COMP3074 Human-AI Interaction
Lab 3: Text classification

Jérémie Clos

October 2023

Contents

1 Introduction

This week's lab focusses on using the bag-of-words representation in order to build a **classifier**. In order to do so, we will introduce a new important library in the Python machine learning ecosystem: Scikit-Learn. Scikit-Learn embeds a lot of the processing necessary to build a classifier in an easy-to-use package.

2 The tools of the trade

If you are working on your own machine, you may not yet have Scikit-Learn. Don't worry, it is extremely easy to install. Follow the instructions on [the official website](https://scikit-learn.org/stable/install.html) to get started.

3 Bag of words

3.1 Building a bag-of-words model

A bag-of-words model is a simplistic representation of documents that assumes that the documents are just multi-sets of words. Let us take for example the first sentence of Mary Shelley's Frankenstein:

You will rejoice to hear that no disaster has accompanied the commencement of an enterprise which you have regarded with such evil forebodings.

After tokenising and removing stop words, the resulting set of tokens ends up being:

```
['you', 'rejoice', 'hear', 'disaster', 'accompanied', 'commencement', 'enterprise
    ', 'regarded', 'evil', 'forebodings']
```

After removing punctuation and stemming each term, the bag-of-words representation of this passage would be a list of those tokens, associated with their frequency in the document (here a sentence):

word (stemmed)	**frequency**
you	2
rejoic	1
hear	1
disast	1
accompani	1
commenc	1
enterpris	1
regard	1
evil	1
forebod	1

Let's have a look at another passage from the same novel:

Six years have passed since I resolved on my present undertaking. I can, even now, remember the hour from which I dedicated myself to this great enterprise.

Putting it through the same treatment, we end up with the following tokens:

```
['six', 'years', 'passed', 'since', 'i', 'resolved', 'present', 'undertaking', 'i
    ', 'even', 'remember', 'hour', 'i', 'dedicated', 'great', 'enterprise]
```

The cool thing now is that we can stem those tokens and add them to the bag-of-words model we had computed for our first passage:

Token	Passage 1	Passage 2
you	2	0
rejoic	1	0
hear	1	0
disast	1	0
accompani	1	0
commenc	1	0
enterpris	1	1
regard	1	0
evil	1	0
forebod	1	0
six	0	1
year	0	1
pass	0	1
sinc	0	1
i	0	3
resolv	0	1
present	0	1
undertak	0	1
even	0	1
rememb	0	1
hour	0	1
dedic	0	1
great	0	1

You can see that the frequency of a lot of those tokens is 0 because they do not appear in one of the two documents. However, representing documents this way allows us to have a common basis on which to compare documents later on. We call the list of tokens used to build our bag-of-words model the **vocabulary** of the model. The table above is usually named a Term-Document (or Document-Term) matrix.

3.2 Term weighting

The bag-of-words model is extremely powerful and, bar deep learning methods, the standard way of analysing text documents. However it runs into an issue when encoding documents of different lengths: a very long novel for example would have high frequencies in all terms, while a smaller novel would have very low term frequencies all throughout. This can become a problem when computing the similarity between documents, because those large documents end up being similar to all documents due to their sheer size. This is where term weighting methods come into play. The most common term weighting methods are **binary weighting**, **logarithmic weighting** and **tf-idf weighting**. We will go in more details in this in future labs.

3.3 Binary weighting

Each term `t` is either present or absent from a document `d`. This has the advantage of putting large and small documents on the same footing.

3.4 Log-frequency weighting

Each term `t` is weighted by a function of the raw frequency in the document `d`. This solves the main drawback of frequency weighting by log-scaling term frequencies, which means that even

a very frequent term will never be more than a few times as important as a less frequent one.

3.5 TF-IDF

For TF-IDF weighting, each term is weighted using its log frequency multiplied by the inverse document frequency of that term (the right-hand part of the equation) where N is the number of documents in the text collection, and n is the number of documents that contain that term. The inverse document frequency acts as proxy measure of the discriminative power of a term, so that a term that is present in almost every single document will have a low weight. This is a handy way to make sure that some words like "the" do not overpower other more meaningful words.

