Source: https://commons.wikimedia.org/wiki/File:Healthy_Food_-_Colourful_Fruit_and_Veg_-_501916991151.jpg

# Recipe Ingredient Classification

Udacity Machine Learning Engineer Nanodegree

Yeray García Quintana
Munich, Germany
Aug 2, 2021

linkedin.com/in/yeray-garcia/
github.com/y-garcia/

# Recipe Ingredient Classification

## Definition

### Project Overview

A problem that arises often in the Natural Language Processing (NLP) domain is the necessity to apply a category to certain parts of an unstructured text. One approach to solve this problem could be named-entity recognition (NER), where e.g. a name of a person, say "Albert Einstein",  is assigned a label, say “PERSON”. This is helpful in order to extract relevant information from an unstructured text and storing it in a structured manner.

One practical example could be found in the recipes app, I’m currently working on. One important feature of the app should be the ability to quickly add recipes without the need to type too much. So instead of offering a form to manually enter all the ingredients and preparation steps of a recipe found on the Internet by hand, the user can just enter the link and the app should be able to recognize e.g. what is an ingredient and what is a preparation step.

In this particular project I’m going to focus on recognizing the different parts of an ingredient. That is, if the recipe contains an ingredient described as “400 g Broccoli”, our machine learning algorithm should be able to divide it into its different parts, e.g. “400” for quantity, “g” for unit of measurement and “Broccoli” for ingredient’s name.

Since I already have some recipes (in german) in the app with their corresponding ingredients stored with such a structure in a relational database, I’m going to use that data as my training data. Below you can see an excerpt of the training data which contains 573 rows in total:

| ingredient | quantity | unit | name |
|---|---|---|---|
| 2.00 Becher süße Sahne | 2 | Becher | süße Sahne |
| 6.00 Blätter Basilikum | 6 | Blätter | Basilikum |
| 4.00 Blätter Petersilie | 4 | Blätter | Petersilie |
| 0.52 Bund Minze | 0.52 | Bund | Minze |
| 0.52 Bund Petersilie | 0.52 | Bund | Petersilie |
| 1.00 Bund Frühlingszwiebeln | 1 | Bund | Frühlingszwiebeln |
| 0.52 Bund Petersilie | 0.52 | Bund | Petersilie |
| ... | ... | ... | ... |

# Problem statement

Given the plain text of a recipe's ingredient, we want to find and label its parts: quantity, unit of measurement, and ingredient name.

This is a classification problem where the model takes plain text as input and produces a list of words contained in that text and classifies them as corresponding to a particular entity.

For example, given following list of ingredients in plain text

Ingredients:
- 4 large eggs
- 1/4 cup milk
- pinch salt
- pinch pepper
- 2 tsp. butter

the model should extract the information in this manner (or similar):

Ingredients = [eggs, milk, salt, pepper, butter]
Quantities = [4, 1/4, null, null, 2]
Units = [null, cup, pinch, pinch, tsp.]

This problem can be solved with named-entity recognition. This is a supervised learning problem, so I will need labeled data consisting of plain text representing the full ingredient description and the mapping of the relevant words to their corresponding entity (quantity, unit, name).

# Metrics

Different types of metrics can be used:

1. accuracy:
    - **Formula**: (true positives + true negatives) / total
    - **In other words**: correctly predicted / total
    - **Question accuracy answers**: How many selected items were correctly categorized?
2. recall:
    - **Formula**: true positives / (true positives + false negatives)
    - **In other words**: words correctly identified as entities / all words that are entities
    - **Question recall answers**: How many relevant items are selected?
3. precision:
    - **Formula**: true positives / (true positives + false positives)
    - **In other words**: words correctly identified as entities / words correctly and incorrectly identified as entities
    - **Question precision answers**: How many selected items are relevant?
4. F1-score:

    - **Formula**: 2 · precision · recall / (precision + recall)
    - **In other words**: It tries to optimize both precision and recall. F1-score 1 means perfect precision and recall. F1-score 0 means either precision or recall is zero.

**Accuracy** can be misleading if we have imbalanced data.

**Recall** is in our case more important than **precision**, since we want to catch all ingredients, and we don't care if some non-ingredients are marked as such, we can just delete them later in the app with a single click of a button. On the other hand not recognizing ingredients would force us to type them manually.

