying the more complex, bidirectional one. The larger models needed significantly more time and resources to train, which was expected to slow down the iterations of trainings with different datasets and hiperparameters.

5.3 Control tokens

While pretrained GPT-2 offers rich dictionary of language tokens, the structure generation task, which was important for this solution, enforced use of additional, custom control tokens. These tokens were formed as uppercase control phrases in angle brackets, with words divided by underscores. The proposed text structure resembles XML format, with opening and closing tokens, that hold the ”values”, (e.g. title) within. The model was expected to understand and reproduce this schema, making it easy to fill all the expected fields and populate the typical recipe template.

Table 5.1: Custom control tokens in GPT2

Token	Description
<INPUT_START>	Beginning of user input - from predefined list
<NEXT_INPUT>	Next element of input list
<INPUT_END>	End of user input
<TITLE_START>	Beginning of recipe title
<TITLE_END>	End of recipe title
<INGR_START>	Beginning of generated ingredients list
<NEXT_INGR>	Next element of generated ingredients list
<INGR_END>	End of last generated ingredient
<INSTR_START>	Beginning of first cooking instruction
<NEXT_STEP>	Next cooking instruction
<INSTR_END>	End of last cooking instruction
<RECIPE_START>	Start of the whole recipe
<RECIPE_END>	End of recipe. Works as end of text delimiter

5.4 Data ingestion

The complete data ingestion flow that allowed transformation of prepared CSV dataset into the final tokenized H5 database consisted of three steps: formatting, train-test split and tokenization. In each step, the whole dataset was processed and then saved to the temporary output files. The final h5 file was used directly by the training script to provide fast random access to dataset rows.

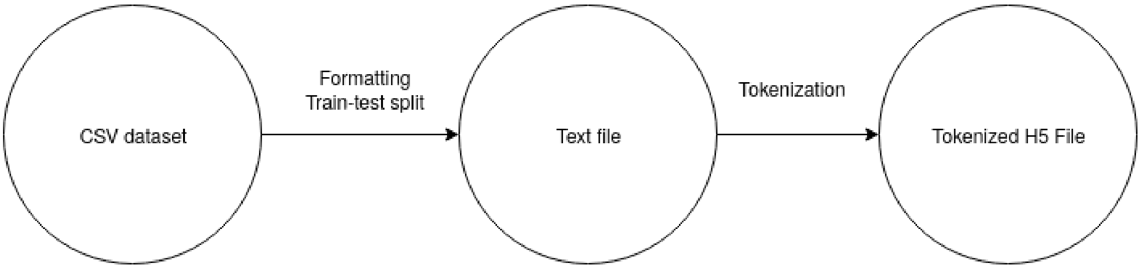


Figure 5.3: A complete data ingestion flow

5.4.1 Formatting

The data processing step, that preceded language modelling resulted in a four-column CSV dataset:

- **title** - a plain-text recipe title
- **ner** - a list of ingredient names extracted by NER
- **ingredients** - a list of original ingredients, together with their amounts
- **directions** - a list of cooking steps

In the formatting step, the dataset was prepared in such a way, that control tokens were inserted into the recipe. Each recipe was embraced with *RECIPE* tokens and consisted of: list of inputs, list of ingredients, list of instructions, and finally: a title. The order of elements of recipe was made for purpose. It was decided, that for the generative left-to-right model, it is reasonable to first provide it with input, context data, then allow it to generate the recipe itself, concluding with the title with regard to all the generated data.

```
<RECIPE_START> <INPUT_START> beef <NEXT_INPUT> garlic <NEXT_INPUT>
↪  pepper <NEXT_INPUT> rice <NEXT_INPUT> salt <NEXT_INPUT> tomato
↪  sauce <INPUT_END> <INGR_START> 4 lg. peppers <NEXT_INGR> 1 c.
↪  quick cook rice <NEXT_INGR> 1/2 teaspoon salt <NEXT_INGR> 1/4
↪  teaspoon pepper <NEXT_INGR> 3/4 lbs. lean beef <NEXT_INGR> 1/4
↪  teaspoon garlic pwdr <NEXT_INGR> 1 (8 ounce.) can tomato sauce
↪  <INGR_END> <INSTR_START> Combine all ingredients. <NEXT_INSTR>
↪  Clean and core peppers, spoon ingredients into peppers and place
↪  on microwave pie plate. <NEXT_INSTR> Cover with plastic wrap.
↪  <NEXT_INSTR> Cook on High 18 to 20 min. <NEXT_INSTR> Remove wrap
↪  and top with Cheddar cheese if you like. <INSTR_END>
↪  <TITLE_START> Microwave Spicy Stuffed Peppers Recipe <TITLE_END>
↪  <RECIPE_END>
```

Figure 5.4: Example of recipe text prepared for training

5.4.2 Train-test split

The formatting step resulted in the list of strings, each being one recipe ready to tokenize. This was the moment where 5% of evaluation data was extracted from the dataset. The amount of data was intentionally selected big enough to be further divided and used as dev set and test set. The additional test set was also prepared for the purpose of checking evaluation metrics. The text form of train and test datasets was saved into two distinct files, one recipe by line.

5.4.3 Tokenization

The implementation of GPT-2 that was used for this work (see Section 5.5) makes use of its own tokenizer. As this task may be still considered fine-tuning of the ready language model (even though the dataset is uncommonly big), it was decided that the original tokenizer will be used to convert all of the data into the tokens. The only difference was addition of recipe control tokens to the set of known tokens before tokenization started. However, it might be the case that some of less-common ingredients or kitchen utensils were transformed into the sub-word form, due to their absence in the dataset that the original model was pretrained on. It was concluded that, while the problem exists, it should not highly affect fine-tuned model performance.

Therefore, in this step, every word in the dataset had to be processed through tokenizer, resulting in one or more tokens. It was then possible to store recipe as an array of integers (token ids) that could be read by model and used for training.

In GPT-2 language model, the maximum amount of tokens processed as one context is equal to 1024. As it was discovered that the recipes consist of much smaller amount of tokens (around 300 tokens per recipe), it was decided that multiple training samples may be squashed into one context window delivered as a training example. This way, the training time on the whole dataset was made more than 3 times shorter. While there was a risk of lowering the quality due to having contexts coming from different recipes in the single context window, no significant model performance decrease was observed.

The output of the tokenizer, which was a matrix of integers of shape $(n\_squashed, 1024)$ was finally stored in HDF5 file. The HDF format was selected, because it allows fast random lookups into the memory while removing the need of loading two files (both train and test set can reside in one file in the distinct *hdf5 databases*). The use of HDF data files make data processing more efficient. The HDF support was provided by **h5py** Python library.

5.5 Hyperparameter tuning and training

While original GPT-2 was trained using TensorFlow framework, our work made use of a dedicated library, called Huggingface Transformers, which uses PyTorch in its backend.

PyTorch is an open source machine learning framework that allows design and execution of computational graphs on a number of different processor and accelerator architectures. The code is written in Python, interpreted and sent to the target machine using its native computational framework. Two different backends were used for this workflow: CUDA, which runs on NVIDIA graphics accelerators and XLA (Accelerated Linear Algebra) [61], which enables PyTorch training and inference on Google Cloud TPU devices. For this work, version 1.3.1 of PyTorch was used (latest at the time of solution development)

Huggingface Transformers is a de facto standard library providing a set of state of the art language models along with all the utilities needed to work with them. The library maintains a curated repository of pretrained models. It provides three pretrained GPT-2 models out of the box:

Table 5.2: GPT-2 Configuration for different model sizes

Model name	No Layers	Parameters per layer	No attention heads	Total no parameters
gpt2(-small)	12	768	12	117M
gpt2-medium	24	1024	16	345M
gpt2-large	36	1280	20	774M

The library also allows fast prototyping on different GPT-2 model sizes, which was particularly useful given the accelerator sizes that were used. Furthermore, it provides default tokenizer for GPT-2. **Because additional control tokens were added, the embedding layer was also resized to learn representation of those during the training process.** Moreover, the example finetuning implementation code was not suited to the needs of this project. Most notably, the data ingestion flow used python serialization pickle format, and therefore was not optimized for storing large datasets. The code also didn't provide remote model checkpoints to the cloud object storage. As these features were important for our training, **a custom execution script was prepared**. For this work, version 2.3.0 of Huggingface Transformers was used (latest at the time of solution development).