4 Pre-processing the data

Let's go back to Gutenberg and download a set of short stories from different countries, for the rest of the lab.

```
from urllib import request

doc_urls = {
    "Russia": "http://www.gutenberg.org/cache/epub/13437/pg13437.txt",
    "France": "http://www.gutenberg.org/cache/epub/10577/pg10577.txt",
    "England": "http://www.gutenberg.org/cache/epub/10135/pg10135.txt",
    "USA": "http://www.gutenberg.org/cache/epub/10947/pg10947.txt",
    "Spain": "http://www.gutenberg.org/cache/epub/9987/pg9987.txt",
    "Scandinavia": "http://www.gutenberg.org/cache/epub/5336/pg5336.txt",
    "Iceland": "http://www.gutenberg.org/cache/epub/5603/pg5603.txt"
}

documents = {}

for country in doc_urls.keys():
    content = request.urlopen(doc_urls[country]).read().decode('utf8', errors='
    ignore')
    documents[country] = content
```

4.1 Simple counts

Scikit-Learn gives us access to its own functions for tokenising, filtering, etc. We could work around them using the NLTK functions as they achieve the same thing, but since we want to use Scikit-Learn for the classification task, it is better to use the tool through the entire pipeline as it reduces the need for writing code translating from one library to the other. The simplest function is the CountVectorizer, which takes care of tokenising, filtering (stopwords and low-frequency words) and finally counting terms in documents.

```
from urllib import request
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer

doc_urls = {
    "Russia": "http://www.gutenberg.org/cache/epub/13437/pg13437.txt",
    "France": "http://www.gutenberg.org/cache/epub/10577/pg10577.txt",
    "England": "http://www.gutenberg.org/cache/epub/10135/pg10135.txt",
    "USA": "http://www.gutenberg.org/cache/epub/10947/pg10947.txt",
    "Spain": "http://www.gutenberg.org/cache/epub/9987/pg9987.txt",
    "Scandinavia": "http://www.gutenberg.org/cache/epub/5336/pg5336.txt",
    "Iceland": "http://www.gutenberg.org/cache/epub/5603/pg5603.txt"
}

```

```
documents = {}

for country in doc_urls.keys():
    content = request.urlopen(doc_urls[country]).read().decode('utf8', errors='
    ignore')
    documents[country] = content

all_text = documents.values()

count_vect = CountVectorizer(stop_words=stopwords.words('english'))
X_train_counts = count_vect.fit_transform(all_text)

print(X_train_counts.shape)
```

The `fit_transform` function feeds the documents in the `CountVectorizer`, which returns a term-document matrix. When we print its shape, it returns a terrifying `(7, 24128)`. That means that we are working with a vocabulary of 24,128 terms, over 7 documents. We are working in a 24,128 dimensional space. Take that, physicists. For obvious reasons, I will not attempt to draw such space in this document - just imagine a 3D space and add 24,125 more dimensions.

The CountVectorizer can be instantiated with parameters to modify the vocabulary:

- `max_df`: maximum document frequency. Words which appear in more than `max_df` documents will not be considered. You can also provide a floating point number between 0 and 1, in which case it will filter based on percentage rather than raw number.
- `min_df`: minimum document frequency. Words which appear in fewer than `min_df` documents will not be considered. Similarly to `max_df`, you can provide a float between 0 and 1 to filter by percentage.
- `max_features`: the maximum number of terms in the vocabulary (cutting down the words with the lowest frequency).
- `vocabulary`: if you want to pre-specify the vocabulary you want to use.
- `binary`: can be True or False, and determines whether it will count the frequency of terms or treat them as binary feature (1 if the term is in the document, 0 otherwise).