We still don't want to have too many non-ingredients in our ingredient's list though, so we could use the **F1-score** which combines recall and precision as a metric for our problem.

Another approach could be to use precision as a **satisficing metric** and recall as an **optimizing metric**, meaning that we could try to optimize recall to the highest degree possible and having precision to be just good enough.

For this problem I am going to use the **F1-score** as a metric, since it optimizes both for recall and precision and we want both to be reasonably good.

# Analysis

## Data Exploration and Visualization

Since I already have a list of recipes (in german), I will use those ingredients and label all parts myself in a semiautomatic way, since they are already stored in a structured manner in the database.

Below you can see an excerpt of the data which contains 573 rows in total:

| ingredient | quantity | unit | name |
|---|---|---|---|
| 2.00 Becher süße Sahne | 2 | Becher | süße Sahne |
| 6.00 Blätter Basilikum | 6 | Blätter | Basilikum |
| 4.00 Blätter Petersilie | 4 | Blätter | Petersilie |
| 0.52 Bund Minze | 0.52 | Bund | Minze |
| 0.52 Bund Petersilie | 0.52 | Bund | Petersilie |
| 1.00 Bund Frühlingszwiebeln | 1 | Bund | Frühlingszwiebeln |
| 0.52 Bund Petersilie | 0.52 | Bund | Petersilie |
| 1.00 Bund Frühlingszwiebeln | 1 | Bund | Frühlingszwiebeln |
| 1.00 Bund Koriander | 1 | Bund | Koriander |
| 0.52 Bund Petersilie | 0.52 | Bund | Petersilie |
| ... | ... | ... | ... |

### Column 'ingredient'

The first column **ingredient** contains the ingredient plain text as we would find it in a recipe description. This plain text contains normally three pieces of information: the quantity, the unit of measurement and the ingredient name. These parts were extracted into the following three columns.

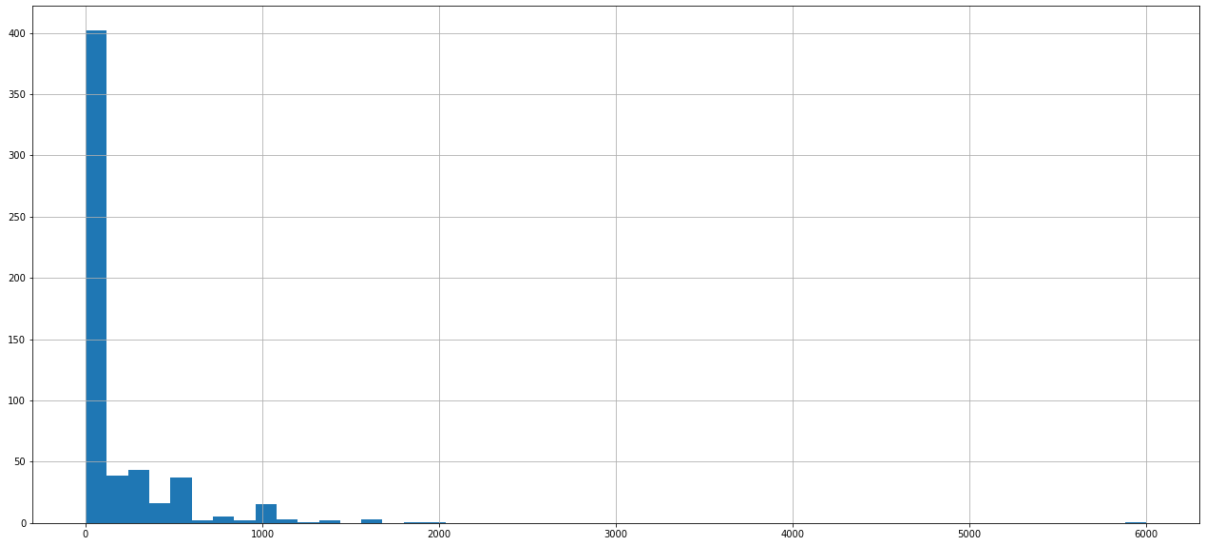
## Column 'quantity'

The second column **quantity** contains the numeric amount and is somewhat skewed as shown below so it may need some adjustment in the training phase.

