Part I

Exploration
Finding Structure in the Data

DRAFT

Now that we have seen what the basic building blocks linguistic analysis are, and how we can represent text in the computer, we can start looking at extracting information out of the text. We will see how to search for flexible patterns in the data to identify things like email addresses, company names, or numerical amounts in Chapter I.1; how to find the most meaningful and important words in a corpus (given that pure frequency is not a perfect indicator) in Chapter I.2; how to capture the notion that some words occur much more frequently together than on their own, and how to define this notion in Chapter I.3; how to spot unusual or creative sentences (and potentially generate our own) in Chapter **??**; and how to find the most important themes and topics (via topic models) in Chapter **??**. We will then see how to incorporate external information into our text representations in Chapter **??**, before we look at ways to visualize the information we have (Chapter **??**) and to group texts together into larger clusters (Chapter **??**).

DRAFT

CHAPTER I.1

Regular Expressions

Say you are working with a large sample of employment records from Italian employees, with detailed information when they worked at which company, and a description of what they did there. However, the names of the companies are contained in the text, rather than in a separate entry, and you want to extract them to get a rough overview of how many different companies there are, and to group the records accordingly. Unfortunately, the NER tool for Italian does not work as well as you would like. You do know, though, that Italian company names often end in a number of abbreviations (similar to "Co.", "Ltd.", or "Inc."). There are only two you want to consider for now ("s.p.a." and "s.r.l."), but they come in many different spelling variations (e.g., *SPA*, *spa*, *S.P.A.*, etc.). How can you identify these markers and extract the company names right before them?

In the same project, you have collected survey data from a large sample of employees. Some of them have left their email address as part of the response, and you would like to spot and extract the email addresses. You could look for anything that comes before or after an sign, but how do you make sure it is not a Twitter handle? And how do you make sure it is a valid email address?[1]

String variation is a textbook application of **regular expressions** (or RegEx). They are flexible **patterns** that allow you to specify what you are looking for in general, and how it can vary. In order to do so, you simply write the pattern you want to match.

regular expressions
patterns

sequence	matches
`e`	any single occurrence of e
`at`	`at, rat, mat, sat, cat, attack, attention, later`

TABLE 1. Examples of simple matching.

DRAFT

In Python, we can specify regular expressions and then apply them to text with either `search()` or `match()`. The former checks whether a pattern is *contained* in the input, the latter checks whether the pattern *completely matches* the input:

[1]With "valid", we don't mean whether it actually exists (you could only find out by emailing them), but only whether it is actually correctly formatted (so that you actually *can* email them).

```python
import re
pattern = re.compile("at")

re.search(pattern, 'later') # match at position (1,3)
re.match(pattern, 'later') # no match
```

quantifiers

Sometimes, we know that a specific character will be in our pattern, but not how many times. We can use special **quantifiers** to signal that the character right before them occurs any number of times, including zero (*), exactly one or zero times (?), or one or more times (+).

quantifier	means	example	matches
*	0 or more	`fr?og`	`fog, frog`
+	1 or more	`hello+`	`hello, helloo, hellooooooo`
?	0 or 1	`cooo*l`	`cool, coool`

TABLE 2. Examples of quantifiers.

In Python, we simply incorporate them into the patterns:

```python
pattern1 = re.compile("fr?og")
pattern2 = re.compile("hello+")
pattern3 = re.compile("cooo*l")
```

character	means	example	matches
`.`	any single character	`.el`	`eel, Nel, gel`
`\n`	newline character (line break)	`\n+`	One or more line breaks
`\t`	a tab stop	`\t+`	One or more tabs
`\d`	a single digit [0-9]	`B\d`	`B0, B1, ..., B9`
`\D`	a non-digit	`\D.t`	`' t, But, eat`
`\w`	any alphanumberic character	`\w\w\w`	`Top, WOO, bee,`...
`\W`	non-alphanumberic character		
`\s`	a whitespace character		
`\S`	a non-whitespace character		
`\`	"Escapes" special characters to match them	`.+\.com`	`abc.com, united.com`
`^`	the beginning of the input string	`^...`	First 3-letter word in line
`$`	the end of the input string	`^\n$`	Empty line

TABLE 3. Examples of special characters.