5.5.1 Loss and evaluation metrics for models

During model training, two metrics were periodically checked and stored in order to show the progress of model learning process. The first one was cross entropy loss function, calculated on training data and directly used for model training. The other one - perplexity - was calculated on dev set every 10000 optimization steps. The perplexity was checked and plotted gradually, but the final value of this metric was calculated on a dedicated test set.

Cross entropy is a loss function used by the language model to show the model performance and backpropagate the results. The idea is to show, how much the probability of the actual word in text is different from the probability of the word generated by model. For this to work, language model is given part of the text from the training sample, and is expected to generate the next word. If the word used by the model has similar probability to the word that originally stood in that place in the sample, the cross entropy value is low. The formula to calculate the value, where w is the word from the sample and $\hat{w}$ is a word selected by model:

$$-\sum_{i} w_i \log(\hat{w_i})) \tag{5.1}$$

Perplexity is a measure that shows how well a probability distribution can predict a sample. Because language model can be treated as a probability distribution over generated text, the measure is commonly used to show model fitness to particular task. However, it is known that the model with lower perplexity level could sometimes be less convincing for humans that the one that generalizes worse (and has higher perplexity). As such, perplexity is a good automatic metric that cannot be treated as a single source of truth about model performance.

By definition, perplexity evaluation value can be described as an inverse probability of test set returned by the trained model. That's why an evaluation metric is expected to be as low as possible. The general formula for perplexity can be transformed to the form that is easy to use for our solution:

$$\exp\left(\sum_{x} w_x \log \frac{1}{\hat{w_x}}\right) \tag{5.2}$$

Therefore, calculation of the perplexity value given the loss of forward passes of language model is an **exponential value of mean evaluation loss**.

5.5.2 Efforts to train on Google Cloud TPU

Thanks to TFRC (Tensor Flow Research Cloud) quota[57], that the project was provided with, it was possible to train GPT-2 model on Google Cloud TPU in version 3. Each TPU machine consisted of 8 processing cores, and 16 GB memory per core. The TPU architecture, which is dedicated for deep learning models training and inference, was much more powerful than the CUDA that was used in another approach outlined in Section 5.7.2. However, TFRC grant was only one month long, and didn't account Google Cloud Compute VM instances costs, so it was only used after the final dataset and model architecture was prepared and tested on smaller GPT-2 shape.

PyTorch XLA is an experimental PyTorch extension, used together with patched version of PyTorch. The key idea of the library is to translate PyTorch computational graphs to be run on Google Cloud TPUs. While the library is currently in early beta state, it claims to provide decent performance and ease of use. Indeed, the programming effort to prepare the code for XLA training was not very high, especially after code parts from **pytorch-tpu/transformers**[46] and **allenai/tpu_pretrain** [3] were found and used to transform the original training script into XLA-compatible one.

The first models were successfully trained and the training speed was much higher than on GPUs. What is important, each Tensor Core was capable of processing up to 6 gpt2-small training samples at time, and TPU consists of 8 such cores, which easily makes computations up to 48 times faster (minus the synchronization barrier time after each optimization step) than the GPU that was used (see below for GPU training specification).

Unfortunately, it was fast discovered, that **the generative performance of models trained by TPU was very low compared to GPU-trained model**. The resulting text was barely readable, and didn't follow the control tokens structure. The problem was being troubleshooted and distributed training was considered the most probable reason, however, when TPU was run in one Tensor Core mode, the performance issue remained all the same. After exhaustive code audits and multiple runs with different configurations, the problem was discussed on pytorch-xla bugtracker[45]. As the problem was very obscure (on XLA both evaluation and loss suggests decent model training, and the performance issue can be noticed only by human) and the creators of library were unable to reproduce it without using the dataset and implementation (which were not published at the moment), **the TPU training was abandoned and we made use of GPUs as main training backend**.

5.5.3 Training on NVIDIA CUDA

The Google Cloud GPUs, along with local development GPUs were used to suit computational needs of the project. While the solution was emerging, two general-purpose NVIDIA graphics cards were used: GTX980 with 8GB RAM and RTX2060 with 6GB RAM. The final model was trained using Google Cloud GPU NVIDIA Tesla P100 graphics accelerator with 16GB RAM memory. All cards made use of NVIDIA CUDA and cuDNN frameworks to greatly accelerate the model training times.

NVIDIA CUDA, together with deep learning extensions such as cuDNN, is a mature and frequently used framework for parallel computations. It takes advantage of a NVIDIA GPU architecture, which exposes thousands of FPUs (floating point processing units) to increase speed of both forward and backward passes in transformer model fully-connected layers. The full model is loaded into GPU memory and the kernel - a CUDA program compiled especially for GPU - executes the training on the GPU's local copy of model. The training data is loaded from RAM in batches of fixed size. For this work, version 10.1 of CUDA was used (latest at the time of solution development)

NVIDIA Apex library was also installed and used, to further optimize computation on NVIDIA GPUs. Apex (A PyTorch Extension) is a set of routines and amendments to PyTorch code, prepared and curated by NVIDIA, which can eventually improve training performance. In this work an AMP (Automated Mixed Precision) extension was used. AMP automatically refactors PyTorch computational graphs by replacing some of FP32 tensors with FP16 ones. This operation aims to both decrease model memory usage and training times, while it could negatively affect model precision and loss function decreases. Apex AMP offers four levels of optimization (O0 - O3). O2 was selected and used for all the training operations on GPUs. The Apex library is not versioned - the latest repository master revision at the time of solution development was used.

5.5.4 Final models

The two final models were trained for comparable amount of time, with setup and parameters outlined in the table below. The final checkpoints for gpt2-medium managed to get lower perplexity value than gpt2-small, and therefore might be considered better trained. Based on the graphs above, it's reasonable to say that the gpt2-small model cannot offer much more performance increases with more epochs or bigger dataset training. **However, it seems that the perplexity metric may be still improved with longer training of gpt2-medium or gpt2-large on more powerful accelerators. In such a circumstances, the NVIDIA Apex optimization may be also switched off to increase model capacity.**

Table 5.3: Final GPT-2 models training parameters

model	bs	lr	% of data used	dev-set size	test perplexity	training time
gpt2-small	4	5e-5	100%	8000	3.056	42h 49m
gpt2-medium	1	5e-5	around 35%	1000	2.905	42h 42m

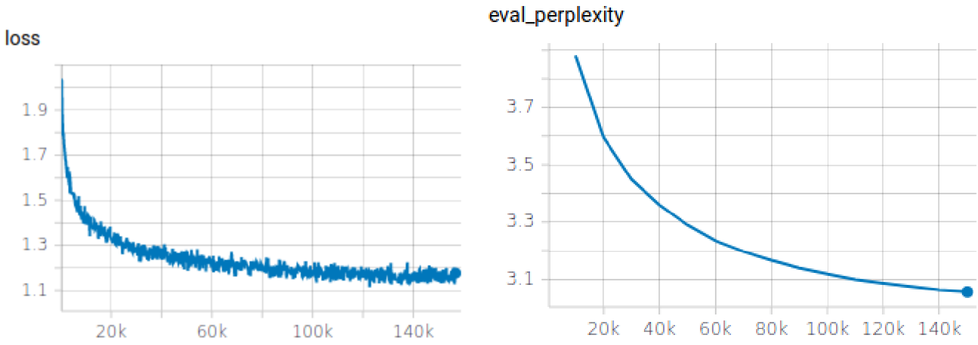


Figure 5.5: Training performance graphs for GPT2-small training on Google Cloud GPU NVIDIA Tesla P100 accelerator. The left graph shows the batch training loss (Y axis) counted every 50 optimization steps (X axis). The right graph shows the evaluation perplexity on dev set (Y axis) counted every 10000 optimization steps (X axis). Each optimization step batch consisted of 4 samples.