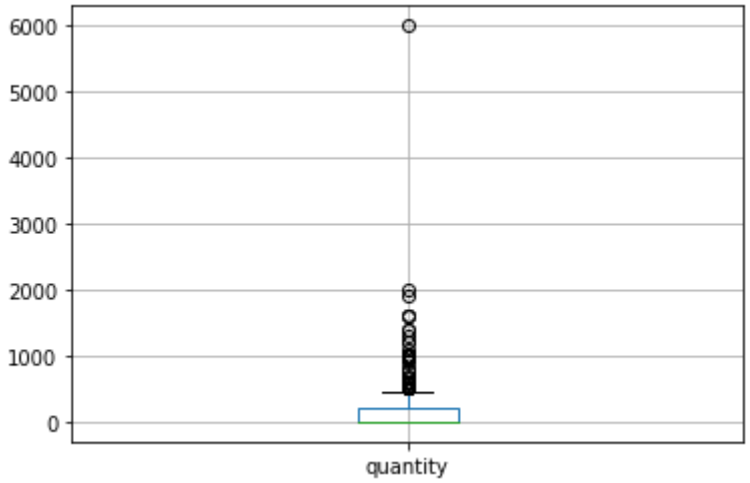
### Statistics

| | |
|---|---|
| **count** | 573 |
| **mean** | 165.73 |
| **std** | 383.72 |
| **min** | 0.12 |
| **25%** | 1.52 |
| **50%** | 4.00 |
| **75%** | 200.00 |
| **max** | 6000.00 |