If you wanted to remove the top 10% of documents in terms of document frequency as well as all terms present in fewer than 5 documents, you could do so by using the following command:

```
count_vect = CountVectorizer(stop_words=stopwords.words('english'), min_df=5,
    max_df=0.9)
```

Similarly, if you wanted to only keep the 1,000 most common unigrams:

```
count_vect = CountVectorizer(stop_words=stopwords.words('english'), max_features
    =1000)
```

Finally, the `ngram_range` parameter lets you change the tokenisation in order to analyse combinations of ngrams. A range of (1, 1) would be unigrams only, while a range of (2, 2) would be bigrams only. A range of (1, 2) would be both unigrams and bigrams. Keep in mind that higher degree ngrams are computationally more expensive to compute and store and will make the size of your vocabulary explode. For example, the following command would yield a vocabulary of size 737,043.

```
count_vect = CountVectorizer(stop_words=stopwords.words('english'), ngram_range
    =(1,4))
```

4.2 Term weighting methods

The `TfidfTransformer` method is how Scikit-Learn lets us apply transformation functions on the simple counts of the `CountVectorizer`. It has many useful parameters to let us change the way it generates those term weights:

- `use_idf`: can be True or False, and specifies whether the term weights will be transformed using their IDF weight.
- `smooth_idf`: can be True or False, and specifies whether the IDF weights will be smoothed according to the following formula, which results in an IDF which is not as punitive as the standard formula.
- `sublinear_tf`: can be True or False, and specifies whether the term frequency is log-scaled. If True, the term frequency is as follows:

The following code listing gives an example of applying the TfidfTransformer to a CountVectorizer output (assuming the CountVectorizer has been applied to existing documents).

```
from sklearn.feature_extraction.text import TfidfTransformer

count_vect = CountVectorizer(stop_words=stopwords.words('english'))
X_train_counts = count_vect.fit_transform(all_text)
tf_transformer = TfidfTransformer(use_idf=True, sublinear_tf=True).fit(
    X_train_counts)
X_train_tf = tf_transformer.transform(X_train_counts)
print(X_train_tf.shape)
```

That's it, we have our document in a matrix form that (almost) any machine learning algorithm can understand and process. However, because we would like to do some experiments, we need to learn about the Train/Test paradigm.

4.3 A note about fit, transform, and fit_transform

Some of you might have noticed that we used fit, transform and `fit_transform` methods and might be wondering what the difference was (and there is one). We use `fit_transform` with the CountVectorizer because we want it to fit a model (the vocabulary) and transform it (return the counts) in the same operation. We could have done both things separately by calling fit first and then transform. For the TfidfTransformer we called them separately by first fitting the TfIdf model to the counts, and then transforming them in a second operation. At training time both are equivalent, and I encourage you to play around with those operations by yourself to become familiar with them. There will be a distinction when it comes to testing the classifiers that we will see in due course.

4.4 What about stemming/lemmatising?

You might be justifiably enraged to notice that I seem to have forgotten about stemming/lemmatising, after explaining their importance. I did not forget them. The CountVectorizer gives you the flexibility to bring your own stemmer quite easily. There is one additional argument that I have not discussed yet: `analyzer`. Analyzer lets you provide the following potential inputs:

- `analyzer = 'word'` is the default option, and means the features will be word n-grams.
- `analyzer = 'char'`: means that the features will be character n-grams. For example, "two cats" in character bigrams would be `[tw, wo, o_, _c, ca, at, ts]`.
- `analyzer = 'char_wb'`: means that the features will be character n-grams respecting word boundaries. For example, "two cats" in character bigrams would be `[tw, wo, ca, at, ts]`.

- any callable function: giving it a callable function will make it use it to extract a sequence of features from the input.

For example, the following script would stem the input before feeding it into the CountVectorizer vocabulary, using the Porter stemmer.

```
from sklearn.feature_extraction.text import CountVectorizer
from nltk.stem.snowball import PorterStemmer

p_stemmer = PorterStemmer()
analyzer = CountVectorizer().build_analyzer()

def stemmed_words(doc):
    return (p_stemmer.stem(w) for w in analyzer(doc))

stem_vectorizer = CountVectorizer(analyzer=stemmed_words)
print(stem_vectorizer.fit_transform(['This sentence should get seriously mangled
    in the stemming process, but it is fine.']))
print(stem_vectorizer.get_feature_names())
```

As you might expect, it would work the same way with the Lemmatiser.