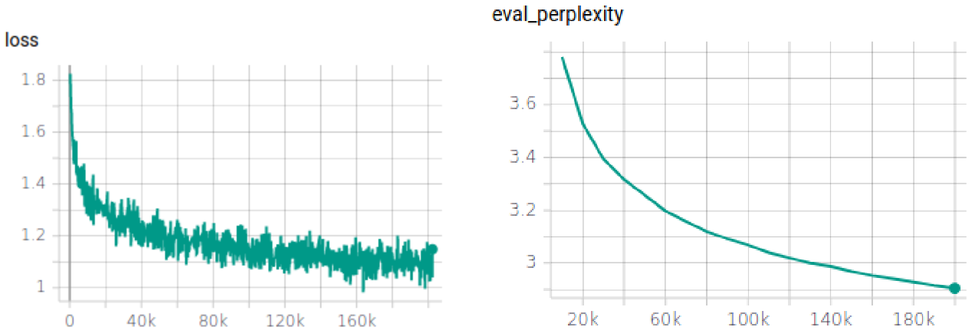


Figure 5.6: Training performance graphs for GPT2-medium training on Google Cloud GPU NVIDIA Tesla P100 accelerator. The left graph shows the training loss (Y axis) counted every 50 optimization steps (X axis). The right graph shows the evaluation perplexity (Y axis) on dev set counted every 10000 optimization steps (X axis). Optimization step consisted of 1 sample.

6 Evaluation

Due to the model specialization and fine-tuning for a very specific task, creation of meaningful evaluation metric was not trivial. There were multiple different approaches considered, which may be intuitively split into two groups. Computational methods evaluate the model without using any human knowledge during the evaluation process, whereas human expert methods are entirely human-led.

6.1 Computational methods

Computational methods can also be split into several categories. The biggest issue seems to be that the presented problem of Natural Language Generation **does not have universally good metrics**, that could automatically evaluate the generated text [39]. Moreover most of the existing metrics are dedicated to the machine translation problem and are not perfectly suited to the problem of recipe generation and can give results of questionable reliability. Other metrics can only evaluate only single parameters of the text like grammatical correctness or readability [39]. Given the unsupervised nature of the generator and the variety of types of recipes that can be valid there does not seem to be one single metric that can reasonably evaluate the model, therefore a combination of metrics seems to be the best way to automatically estimate the quality of the generated text.

6.1.1 Test data selection

A part of the training data from the dataset (5%) has been set aside for testing purposes and not used in training. From this set 100 recipes were selected (due to processing time required for evaluation) to form a **sample set**. A process of generating recipes based on the input ingredients from recipes from the test set (as shown in 6.1) has been performed to generate 10 new recipes per every of the 100 selected recipes. This process was done for two different models that have been trained - gpt2-small and gpt2-medium. This resulted in a sum of 2000 generated recipes that have been used to automatically evaluate the two models, with every set of 10 having the same inputs as a reference real recipe from the test set.

6.1.2 Cosine similarity

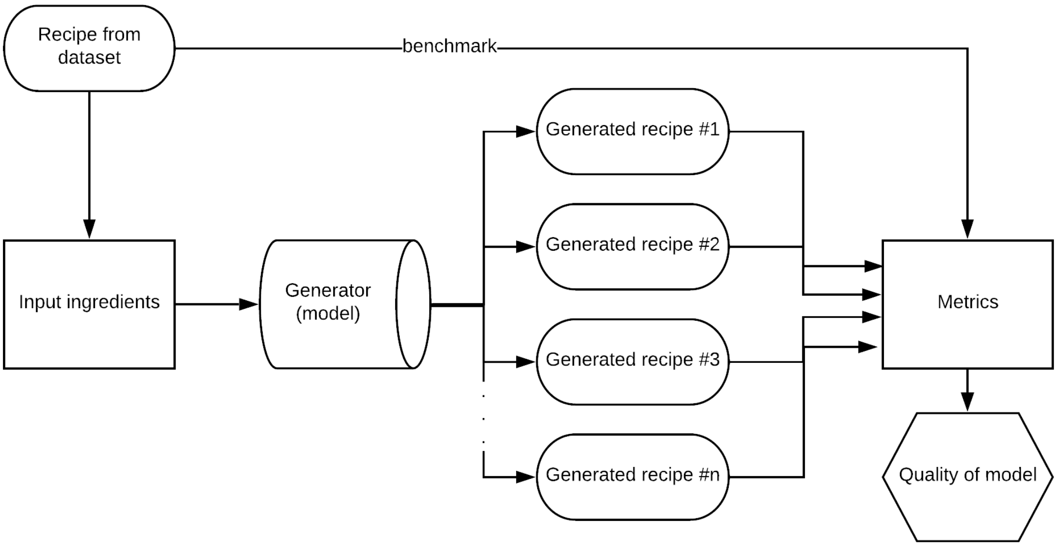
A measure of similarity of two texts can be measured by using **cosine similarity** [10]. The texts are represented as multidimensional vectors that represent the frequency of every word. Next the cosine of the angle between these two vectors is calculated by the following formula:

$$\text{cosine similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2}\sqrt{\sum\limits_{i=1}^{n} B_i^2}},$$

where A_i and B_i are components of vector A and B.

Testing

Figure 6.1: Pipeline of the automatic evaluation process (for one recipe from the test set and for a specific model



Cosine similarity was tested on both the gpt2-small and gpt2-medium models. As this method is based on comparing two texts, the model test sets are compared to the sample set, which is the source of the input for the generated recipes. Each recipe in the sample set was compared using cosine similarity to the 10 recipes that were generated based on its input ingredients. To take into account the variety of dishes that can be made using the same set of ingredients and under the assumption that at least one of 10 generated recipes describes a highly similar dish, only the highest score was used to calculate the total average for a given model.

Table 6.1: Comparison of the average cosine similarity between the sample set and the two models test sets

	gpt2-small	**gpt2-medium**
Directions	0.518	0.519
Ingredients	0.638	0.647
Title	0.459	0.47

Analysis of results
The results for both models are very similar, with the gpt2-small model achieving slightly better results. This is possibly due to it being trained on more data that the gpt2-medium model and thus can create on average more similar recipes to the ones in the sample set.

6.1.3 Grammatical and spelling check

Another metric worth considering is the overall quality of the text in regards to the amount of spelling and grammar mistakes it contains. To calculate this for a recipe, a module for Python called **Language Check** [44]. It is capable of detecting different types of errors based on specific grammar rules and a built-in dictionary that the text is checked against. While running the module on some test recipes, it has become obvious that the dictionary used by it is not exhaustive enough to reliably check for misspellings in a text that contains many rare and occasionally foreign words. Therefore, the spelling check was disabled, leaving the module to only evaluate the grammatical correctness. Some of the errors detected in the generated recipes may have occurred because of these types of errors in the recipes from the training set. To counter for this, a benchmark was run on all the recipes in the training set to determine an average number of mistakes per recipe. Another advantage of using a comparison to a benchmark, rather than using the absolute number of error, is that any mis-classification done by the language check module will not significantly affect the results.

Testing
The language check module was ran on the following sets of recipes:

- Training recipes - the human created recipes used in training of the model (~2.4 million recipes)
- Test set - part of the dataset that was left out for testing purposes (~120k recipes)
- Sample set - randomly selected part of the test set, for their inputs for creating a test set of generated recipes (100 recipes)
- gpt2-small - set of recipes generated from the gpt2-small model based on input ingredients from the sample set (1000 recipes)
- gpt2-medium - set of recipes generated from the gpt2-medium model based on input ingredients from the sample set (1000 recipes)

The results of these tests can be seen in 6.2.