### Histogram



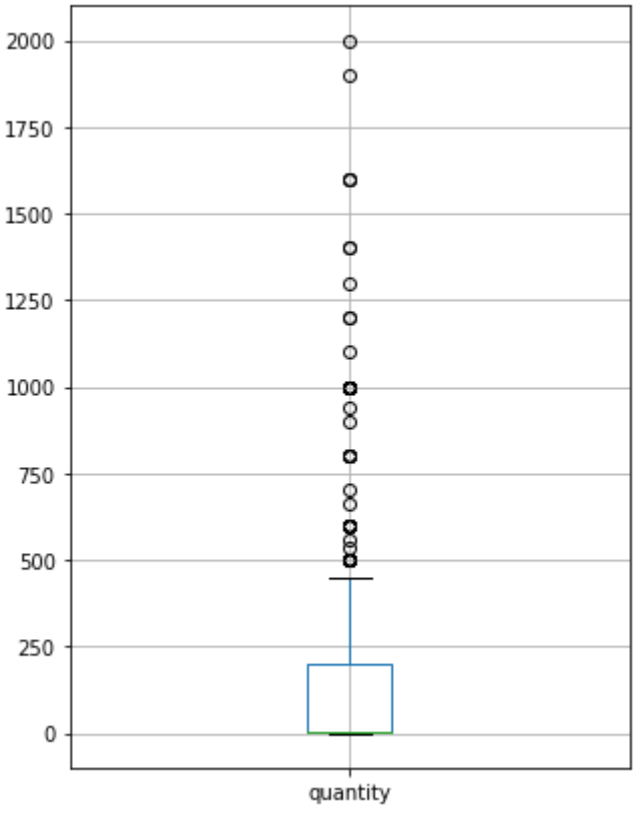
Boxplot



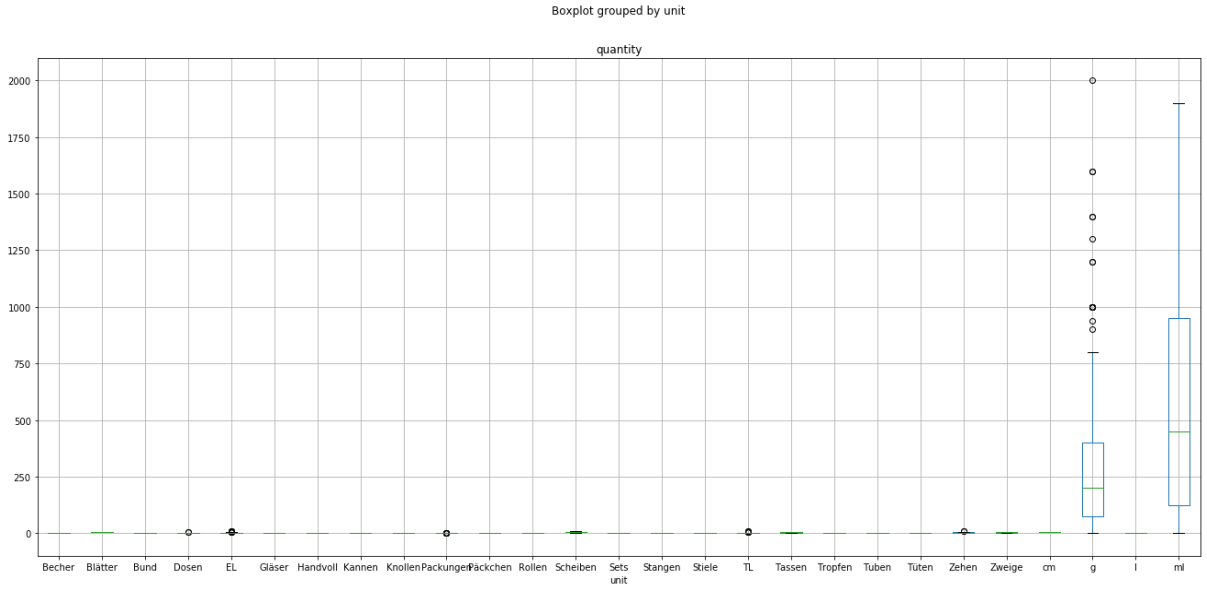

In the boxplot above we can see we have some outliers. Let's dive into the data to investigate that further.

| ingredient | quantity | unit | name |
|---|---|---|---|
| 6000.00 ml Berchtesgadener frische Milch 1,5% | 6000.0 | ml | Berchtesgadener frische Milch 1,5% |

In this particular recipe 6 liters of milk are needed, which seems somewhat off. Let's plot the boxplot without the outlier:



Let’s now plot the boxplot grouped by unit of measurement.




We can see that most units of measurement (like tablespoon=EL or teaspoon=TL) have a range of values close to 0 (like 0.5, 1, 2, 3), whilst the units gram and milliliter have a range between 0 and 2000, which is expected.

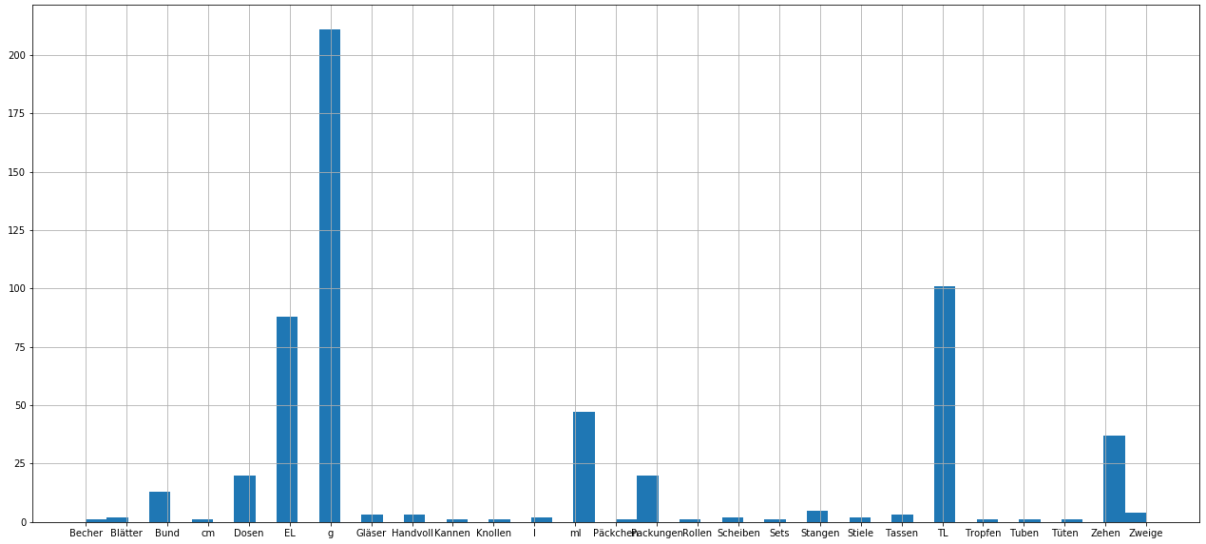
## Column 'unit'

The third column **unit** contains the different types of units of measurement for the ingredients and contains 27 unique values as shown below.