5 The Train/Test paradigm

We are going to do some classification, and one of the most sacred practices in running machine learning experiments is that of the train/test paradigm. A classification model should be trained with a specific dataset (called the training set) and tested using another completely different data set (called the test set). This is to make sure that the model that we are training is learning **generalisable knowledge**, i.e. knowledge which can apply to data it has not seen at the time of the training. Imagine that you are building a chatbot. It would be a pretty poor chatbot if it could only reply to specific sentences that you have encoded in the bot. We would prefer that the chatbot can recognise a variety of inputs and try to respond as appropriately as it can. If we only benchmarked algorithms against data they were trained on, they could just memorise the answer and appear to be perfect, only to collapse miserably when deployed in the wild.

First, to conduct some proper training/testing, we need data. More data. Fortunately, it is the 21st century and there is data everywhere. This handy Github repository[1] keeps track of useful NLP datasets, many of which are too large for this lab. On Moodle you can find a sample of an existing dataset, the large movie review dataset extracted from IMDB. This dataset is divided into positive and negative reviews. The script below will help you turn that into one handy data structure, so that we can do a train/test split on it.

```
import os
from sklearn.model_selection import train_test_split

label_dir = {
    "positive": "data/positive",
    "negative": "data/negative"
}

data = []
labels = []

for label in label_dir.keys():
    for file in os.listdir(label_dir[label]):
        filepath = label_dir[label] + os.sep + file
        with open(filepath, encoding='utf8', errors='ignore', mode='r') as review
    :
            content = review.read()
```

[1] https://github.com/niderhoff/nlp-datasets

```
            data.append(content)
            labels.append(label)

X_train, X_test, y_train, y_test = train_test_split(data, labels, stratify=labels
    , test_size=0.25, random_state=1)
```

A common notation in machine learning for data and label is X and y. `X_train` thus refers to the data (text) of the training set, `y_train` to the labels of the training set (positive or negative) and similarly for `X_test` and `y_test`. The `train_test_split` method takes as input the data itself (in the form of an iterable data structure, like a list), the corresponding labels (a list of the same size), and the following arguments:

- `test_size`: refers to the proportion of samples that will be kept for testing. Here, 0.25 means that we will use 75% of the data for training our model, and 25% for testing.
- `stratify`: stratification refers to the process of keeping the proportion of classes the same in training and testing. So for example, if your data is 50% positive reviews and 50% negative reviews, stratification makes sure that this same ratio will be kept in your training set and in your test set.
- `random_state`: refers to the random seed. If you don't provide it, the randomisation will be different every time you split your data and therefore you will have slightly different results. If you fix it to any number, you will always randomly split the data in the same way, and therefore have the same result.

6 Preparing the Bag-of-Words and training

Now that we have split our data we can build our bag-of-word representation, by applying the CountVectoriser and TfidfTransformer to our training data stored in `X_train`.

```
import os
from sklearn.model_selection import train_test_split
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.linear_model import LogisticRegression

# import all data first
label_dir = {
    "positive": "data/positive",
    "negative": "data/negative"
}

data = []
labels = []

for label in label_dir.keys():
    for file in os.listdir(label_dir[label]):
        filepath = label_dir[label] + os.sep + file
        with open(filepath, encoding='utf8', errors='ignore', mode='r') as review
    :
            content = review.read()
            data.append(content)
            labels.append(label)

X_train, X_test, y_train, y_test = train_test_split(data, labels, stratify=labels
    , test_size=0.25, random_state=42)

count_vect = CountVectorizer(stop_words=stopwords.words('english'))
X_train_counts = count_vect.fit_transform(X_train)

```

```
tfidf_transformer = TfidfTransformer(use_idf=True, sublinear_tf=True).fit(X_train_counts)
X_train_tf = tfidf_transformer.transform(X_train_counts)

classifier = LogisticRegression(random_state=0).fit(X_train_tf, y_train)
```

In this code listing, we are training a Logistic Regression algorithm. It is a very common classification algorithm used in NLP because it is fairly simple, relies on strong statistical foundations, and is quite fast to train. The Scikit-Learn official documentation[2] explains its inner working in a very succinct way.