Table 6.2: Results of grammatical correctness check for different recipe sets (average number of errors per recipe)

Recipe set	Training set	Test set	Sample set	gpt2-small	gpt2-medium
Average no. errors	0.491	1.237	1.59	0.68	0.869

Analysis of results
The smallest amount of errors has the training set. Surprisingly throughout the ~15h computing time the average kept steadily raising from a 0.1 to almost 0.5. This would suggest that the recipes in the beginning of the dataset were generally better in terms of grammatical correctness than the ones at the end of the dataset. A potential reason for this can be that in the version of dataset used the recipes were mostly grouped by origin, which would explain this curious property. Next, the test set and its subset (sample set) had the highest score, which may have resulted from their relatively small size (compared to the training set), which makes the results less reliable. The last and most important results are the tests done on the generated recipes test set, which again were not too big, due to high computing power required for the generation of recipes. The smaller model received a better score than the larger one, which may have resulted from the small model constructing simpler and shorter sentences that usually have less of a chance to create erroneous sentences. In

general the models, at least in terms of the language check, yield pretty satisfying, yet not clearly reliable results (what can be seen when comparing scores of different sized set of real recipes). The models performed a bit worse than the broadest benchmark test (the training set), which was expected, but it was still better than a randomly selected subset of the dataset, which places it in the range of the results on real recipes.

6.1.4 Readability

A metric called readability is also a property of text that can be used to evaluate the text. It determines the ease of reading a specific text. Lower scores indicate a text is easier to read and understand, while a high score is assigned to texts using more complex and convoluted sentences. When calculating scores for the models, a smaller score is more desirable, because in the directions i recipes should ideally be easy to read and simple to understand. There are many different ways to calculate readability, thus an average from 4 popular formulas were used [49]:

- Dale Challe Formula [8] Dale–Chall readability score is calculated based on the following formula:

  $$\text{Dale–Chall score} = 0.1579\left(\frac{\text{difficult words}}{\text{words}} \times 100\right) + 0.0496\left(\frac{\text{words}}{\text{sentences}}\right)$$

  (Difficult words are defined as word with two or more syllables.)

- SMOG Formula / Grading [29]
  The SMOG grade is calculated based on the following formula:

  $$\text{SMOG grade} = 1.0430\sqrt{\text{number of polysyllables} \times \frac{30}{\text{number of sentences}}} + 3.1291$$

- Gunning Fog Formula [18]
  Gunning fog index is calculated based on the following formula:

  $$\text{Gunning fog index} = 0.4\left[\left(\frac{\text{words}}{\text{sentences}}\right) + 100\left(\frac{\text{complex words}}{\text{words}}\right)\right]$$

- Flesch Formula [15]
  Reading ease for the Flesch formula is calculated based on the following formula:

  $$\text{Reading ease} = 206.835 - 1.015\left(\frac{\text{total words}}{\text{total sentences}}\right) - 84.6\left(\frac{\text{total syllables}}{\text{total words}}\right)$$

Testing
In order to create a reference values for the readability metrics, an average of each of these metrics was calculated on the sample set, containing 100 random recipes from the test set, as seen in 6.3. Because of the extended period of time necessary to calculate readability scores (around 25 seconds per recipe) it would be very computationally heavy to achieve an average for the entire test set, thus only the smaller sample set was used. The scores were also calculated on the gpt2-small and gpt2-medium test sets, which contain 1000 recipes each and have been generated using input ingredients from the recipes in the

sample set. Results for these two sets can be seen in 6.4 and 6.5. For each of the recipes in all the tested sets the scores were calculated separately for the directions, ingredients and title parts of the recipe.

Table 6.3: Readability scores for the sample set (100 recipes from the test set)

	Dale–Chall score	**SMOG grading**	**Gunning fog index**	**Flesch Formula**
Directions	6.483	6.094	12.638	104.729
Ingredients	6.734	5.912	13.852	106.57
Title	2.537	3.113	4.96	195.444

Table 6.4: Readability scores for the gpt2-small test set (1000 recipes)

	Dale–Chall score	**SMOG grading**	**Gunning fog index**	**Flesch Formula**
Directions	6.516	6.101	12.486	105.964
Ingredients	6.758	5.46	13.996	106.185
Title	2.86	3.1	5.183	195.363

Table 6.5: Readability scores for the gpt2-medium test set (1000 recipes)

	Dale–Chall score	**SMOG grading**	**Gunning fog index**	**Flesch Formula**
Directions	6.42	6.114	12.292	106.106
Ingredients	6.806	5.621	14.273	105.132
Title	2.639	3.102	5.072	195.171

Analysis of results

The most significant part of the recipe in term of readability is the directions part, as it contains the most complex grammatically text. In regard to this, the results from the sample set and the gpt2-medium set seem to be overall lower than the scores from the gpt2-small model, with there being no significantly dominant set of scores between the sample and gpt2-medium sets. This may suggest that the gpt2-small model produces text that has lower ease of reading overall, while the gpt2-medium has successfully matched the readability of real human made recipes. The scores in the ingredients and title category seem to be more or less similar in value for all the sets, with these scores being less significant because of the list format used in the ingredients part of the recipe and the short length of the titles.

6.1.5 Machine translation metrics

The overall issues

Even though the given problem is not specifically machine translation, it is possible to approximate it that way by treating the input ingredients of the model as a source language and the generated recipe as the target language. Most machine translation metrics rely on reference text for evaluation, so it had been necessary to find reference text for the tested generated recipes. It would be hard to find recipes in the dataset that match ingredients with the generated recipe, because of the big, but still limited size of the dataset, that cannot possibly contain all ingredient combinations. Also, the quality of reference based metrics heavily rely on a big amount of references, especially with text like these ones, that have a big variety. For that purpose the whole dataset would have to be used to increase the number of references found, which would mean that the model is evaluated against examples from its training set. An easier way to find references would be to reverse the process by starting with a real recipe and generating recipes based on input as the ingredients used in the recipe. This also gives an additional advantage of being able to use just a separate test for evaluation, without decreasing the number of possible references.

The metrics

In the automatic evaluation three common machine translation metrics [9] were used from the Pyhton NLTK module [38]:

- BLEU [41] (BiLingual Evaluation Understudy)
  This metric measures the amount of overlapping words in the given text in comparison to multiple reference texts. The best matching reference is always used as the final score.

- GLEU [1] (Google-BLEU)
  A modified version of the BLEU metric, designed to improve its reliability by additionally using the precision of the comparison as opposed to just its recall.

- WER [60] (Word Error Rate)
  Calculates the Levenshtein distance of the two text, which is the minimal number of edits necessary to change the tested text into the reference text. If many reference text are present a minimum score for them can be outputted as the final result.

Testing

The scores for these three metrics were calculated on the 100 recipe sample set. From the 10 generated recipes corresponding to each sample recipe only the highest score was taken into account in the the WER metric, while BLEU and GLEU were designed to have many possible references.

Table 6.6: Results of machine translation metrics on the gpt2-small and gpt2-medium test sets.

	BLEU	**GLEU**	**WER**
gpt2-small	0.02	0.003	7.59
gpt2-medium	0.033	0.007	7.71

Analysis of results

The BLEU and GLEU results are better with higher scores, while the WER shows more similarity with a lower score. Taking this into account the gpt2-medium model had better results in BLEU and GLEU, while having a worse score in the WER analysis.

6.1.6 Automatic evaluation conclusions

As mentioned before, the automatic metrics are far from perfect in evaluating generative language models like the the ones presented. Also due to high computational power required to generate and run some of the metrics, many of them were calculated only on a small am mount of data. Nevertheless the metrics seem to indicate great similarity of the generated recipes to human made recipes that have been acquired. Overall the smaller gpt2-small model performed slightly better than its counterpart with more parameters in all metrics except of readability. Given that the gpt2-medium model was only trained on a third of the data that the gpt2-small model used, it seems possible that this may be the reason behind it. Also the during testing of the recipes it was revealed that the gpt2-medium model occasionally produced recipes that were incorrectly formatted due to the a too long recipe being generated. If the gpt2-medium model had been trained on the whole dataset, it probably would have performed better that its counterpart with less parameters, as more parameters would give it an advantage.