### Statistics

| | |
|---|---|
| **count** | 573 |
| **unique** | 27 |
| **top** | g |
| **freq** | 211 |

### Histogram




Here we can see **grams** (g) is the most common unit of measurement, followed by **tablespoon** (EL), **teaspoon** (TL) and **milliliters** (ml). The distribution makes sense, since these are units commonly used for recipes. One that stands out is **cloves** (Zehen), which indicates that garlic is probably used in many recipes.

## Column ‘name’

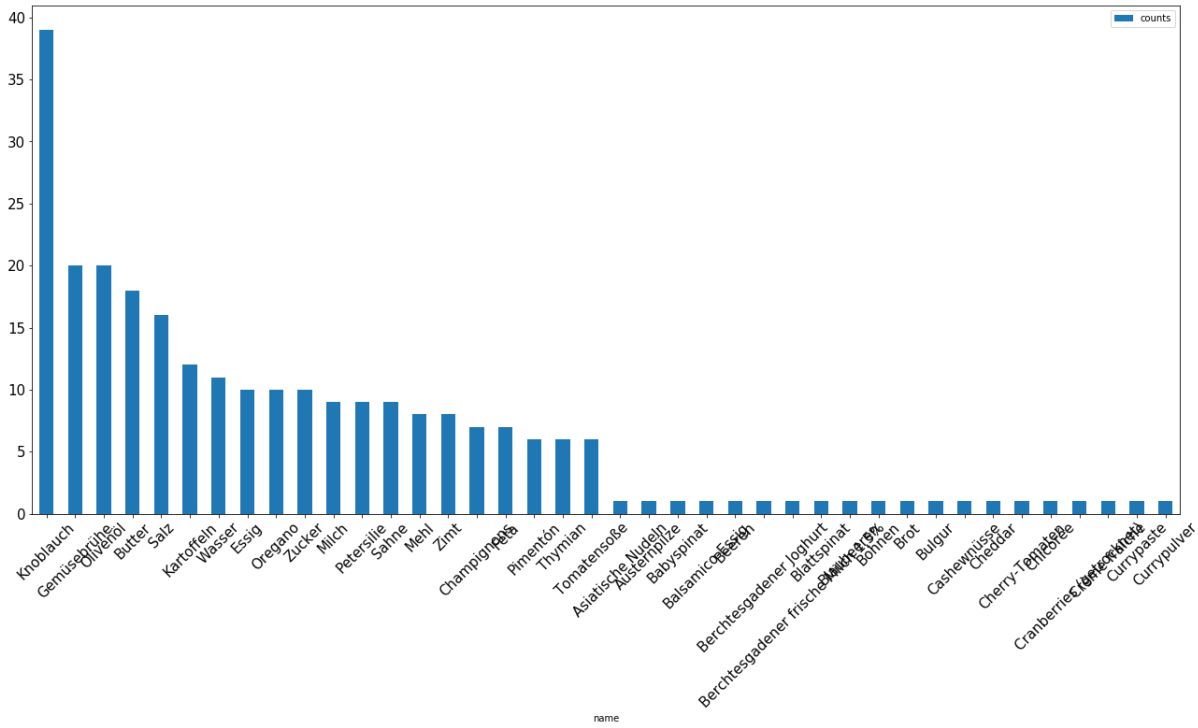
Finally we have the **name** column which contains the name of the ingredients without the quantity and unit. The column contains 230 unique values.

### Statistics

| | |
|---|---|
| **count** | 573 |
| **unique** | 230 |
| **top** | Knoblauch (garlic) |
| **freq** | 39 |

### Histogram

The histogram shown below contains the top and bottom 20 ingredients by frequency. The distribution shows that common ingredients such as garlic (=Knoblauch) are used in many recipes, whilst other less common ones like Currypowder (=Currypulver) only appear once.




## Algorithms and Techniques

This problem can be solved with some sort of information extraction technique. One relevant subdomain could be entity extraction, which is the task of identifying and categorizing relevant concepts (called entities, e.g. companies, people, places, ...) in unstructured data such as text. There are different ways of tackling this task:

**Regex-based approach**: consists in defining a regular expression to find an entity which is always represented using the same structure, e.g. phone numbers, email addresses, credit card numbers.