7 Evaluating the model

Now that we have trained our algorithm with the training data and training labels, we can see how it performs on unobserved data: our test set and its labels. We import the `accuracy_score` and `f1_score` methods from `sklearn.metrics` in order to measure how well (or badly) our algorithm is doing.

```
import os
from sklearn.model_selection import train_test_split
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix

# import all data first

label_dir = {
    "positive": "data/positive",
    "negative": "data/negative"
}

data = []
labels = []

for label in label_dir.keys():
    for file in os.listdir(label_dir[label]):
        filepath = label_dir[label] + os.sep + file
        with open(filepath, encoding='utf8', errors='ignore', mode='r') as review:
            content = review.read()
            data.append(content)
            labels.append(label)


X_train, X_test, y_train, y_test = train_test_split(data, labels, stratify=labels, test_size=0.25, random_state=42)

count_vect = CountVectorizer(stop_words=stopwords.words('english'))
X_train_counts = count_vect.fit_transform(X_train)

tfidf_transformer = TfidfTransformer(use_idf=True, sublinear_tf=True).fit(X_train_counts)
X_train_tf = tfidf_transformer.transform(X_train_counts)

clf = LogisticRegression(random_state=0).fit(X_train_tf, y_train)

X_new_counts = count_vect.transform(X_test)
X_new_tfidf = tfidf_transformer.transform(X_new_counts)

```

[2] https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

```
predicted = clf.predict(X_new_tfidf)

print(confusion_matrix(y_test, predicted))
print(accuracy_score(y_test, predicted))
print(f1_score(y_test, predicted, pos_label='positive'))
```

You should notice a few things in that script.

1. We only apply transform and not `fit_transform` on the `X_new_counts`. The reason for this is that we fit our TF-IDF scores on the training data and as such it is, sort of, part of the entire model. Now we only need to use those TF-IDF weights to transform the new data in the same format, so we just use transform.

2. We store the results in a variable called predicted. This will let us compute multiple quality metrics.

We then print three things: confusion matrix, accuracy, and f1-score. All are useful tools to see whether our classifier is working properly.

7.1 Confusion matrix

The confusion matrix is a handy way of visualising the behaviour of the classifier. Let's take the confusion matrix generated by my own run as an example (yours may vary, depending on the random seed you put in the train-test split and the classifier):

Actual / Predicted	**Negative**	**Positive**
Negative	114	11
Positive	12	113

We can see that out of the 125 negative samples, 114 were successfully classified as negative (true negative) and 11 were incorrectly classified as positive (false positive). Of the 125 positive samples, 113 were correctly classified as positive (true positive) and 12 were incorrectly classified as negative (false positive). This tells us that overall our algorithm is really not that bad - most samples were classified in the correct category. What would be neat, however, would be to have a single number to put on that correctness, so that we can compare it with other algorithms. Accuracy and F1-Score are such numbers.

7.2 Accuracy

Accuracy is the most straightforward metric to interpret:

$$\frac{\text{number of correct predictions}}{\text{number of predictions}}$$

Or, put in the terms defined above:

$$\frac{\text{true positives} + \text{true negatives}}{\text{true positives} + \text{true negatives} + \text{false positives} + \text{false negatives}}$$

Accuracy's strength is that it is representative of a quantity we **care** about: "if I release this algorithm in the wild, how many times is it going to get it right and how many times is it going to get it wrong?". However, that does not mean that it is a perfect measure in any way because it is highly sensitive to class imbalance issues. The best way to understand that intuitively is with a thought experiment: Imagine you are working for a big tech company and you are building a hate speech classifier, i.e., a classifier that will put any piece of text in one of the following categories: [hate speech, normal speech]. The issue here is that we can expect that

there is a lot more normal speech than there is hate speech. For the example, let us imagine that 1% of speech online is hate speech. What does that mean? It means that a classifier that always outputs "normal speech" will be correct 99% of the times - it will have an accuracy of 99%. But does that mean that it is doing its job well? Not at all! It is a terrible classifier. A "worse" classifier with 70% accuracy might be better if it is more balanced in its errors.