6.2 Human expert methods

The human evaluation was done using an online survey. Its first part was getting some basic information from the participants, which will help set up some context for their answers. Next the participants were given recipes to give back feedback on whether they think the recipe was made by a human or a machine. The survey itself was created and managed using a survey engine called LimeSurvey [24]. A screenshot of a recipe survey question can be seen in 6.2.

6.2.1 Basic information

The information collected on the beginning was the following:

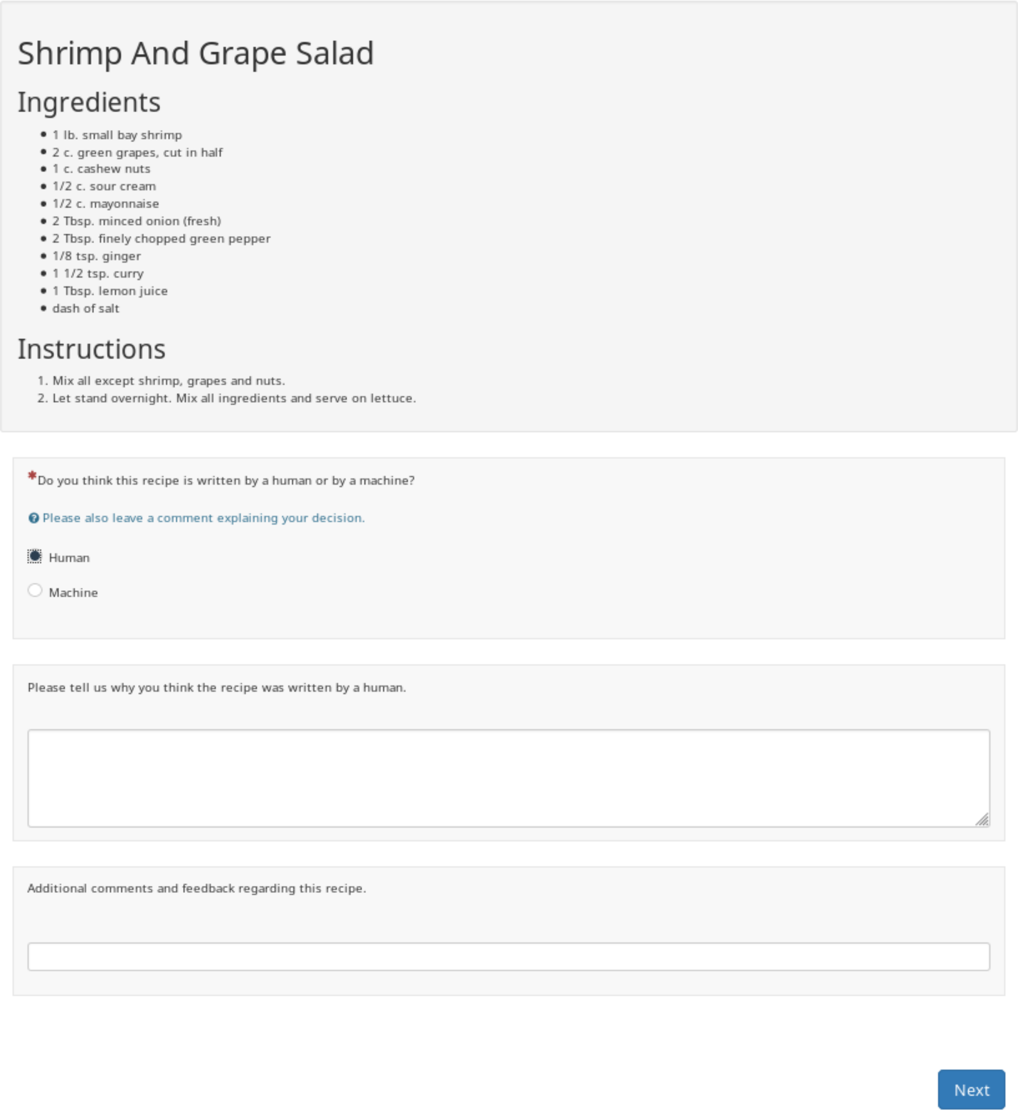
- *"How often do you cook?"*
  Cooking frequency is important in terms of the evaluation as more cooking experience makes the person more knowledgeable about typical everyday recipes and the rules that govern them.

- *"How would you rate your cooking skills?"*
  Cooking skills are more important when the participant is evaluating more exotic recipes, (which the model generates, because of their presence in the training data). This can help us evaluate if the ingredient combinations, even unusual can be viable and good in taste.

- *"What is your English level?"*
  Having a certain level of English is crucial when evaluating recipes since the survey participant needs to be able to understand the text. Participants with a high English level are especially useful to determine the language correctness of the generated recipes.

- *"How would you rate your knowledge in Natural Language Generation and Deep Learning?"*
  Knowledge of the processes that are used in the model can make the participant more aware of typical errors made by these kind of Natural Language Generation models.

Figure 6.2: The question page from the survey presented to the participants.



Shrimp And Grape Salad

Ingredients

- 1 lb. small bay shrimp
- 2 c. green grapes, cut in half
- 1 c. cashew nuts
- 1/2 c. sour cream
- 1/2 c. mayonnaise
- 2 Tbsp. minced onion (fresh)
- 2 Tbsp. finely chopped green pepper
- 1/8 tsp. ginger
- 1 1/2 tsp. curry
- 1 Tbsp. lemon juice
- dash of salt

Instructions

1. Mix all except shrimp, grapes and nuts.
2. Let stand overnight. Mix all ingredients and serve on lettuce.

*Do you think this recipe is written by a human or by a machine?

Please also leave a comment explaining your decision.

● Human

○ Machine

Please tell us why you think the recipe was written by a human.

Additional comments and feedback regarding this recipe.

6.2.2 The recipe test

Survey participants were asked to determine if a given recipe was written by a human or generated by a machine. Additionally they have been asked to give feedback to support their decision and potentially help pinpoint telltale signs of both machine generated and human written recipes. The participants were given 12 recipes to evaluate, of which the first two were test recipes and a mix of 4 real ones, 3 from the medium model and also 3 from the small model.

6.2.3 The first two recipes

The first two recipes in the survey have been inserted there for validation purposes. The first one has been a recipe from the broken ”zucchini” model. It contained very obvious and visible errors, not actually keeping to the recipe format. The purpose of this recipe has been to eliminate not serious answers. Evaluation from participants that have marked this recipe as human made has not been taken into account. The second test recipe was also purposely flawed in a more subtle way to check the attention of the participants. Originally it was also supposed to verify how reliable the rest of the answers are, given that not noticing obvious mistakes may result in said participant not being able to notice more subtle errors. After analysis of the results, it was decided against using the second recipe as validation, because surprisingly most participants marked it as correct. A possible explanation for this may be that after the first recipe the participants have wrongly assumed that the rest of the generated recipes will be of similar quality, which may have lead to them marking the second recipe as human made just by looking at its correct format and proper grammar.

6.2.4 Evaluation recipes

The next 10 recipes presented to the participant have been a the actual recipes that have been prepared to evaluate the two trained models. A randomly ordered set of 4 real recipes from the test set, 3 generated recipes from the small model test set and 3 generated recipes from the medium model test set was used in the evaluation. Each recipe was presented to the participant via HTML format with radio buttons to select the creator of the recipe: ”Human” or ”Machine”. Additionally a text box was included lower on the page in which the participants were asked to provide feedback to justify their decision. For the purposes of making the recipes more realistic a common error made by the models was corrected automatically in post processing. This error resulted from part of the dataset missing the slash symbol between fractions, something that could be relatively easily remedied by replacing common incorrect concatenated fractions (like 12 or 34) with their correct format.

6.2.5 Survey results

Over the course of 5 days that the survey has been active 176 responses were collected, out of which 46 were complete responses with answers to every question. Given that the number of responses was not particularly high, **answers to the ”Basic information”**

section were not used to filter responses in order not to decrease their reliability due to a small sample size any further.