**Dictionary-based approach**: consists in looking for entities in the text from a list of known entities (dictionary), e.g. units, months, employees from a company.

**Linguistics-based approach**: consists in using linguistic features such as phrase structure, punctuation, type of word (noun vs verb), words used in the near vicinity (context), in order to extract relevant entities.

**Machine Learning and statistical approach**: consists in using models and algorithms trained on labeled data in order to identify and categorize entities in a text. Some examples of these models and algorithms are Conditional random fields (CRF), Long short-term Memory Recurrent Neural Networks (LSTM-RNN), and pretrained models such as ELMo and BERT.

In this project I tackled the problem with the machine learning approach. In this case we are dealing with a supervised learning problem with the goal of extracting entities from text.

For that I will need labeled data consisting of:

- Plain text representing the full ingredient description (e.g. 100 g Flour)
- A mapping of the relevant words to their corresponding entity:
    - Quantity: 100
    - Unit: g
    - Name: Flour

For the algorithm I chose named-entity recognition using spaCy, which is a free open-source library for Natural Language Processing in Python. I decided to use spaCy for various reasons:

1. It supports the german language which matches the language of the training data.
2. It comes with pretrained models that can be used for transfer learning if needed.
3. It comes with a component for named-entity recognition which is suited for this problem, since we are trying to label entities in unstructured text.

## Benchmark

I used following source as a benchmark for my problem:

- https://www.depends-on-the-definition.com/identify-ingredients-with-neural-networks/

In that article the author tries to identify ingredients in cooking recipes. For that he uses data from the Kaggle German Recipes Dataset. The data contains recipes from the well-known german recipe website chefkoch.de and consists of a list of ingredients and a text description of the instructions for every recipe. The ingredients list was used to label the words in the instructions. If a word in the instructions was found in the ingredients list, it was labeled with a 1, otherwise it was labeled with a 0. After that an LSTM-based model was trained to identify ingredients in plain text.

After running the benchmark model on my own data I could achieve following performance:

- precision: 0.26%
- recall: 99.83%
- f1-score: 0.53%

Taking into account that my data doesn't resemble with a high degree the data used to train the benchmark model this is a good result. On the other hand the benchmark model seems to optimize for recall. In my case I will try to not only achieve high recall, but also a high precision, hopefully improving the overall f1-score, which is a combination of both metrics.

# Methodology

## Data Preprocessing

The data was extracted from a recipes app that I have developed myself, so it was already very structured and clean. No abnormalities were found, so nothing was discarded.

Since we are using spaCy for training the named-entity recognition model we have to adapt the data to suit the required format.

A training example for the spaCy algorithm needs to be in following format:

*(text, list_of_entities)*

Here is an example:

*('100 g Flour', {'entities': [(0, 3, 'Quantity'), (4, 5, 'Unit'), (6, 11, 'Flour')]})*

That is, a training example consists of a tuple of the text we want to extract entities from and the list of the entities it contains, including the position of start and end characters.

All training examples are then collected into 3 lists:

- A training set, corresponding 60% of the data
- A validation set, corresponding 20% of the data
- A test set, corresponding 20% of the data

## Implementation

The model is trained following the recommendations in https://v2.spacy.io/usage/training.

We start with an empty model and a single named-entity recognition (NER) pipeline component.
All 3 labels (quantiy, unit, name) are added to the NER pipeline to be recognized by the model.

Some hyperparameters are defined in advance like the number of iterations, batch size and dropout rate.

After that we loop through the training set, shuffling it before every iteration. The training set is then divided into batches, for which the **nlp.update()** function is run with the texts and annotations passed as parameters, thus updating the internal weights of the model.

The loss is calculated and tracked for each iteration. It is then plotted into a graph and analyzed to make sure that it gets smaller over time.

## Complications

One of the first complications that occurred whilst training the model was the fact that it was not possible to label a word twice. This occurred for example when labelling ingredients that were composed of more than one word.

Let's say there was an ingredient called "crushed tomatoes" and another one called just "tomatoes". The spaCy matcher would encounter a text of the form "100 g crushed tomatoes" and would try to label "crushed tomatoes" as an ingredient and at the same time it would try to label "tomatoes" again as an ingredient. Hence the word "tomatoes" was labeled as an entity twice, which would make spaCy throw an exception.