7.3 F1-Score

The F1-Score is another very common method to put a single number on the performance of an algorithm.

$$F_1 Score = \frac{\text{true positives}}{\text{true positives} + \frac{1}{2}(\text{false positives} + \text{false negatives})}$$

It is a balanced instantiation of the general F-Score, which is defined as:

$$F_\beta = \frac{(1+\beta^2) \cdot \text{true positives}}{(1+\beta^2) \cdot \text{true positives} + \beta^2 \cdot \text{true negatives} + \text{false positives}}$$

One particular feature of the F1-Score is that it punishes algorithms very badly if they ignore one class in favour of another. However, it is not as easy to interpret as accuracy.

8 Using the model

All of this is nice and well, but now that you have trained your model you might want to find a way to use it.

8.1 Saving and loading a model

Python has a handy set of functions to save and load objects called `pickle` and a newer way of doing things through `joblib`. Both work, but `joblib` is slightly better for large machine objects. Here we refer to any sort of instantiated data structure as an object. It could be a language model, a machine learning model, or anything you fancy.

With `pickle`:

```
import pickle
with open("filename.pickle", "wb") as f:
    pickle.dump(someobject, f)
with open("filename.pickle", "rb") as f:
    someobject = pickle.load(f)
```

With `joblib`:

```
from joblib import dump, load
dump(someobject, 'filename.joblib')
loaded_object = load('filename.joblib')
```

This is important if you are building an application that makes use of a complex structure like a classifier or an index because you don't want to have to re-compute it. **This code is not meant to run as is; it is provided as an example of two functions that can help you with your work.**

8.2 Making new predictions

Once you have trained your classifier, you can basically feed it anything. You could, for example, run it on a Web server and build an application around it. The following script assumes that you have built and trained a classifier in the variable named `classifier`. The only requirement is that you store the character string in an iterable (here we are storing it in a list) so that CountVectorizer can process it.

```
new_data = ['this is new data, and it is fantastic']
processed_newdata = count_vect.transform(new_data)
processed_newdata = tfidf_transformer.transform(processed_newdata)
print(classifier.predict(processed_newdata))
```

The code above should result in a prediction of 'positive'.

9 Tasks

A lot of the work for an NLP practitioner is made up of evaluating multiple solutions to determine the one that should be deployed. The following tasks will focus on doing the exact same thing, but on a smaller scale.

1. [⋆] Look up the documentation for more algorithms, such as Multinomial Naive Bayes, Support Vector Machine, and Decision Tree. Run your own evaluation campaign of those algorithms and find out which performs better.

2. [⋆] Similarly, compare multiple term weighting approaches - with and without IDF, with and without log-scaling TF. Try to use multiple algorithms and see if there is an interaction effect between term weighting and the classifier.

3. [⋆] Once you have found the best combination of term weighting technique and classification algorithm, use that best combination and train it using your entire dataset. Try to write your own reviews and use your classifier to classify them. Does it work as expected?

4. [⋆] In machine learning, it is common to introduce new datasets with what we call a dataset paper, which is a research paper that introduces a new dataset for an existing task (e.g., a new dataset for sentiment analysis) or a whole new task altogether that has some practical significance. If you want to practice further, have a look at the following papers and corresponding datasets and try to see how you fare compared to other approaches:

   - "Liar, Liar Pants on Fire": A New Benchmark Dataset for Fake News Detection[3] [1] (Fake News Classification)
   - "GoEmotions: A Dataset of Fine-Grained Emotions" [4] [2] (Emotion and Sentiment Classification)

[3] https://arxiv.org/abs/1705.00648v1
[4] https://arxiv.org/abs/2005.00547

10 References

References

[1] William Yang Wang. "liar, liar pants on fire": A new benchmark dataset for fake news detection. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 422–426, 2017.

[2] Dorottya Demszky, Dana Movshovitz-Attias, Jeongwoo Ko, Alan Cowen, Gaurav Nemade, and Sujith Ravi. Goemotions: A dataset of fine-grained emotions. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4040–4054, 2020.

[3] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

Tips and tricks

You have made it this far, congratulations! In this section, I am going to describe some tips and tricks acquired over my career, in no specific order, to get the most of your efforts when using NLP models in real-world applications.

Tip 1: Not all models are equally hungry

Machine learning can become very hungry with data to work correctly. However, the hungriness depends mostly on the type of models, and a rule of thumb you can use is that the fewer assumptions a model makes, the more data it requires to work. Unfortunately, it is difficult to know in advance which model will take to a task, which is why we usually try a few to see what works.