Test recipes
As expected the fist test recipe was easily recognized by the participants with **104** correct responses and **4** incorrect ones. Participants from the second group were not considered in the following analysis. The second test recipe had a much wider spread with **36** responses labeling it correctly as machine generated made and as much as **62** incorrect responses.

Classification ratio
As seen in 6.7 most recipes were labeled as human made with the gpt2-small model being able to fool more people than the gpt2-medium model. Also the human made recipes achieved only a ~70% correct classifications.

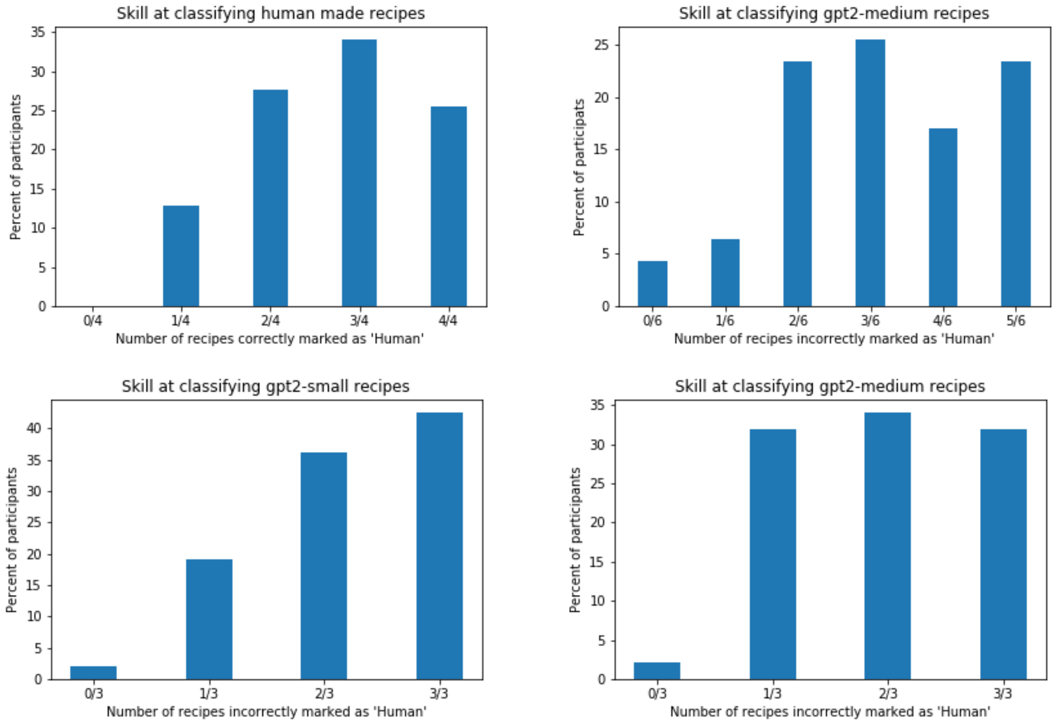
Table 6.7: Ratio of correct responses for every recipe type

	Correct response ratio
Human	68.17%
Small	28.29%
Medium	33.88%

6.2.6 Response analysis

Analysis of the answers by specific participant responses was also made. Only full responses were considered for this analysis (the survey resulted in 46 full responses). 6.3 shows how many of the participants marked a given number of recipes as human made grouped into four graphs: real human made recipes, all machine generated recipes and separately the recipes from the two models.

Figure 6.3: Number of recipes marked as human made in given group by the percentage of participants that have marked them that way



Some interesting conclusions can be drawn from this:

- Less than 5% of participants were able to correctly identify all of the generated recipes, yet all participants identified at least one correctly.
- Less than 11% of participants were able to identify at least 5 out of 6 machine generated recipes.
- More than 40% of participants were unable to identify more then 1 generated recipe and more than 20% of the participants were not able to identify a single one of the machine made recipes.
- The gpt2-small model was able to fool people more often than the gpt2-medium model - participants marked recipes from the latter as machine made more often.

6.2.7 Comments from participants

We have asked the participants to leave comments justifying their decision as to the creator of a presented recipe. Many comments pointing out errors in specific places were received, thus ensuring that at least some of the answers had a solid basis for their classification. Examples of comments include:

- *The recipe makes sense, but carbonara usually includes meat – some kind of pork. The title should indicate that it is vegetarian.*
- *I don't think a NN would generate this ' (I use whole grain, but regular bread crumbs will do) '* (Regarding an actually generated recipes from the medium model)

- *if zucchini is fried, it shouldn't be drained*
- *Odd way to describe ingredients. I don't think a person would use caps lock in such sentences.* (Regarding a human made recipe from the sample set)
- *Some of the recipes were actually pretty tricky to tell apart. I wouldn't be surprised if I took some human mistakes for machine mistakes.*

These comments show just how tricky it can be to properly recognize generated recipes, yet some pointed out errors unveil a certain lack of deeper understanding of the text by the model. Overall these comments can give good insight into some improvements for some possible future work, as they reveal many indicators that humans look for when analyzing recipe authenticity.

6.2.8 Conclusions from the survey

Overall the survey indicates that the models are capable of generating recipes that can easily fool many people, because of their similarity to human made ones. Again, same as in the automatic evaluation, the gpt2-small model performed better than the gp2-medium model. The models seem to have learned some human writing habits, that most people do not expect from something that is entirely computer generated. These results indicate that the models are able to mimic in a mostly indistinguishable way the source data that they have been trained on, while still generating new unique recipe.

7 Productionalization

One of the key goals of this work has been providing a complete proof-of-concept usable product that is based on the previous research and model training. Therefore, it has been decided that a web application will be created. The use case of the application is as follows:

- User owns a fixed set of ingredients and want to use them to prepare meal.
- User access the frontend of the application, selects the ingredients from the list and selects whether the list of ingredients is exhaustive (if it's not, the model might add more ingredients to prepare better meal).
- Frontend sends request to the API endpoint of the backend service. The request is translated into control token language understandable for the model, and the generation process begins.
- The connection to the client remains open while the result is being generated, meanwhile the backend periodically sends a status update.
- When the result is ready, it is displayed back to the user in a human readable form.

7.1 Solution backend

The backend architecture consists of two microservices working in a producer-consumer architecture. The scripts are written in Python and make use of Celery to orchestrate their work. The Celery worker and client are connected using a RabbitMQ message queue. This allows creation of a web service, that apart from serving the request status to the frontend service and creating asynchronous language model inference tasks, also doubles as a Flask web API server which communicates with the backend. The service exposed to the Internet is a Gunicorn web server which acts as a WSGI proxy handling the traffic for web service.

The tasks scheduled by web service are gradually processed by model-service, which runs the actual model inference process.

7.2 Solution frontend

User interface is a simple web application developed with Angular [4] and Angular Material Library [5]. The application utilizes server API, to allow user to request recipes generation from given input. Firstly user chooses ingredients from provided list with help of simple search tool. After request is submitted via API, client application receives request ID and subscribes to Server Site Event listener, with given ID. Server sends information via event stream to notify web application about progress of recipe generation. Last event message in the stream is a complete recipe. Afterwards stream is closed and recipe is presented to the user, who eventually can generate another one in the same process.