To solve this, I implemented the function "**split_words(series)**" with following logic:

- iterate through all known ingredient names, e.g.:
    - ["crushed tomatoes", "tomatoes", "garlic", ...]
- split them into words, e.g.:
    - ["crushed", "tomatoes", "tomatoes", "garlic", ...]
- discard any duplicates
    - ["crushed", "tomatoes", "garlic", ...]

Using our example above, the spaCy matcher would encounter a text of the form "100 g crushed tomatoes" and would label "100" as quantity, "g" as unit, "crushed" as "ingredient" and "tomatoes" as ingredient again.

Another complication was trying to get the benchmark to work. I encountered following problems:

1. The model took quite some time to train.
2. The data used to train the model had a different format as my own data.
3. No metrics were calculated in the original code.

The solution to the first problem involved two steps:

- Increasing the AWS notebook performace by using an instance of type "ml.t2.2xlarge" instead of "ml.t2.medium".
- Saving the model to S3 and load it from then on, instead of training it every time I run the notebook. For this I implemented a small **S3util** library to upload and download data from S3, which I then imported and uses in the benchmark notebook.

The solution to the second problem involved a little refactoring of the original code in order to apply the same preprocessing to my own data. The refactoring included extracting part of the code into the function "**format_labels()**" which creates a label vector as a series of 1's and 0's with the necessary padding at the end.

I then addressed the third problem. In order to compare the performance of the benchmark to my own model I needed to calculate some metrics. I implemented three functions to calculate the precision, recall and f1-score of the model's predictions of the training set, the test set and the data used to train my own model. This allowed me to compare the benchmark performance to my own model's performance. More on this later.

# Refinement

In order to choose the optimal hyperparameters for the model I followed the approach recommended on the book Machine Learning Yearning by Andrew Ng:

1. Divide the data into 3 sets: Training, dev and test set.
2. Train the model on the training set using different hyperpameter combinations.
3. Calculate the error rate on the training set and dev set and calculate bias and variance from that.
4. Manually choose the model based on the bias and variance, both should be low to avoid overfitting and underfitting.

## 1. Data sets

Although the dataset is relatively small I decided to work with all three sets - **training, dev and test set** - in order to learn how to use them to improve the model. I opted for a **60% - 20% - 20%** split. The most important thing is that all sets should have the **same distribution** which I garanteed by shuffling the original data before splitting it.

## 2. Hyperparameters

After that I defined which hyperparameters to optimize. After a couple of training rounds I noticed that **10 iterations** were plenty to get a low enough loss value, so I fixed that parameter. I then defined some values for the hyperparameters **batch size** and **dropout**. The first one helps with better **conversion** and the second one helps with better **generalization**. I then looped through the training set using all hyperparameter combinations.

## 3. Bias and variance

For each hyperparameter combination I then calculated the **f1 score** on the **training set** the model was trained on and on the **dev set**, which the model hadn't seen yet. One can then informally define the error rate on the training set as the **bias** and the difference between the dev error rate and the training error rate as the **variance**. Both bias and variance should be low in order to have a model that performs well and doesn't overfit.

## 4. Choose model

I then created a dataframe where I could visualize the bias and variance for each combination of hyperparameters in a tabular manner. After sorting the dataframe by those metrics I then chose the model with the lowest values.

# Result

## Model Evaluation and Validation

After **training** the model using the training set and **optimizing** it using the dev set, now it is time to **evaluate** it using the test set. The test set contains data with the same distribution as the other sets but hasn't been seen by the model nor hasn't been used to optimize it so it is a good proxy for real world data.

First I wanted to look at a couple of examples myself. I implemented the function **get_entities(text)** which returns an array of entities found in the text passed to the function. I then iterated through 10 random examples from the test set and printed the results in a human-readable format. The model seems to do a good job.

To validate that statement I then calculated the **f1 score** on the **test set**. My model achieved an **f1 score of over 99%**, meaning it does well on **recall** and **prediction**.

## Justification

Comparing it to the benchmark, which achieved a **recall of 99.83%** but a **precision of 0.26%** and hence an **f1 score of 0.53%**, my model seems to be better suited for my problem since it also optimizes for precision achieving a much higher f1 score, which is the metric we were trying to optimize.

Taking into consideration that my model does well on the test set, which is data the model hasn't seen before, I would argue this model is suited for this problem and quite robust for this type of data.