Tip 2: Bootstrap your dataset

While it is tempting to grab some data on the Web, in reality most things can be done in-house. For example, if you were trying to bootstrap a sentiment classification algorithm for a specific use case, you could do the following.

1. Type 50 positive and 50 negative sentences (use your imagination).
2. Write a dictionary that links words with grammatical equivalents. For example, "I" could be replaced with "We".
3. Write a script that goes through your handmade dataset and replaces N random words with words which are semantically equivalent.
4. Write a script that breaks down a document into sentences and repeats a random sentence.
5. Use these two scripts to generate new documents from each existing document.
6. Make sure you do that only on your training dataset, not on your testing dataset.

Why does this work? It's actually fairly simple. Data augmentation is based on the principle of invariance. Let us illustrate it with an image classifier: assume that we trained a machine learning model to differentiate dog pictures from cat pictures. If I took a picture of my cat, regardless of its position in the image, it would still be the image of a cat. If the cat is in the top right corner of the image, it is a cat. If it is in the bottom left corner of the image, it's still a cat. There is no position where the cat is in the picture and the label of that picture is not "cat" - we call it **translation**[5] **invariance** (the image is translated but the class does not change).

Here, the same principle applies. It does not matter whether I write `I liked the movie` or `We liked the food`, it is still a positive sentence. That is an invariance in the data that we can exploit. Similarly, if I repeat the same sentence twice, it will probably not change its meaning. That is another invariance we can exploit.

I can hear you thinking: that sounds like a lot of work! Fortunately, some people have already done most of it, such as the author of the Python library `NLPAug`[6].

[5] `https://en.wikipedia.org/wiki/Translation_(geometry)`
[6] `https://github.com/makcedward/nlpaug`

Tip 3: Beware of class imbalance

If you are classifying something, there will come a time where your classes are so imbalanced (e.g., 90%-10%) that your classifier will simply decide to just output the majority class. This is called the **class imbalance problem** and is subject to a lot of work in the machine learning community. Here are the simplest ways to deal with it in your training set.

Subsampling the majority The simplest is to take a random sample of the majority class so that the ratio is closer to reasonable.

Oversampling the minority The second simplest is to oversample the minority class, that is, to sample some documents multiple times in order for the final training sets to be more or less balanced. Note that oversampling is different from augmentation (see Tip 2). A common oversampling method is SMOTE [3] (Synthetic Minority Over-sampling Technique).

Using cost-sensitive learning Some libraries allow you to specify a cost matrix during training. A cost matrix attributes the weight of making specific mistakes. For example, if the class ratio is 2:1, then doubling the cost of misclassifying the minority class re-establishes balance in the model.

Tip 4: Modularise

Once you have trained your models, you can think of each of them as a branching factor in a chatbot. If you sufficiently engineer your classifier, it will be fairly easy to use it to increase the complexity of your bot. For example, a classifier of `{yes ; no ; maybe}` could help you analyse the response of your user without constraining them to a list of words (it is a bit of a stretch, but it is just an example). A classifier of `{QA ; weather ; smalltalk}` could be used to detect the intent of the user and route them accordingly.

By abstracting this away, you can think of a chatbot as a state machine, and model it using a UML state diagram[7].

Tip 5: Debug your NLP system

How do you find a fault in an NLP system, when so much of it is random? You need to take a principled approach, and treat it like a black box that you are investigating. Here is my own process for doing so:

1. State your assumptions about the system. For example, a sentiment analysis model will take an input in the form of text and output a label, which is either positive or negative.
2. From your assumptions, develop your expectations:
    - What are the expected outcomes? For example, a correct label is output by the system.
    - What are the potential anomalies? For example, the vocabulary of the document to classify is completely different from the vocabulary of our NLP system, or the system outputs an incorrect label due to the presence of sarcasm.
3. Collect or generate some data, which matches the intended use of your system as well as edge cases that might break it.
4. Test the system with the data: does it match your expectations? Go back to step 1 and apply fixes if not, stop there if it does.

[7] `https://en.wikipedia.org/wiki/State_diagram`