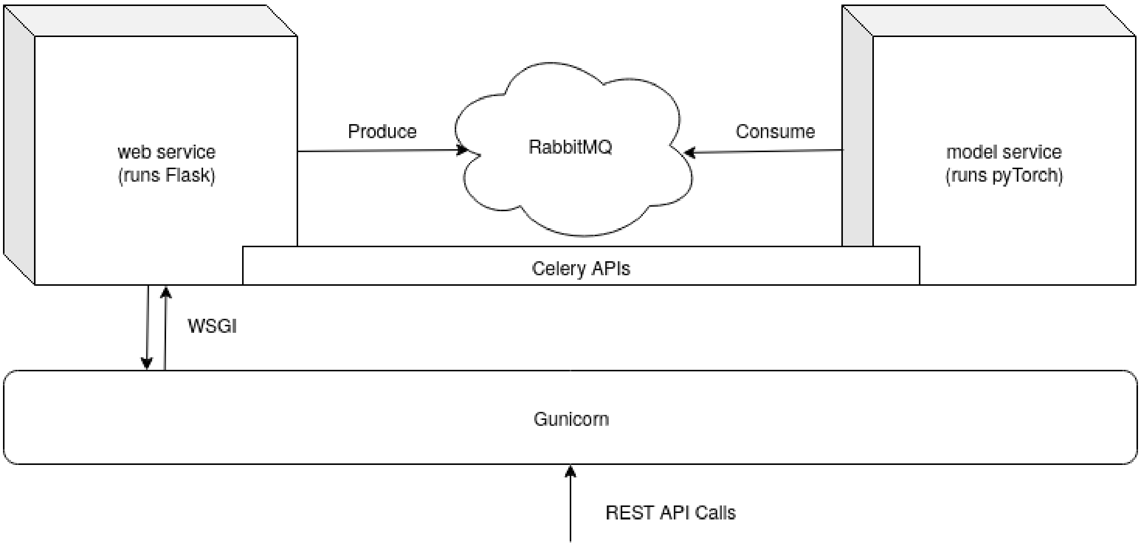
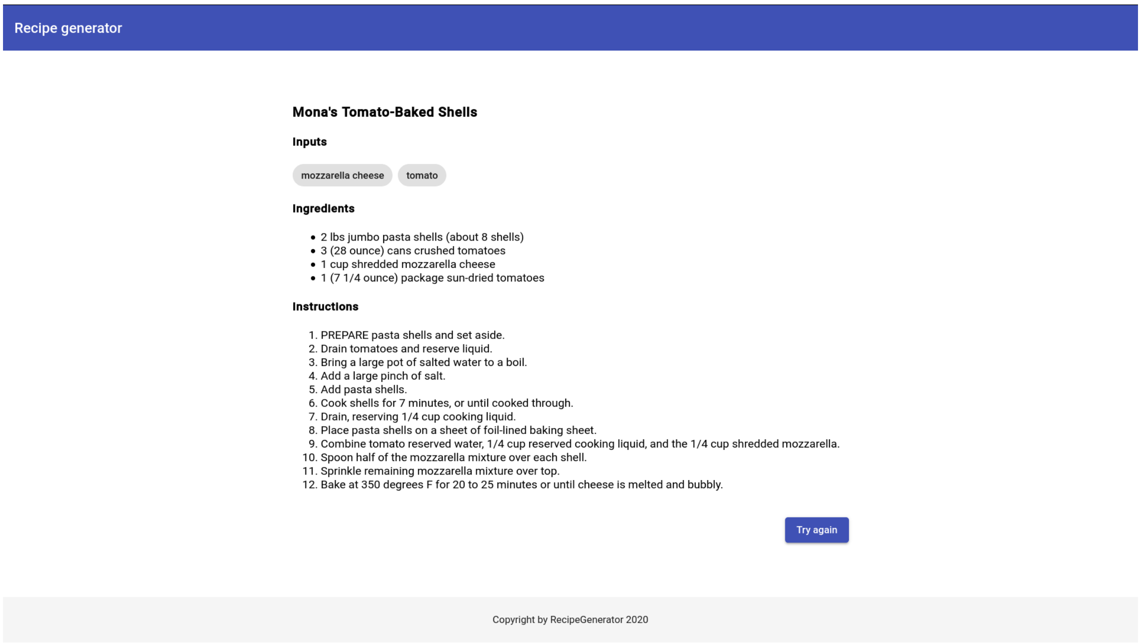


Figure 7.1: The backend architecture schema



Mona's Tomato-Baked Shells

Inputs

mozzarella cheese   tomato

Ingredients

- 2 lbs jumbo pasta shells (about 8 shells)
- 3 (28 ounce) cans crushed tomatoes
- 1 cup shredded mozzarella cheese
- 1 (7 1/4 ounce) package sun-dried tomatoes

Instructions

1. PREPARE pasta shells and set aside.
2. Drain tomatoes and reserve liquid.
3. Bring a large pot of salted water to a boil.
4. Add a large pinch of salt.
5. Add pasta shells.
6. Cook shells for 7 minutes, or until cooked through.
7. Drain, reserving 1/4 cup cooking liquid.
8. Place pasta shells on a sheet of foil-lined baking sheet.
9. Combine tomato reserved water, 1/4 cup reserved cooking liquid, and the 1/4 cup shredded mozzarella.
10. Spoon half of the mozzarella mixture over each shell.
11. Sprinkle remaining mozzarella mixture over top.
12. Bake at 350 degrees F for 20 to 25 minutes or until cheese is melted and bubbly.

Copyright by RecipeGenerator 2020

Figure 7.2: Example view from user interface

8 Conclusions

In this work, we propose a computational NLP solution based on the modern machine learning techniques. The part of this process was passing through the entire data science pipeline.

The data acquisition step used modern web scraping approaches on a big scale, which allowed this work to produce, to the best of our knowledge, **the biggest dataset of cooking recipes publicily available.** The data was then analysed in order to extract food entities from the ingredients lists. This step's needs induced creation of **the only publicily available, precise, food products classifier for SpaCy framework**.

In the next step of data processing pipeline, the collected dataset, together with NER classification result was used to train the **state of the art Transformer model utilizing GPT-2 architecture created by OpenAI.** While the model architecture was used without any changes, the text of the recipes was **combined with novel approach using control tokens in order to induce better recipe generation based on ingredients coming from user input.** The model was then embedded into a **full proof of concept product architecture, consisting of modern backend and frontend stack, with scalability and ease of deployment in mind**. The product allows users to generate new cooking recipes based on their choice of ingredients.

The final part of this work was **an exploratory work in the area of model-generated text evaluation methods comparison and tests for this specific use case**. In the multistage approach, the evaluation process made use of both **carefully selected automatic evaluation metrics and human insight** to assure and compare model performance for different training parameters. Those were successfully assured and **an interesting insight about the evaluation possibilities was compiled and presented**.

The key project objectives, detailed in Chapter 1, were fulfilled. It is expected that some of the interesting results of this work may be further investigated.

Notably, **further work on constraining generative models using control tokens and other methods may lead to interesting results**. While the PyTorch training on Cloud TPU was unsuccessful, after contacting Google and providing feedback, **the project was invited to use TPUs for another 60 days to investigate the issue.** The training on faster accelerators may allow this project to train **larger GPT-2 versions, which may improve quality of generated recipes even further**. The exploratory work on validation methods **may also be continued** as an evergreen effort in NLP to detect and validate text quality and recognize, what is the adequate measure that distinguishes human writing intelligence from the automatically generated text.

References

[1] Yonghui Wu et al. ‘Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation’. In: *CoRR* abs/1609.08144 (2016). arXiv: `1609.08144`. URL: `http://arxiv.org/abs/1609.08144`.

[2] Jay Allammar. *Visualizing machine learning one concept at a time*. URL: `https://jalammar.github.io` (visited on 27/12/2019).

[3] *AllenAI tpu_pretrain code*. URL: `https://github.com/allenai/tpu_pretrain` (visited on 26/12/2019).

[4] *Angular*. URL: `https://angular.io/` (visited on 25/01/2020).

[5] *Angular Material*. URL: `https://material.angular.io/` (visited on 25/01/2020).

[6] Dzmitry Bahdanau, Kyunghyun Cho and Yoshua Bengio. ‘Neural Machine Translation by Jointly Learning to Align and Translate’. In: *CoRR* abs/1409.0473 (2014).

[7] *Beautiful Soup Documentation*. URL: `https://www.crummy.com/software/BeautifulSoup/bs4/doc/` (visited on 25/01/2020).

[8] J.S. Chall and E. Dale. *Readability revisited: the new Dale-Chall readability formula*. Brookline Books, 1995. ISBN: 9781571290083. URL: `https://books.google.pl/books?id=2nbuAAAAMAAJ`.

[9] *Common machine translation metrics*. URL: `https://github.com/gcunhase/NLPMetrics` (visited on 24/01/2020).

[10] *Cosine Similarity in Python*. URL: `https://www.machinelearningplus.com/nlp/cosine-similarity/` (visited on 18/11/2019).

[11] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: `https://dask.org`.

[12] Jacob Devlin et al. ‘BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding’. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran and Thamar Solorio. Association for Computational Linguistics, 2019, pp. 4171–4186. ISBN: 978-1-950737-13-0.

[13] *Docker Compose documentation*. URL: `https://docs.docker.com/compose/` (visited on 02/12/2019).

[14] *Docker documentation*. URL: `https://docs.docker.com/engine/docker-overview/` (visited on 02/12/2019).

[15] R.F. Flesch. *How to write plain English: a book for lawyers and consumers*. Harper & Row, 1979. ISBN: 9780060112783. URL: `https://books.google.pl/books?id=-kpZAAAAMAAJ`.

[16] Felix A. Gers and Jürgen Schmidhuber. ‘LSTM recurrent networks learn simple context-free and context-sensitive languages’. In: *IEEE Trans. Neural Networks* 12.6 (2001), pp. 1333–1340. DOI: `10.1109/72.963769`.

[17] *Google Cloud Platform*. URL: `https://cloud.google.com/` (visited on 25/01/2020).

[18] R. Gunning. *The Technique of Clear Writing*. McGraw-Hill, 1952. URL: https://books.google.pl/books?id=ofI0AAAAMAAJ.

[19] Isabelle Guyon et al., eds. *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. 2017.

[20] Jeremy Howard and Sebastian Ruder. 'Universal Language Model Fine-tuning for Text Classification'. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. Ed. by Iryna Gurevych and Yusuke Miyao. Association for Computational Linguistics, 2018, pp. 328–339. ISBN: 978-1-948087-32-2. DOI: 10.18653/v1/P18-1031.

[21] *Jupyter Notebook*. URL: https://jupyter.org/ (visited on 25/01/2020).

[22] *Kaggle*. URL: https://www.kaggle.com/ (visited on 25/01/2020).

[23] Nitish Shirish Keskar et al. 'CTRL: A Conditional Transformer Language Model for Controllable Generation'. In: *CoRR* abs/1909.05858 (2019). arXiv: 1909.05858. URL: http://arxiv.org/abs/1909.05858.

[24] *LimeSurvey*. URL: https://www.limesurvey.org/ (visited on 27/01/2020).

[25] Thang Luong, Hieu Pham and Christopher D. Manning. 'Effective Approaches to Attention-based Neural Machine Translation'. In: *EMNLP*. 2015.

[26] Javier Marin et al. 'Recipe1M+: A Dataset for Learning Cross-Modal Embeddings for Cooking Recipes and Food Images'. In: *IEEE Trans. Pattern Anal. Mach. Intell.* (2019).

[27] Javier Marı́n et al. 'Recipe1M: A Dataset for Learning Cross-Modal Embeddings for Cooking Recipes and Food Images'. In: *CoRR* abs/1810.06553 (2018). arXiv: 1810.06553.

[28] Wes McKinney. 'Data Structures for Statistical Computing in Python'. In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56.

[29] Harry G. McLaughlin. 'SMOG grading - a new readability formula'. In: *Journal of Reading* (May 1969), pp. 639–646.

[30] Tomas Mikolov et al. 'Efficient Estimation of Word Representations in Vector Space'. In: *CoRR* abs/1301.3781 (2013).

[31] *Mongo DB container*. URL: https://hub.docker.com/_/mongo (visited on 02/12/2019).

[32] *Mongo DB documentation*. URL: https://docs.mongodb.com/ (visited on 02/12/2019).

[33] *Mongo Express container*. URL: https://hub.docker.com/_/mongo-express (visited on 02/12/2019).

[34] *Mongo Express documentation*. URL: https://github.com/mongo-express/mongo-express#readme (visited on 02/12/2019).

[35] Manivannan Murugavel. *spacy-ner-annotator*. URL: https://github.com/ManivannanMurugavel/spacy-ner-annotator (visited on 29/12/2019).

[36] *Nginx container*. URL: https://hub.docker.com/_/nginx (visited on 02/12/2019).

[37] *Nginx documentation*. URL: https://docs.nginx.com/ (visited on 02/12/2019).

[38] *NLTK*. URL: https://www.nltk.org/ (visited on 25/01/2020).

[39] Jekaterina Novikova et al. 'Why We Need New Evaluation Metrics for NLG'. In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sept. 2017, pp. 2241–2252. DOI: 10.18653/v1/D17-1238. URL: https://www.aclweb.org/anthology/D17-1238.

[40] *OpenRecipes Project Repository*. URL: https://github.com/fictivekin/openrecipes (visited on 25/01/2020).

[41] Kishore Papineni et al. 'Bleu: a Method for Automatic Evaluation of Machine Translation'. In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://www.aclweb.org/anthology/P02-1040.

[42] F. Pedregosa et al. 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[43] Matthew E. Peters et al. 'Deep Contextualized Word Representations'. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*. Ed. by Marilyn A. Walker, Heng Ji and Amanda Stent. Association for Computational Linguistics, 2018, pp. 2227–2237. ISBN: 978-1-948087-27-8.

[44] *Python wrapper for LanguageTool*. URL: https://pypi.org/project/language-check/ (visited on 16/12/2019).

[45] *pyTorch XLA issue*. URL: https://github.com/pytorch/xla/issues/1326#issuecomment-568575014 (visited on 26/12/2019).

[46] *pytorch-tpu/transformers*. URL: https://github.com/pytorch-tpu/transformers (visited on 28/12/2019).

[47] Alec Radford et al. *"Improving Language Understanding by Generative Pre-Training"*. 2018. URL: https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf (visited on 31/10/2019).

[48] Alec Radford et al. *Language Models are Unsupervised Multitask Learners*. 2019. URL: https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf (visited on 31/10/2019).

[49] *Readability Index in Python(NLP)*. URL: https://www.geeksforgeeks.org/readability-index-pythonnlp/ (visited on 18/11/2019).

[50] Sebastian Ruder. *NLP's ImageNet moment has arrived*. 12th July 2018. URL: https://ruder.io/nlp-imagenet/ (visited on 31/10/2019).

[51] Amaia Salvador et al. ‘Inverse Cooking: Recipe Generation From Food Images’. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 10453–10462.

[52] Amaia Salvador et al. ‘Learning Cross-modal Embeddings for Cooking Recipes and Food Images’. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017.

[53] *Scrapy cluster documentation*. URL: `https://scrapy-cluster.readthedocs.io/en/latest/topics/introduction/index.html` (visited on 02/12/2019).

[54] *Scrapy documentation*. URL: `https://docs.scrapy.org/en/latest/` (visited on 02/12/2019).

[55] *Scrapyd documentation*. URL: `https://scrapyd.readthedocs.io/en/stable/` (visited on 02/12/2019).

[56] *spaCy*. URL: `https://spacy.io/` (visited on 05/01/2020).

[57] *TensorFlow Research Cloud*. URL: `https://www.tensorflow.org/tfrc/` (visited on 29/12/2019).

[58] *TextBlob*. URL: `https://textblob.readthedocs.io/` (visited on 17/01/2020).

[59] Ashish Vaswani et al. ‘Attention is All you Need’. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al. 2017, pp. 5998–6008.

[60] *Word error rate*. URL: `https://martin-thoma.com/word-error-rate-calculation/` (visited on 25/01/2020).

[61] *XLA*. URL: `https://www.tensorflow.org/xla` (visited on 25/01/2020).

[62] Zhilin Yang et al. ‘XLNet: Generalized Autoregressive Pretraining for Language Understanding’. In: *NeurIPS*. 2019.