DRAFT

However, maybe we don't want really *any* old character in a wildcard position, only one of a small number of characters. For this, we can define a **group** of allowed characters:

group

group	means	example	matches
`[abc]`	Match any of `a, b, c`	`[bcrms]at`	`bat, cat, rat, mat, sat`
`[^abc]`	Match anything BUT `a, b, c`	`te[^ ]+s`	`tens, tests, teens, texts, terrors`…
`[a-z]`	Match any lowercase character	`[a-z][a-z]t`	`act, ant, not, ... wit`
`[A-Z]`	Match any uppercase character	`[A-Z]...`	`Ahab, Brit, In a, ..., York`
`[0-9]`	Match any digit	`DIN A[0-9]`	`DIN A0, DIN A1, ..., DIN A9`

TABLE 4. Examples of groups.

In Python, we can use groups to define our patterns:

```python
pattern = re.compile('[bcrms]at')
```

A warning: regular expressions are extremely powerful – a well-written RegEx can save you a lot of work, and accomplish quite a bit in terms of extracting the right bits of text. However, RegExes can quickly become extremely complicated, when you add more and more variants that you would like to account for. It's very easy to write a RegEx that captures way too much, or is so specific that it captures way too little. It is therefore always a good idea to write a couple a positive and negative examples, and make sure the RegEx you develop matches what you want.

DRAFT

DRAFT

CHAPTER I.2

TF-IDF

Let us say we have 500 companies, and for each of them several years' worth of annual reports. The collected reports of each company are one document. We will not be surprised to find the words "report" or "company" among the most frequent terms, since they are bound to be used in every document. However, if only some of those companies use "sustainability", but do so consistently in every report, this would be a very interesting signal worth discovering. How could we capture this notion?

When we first look at a corpus, we usually want to know what it is generally about, and what terms we might want to pay special attention to. Frequency alone is not sufficient to find important words: many of the stopwords we purposely remove are extremely frequent. However, there definitely is something about frequency that carries information. It is just not the raw frequency of each word.

The answer is to weigh the raw frequency (its **term frequency**, or TF) by the number of documents the word occurs in (its **document frequency**, or DF). This way, words that occur frequently, but in every document (such as function words), will receive very low scores, as will words that are rare, but could occur everywhere (say, "president" in our example: it is not frequent, but will be mentioned by every report). Words that are moderately frequent, but occur in only few documents get the highest number, and this is what we are after. We can get both of these quantities from a discrete count matrix: the TF is the sum over the corresponding column of the matrix, while the DF is the count of the non-zero entries in that column.

term frequency

document frequency

There are many ways to compute the weighting factor, but in the most common one, we check how many documents a term occurs in, and take its **inverse**, or the fraction of all documents in our corpus that contain the word:

inverse

$$idf(w) = log\frac{N}{|d \in D : w \in d|}$$

This is—quite sensibly–called the **inverse document frequency** (or IDF). Because we often have large corpora and terms occur in only few, the value of the IDF can get very small, risking an underflow. To prevent this, we take the logarithm.[1] The idea was first introduced by Spärck Jones (1972).

inverse document frequency

[1]You might also see the IDF as $-log\frac{|d \in D : w \in d|}{N}$, which comes out to the same.

DRAFT

By multiplying the two terms together, we get the TF-IDF score of a word:

$$tfidf(w) = tf(w) \cdot idf(w)$$

There are several variants for both TF and IDF, usually involving Laplace smoothing and taking the logarithm of the frequency. I.e., we add 1 to each count, to prevent division by 0 (in case a term is not in the corpus), and use the logarithm to dampen frequency effects. A common variant is

$$tfidf(w) = (1 + \log tf(w)) \cdot \frac{1}{\log(1 + \frac{N}{df(w)})}$$

By plotting the IDF versus the TFIDF values, and scaling by the TF, we can see how the non-stopwords in a corpus are distributed. Fig 1 shows the content words in Moby Dick. The five words with the highest TFIDF scores are labeled. While the prevalence of "whale" is hardly surprising (the TFIDF is dominated by the high TF, but then, it's a book about whales), we can see a couple of interesting entries to the right, including the doomed protagonist Ahab, and the archaic form "ye" (for "you").

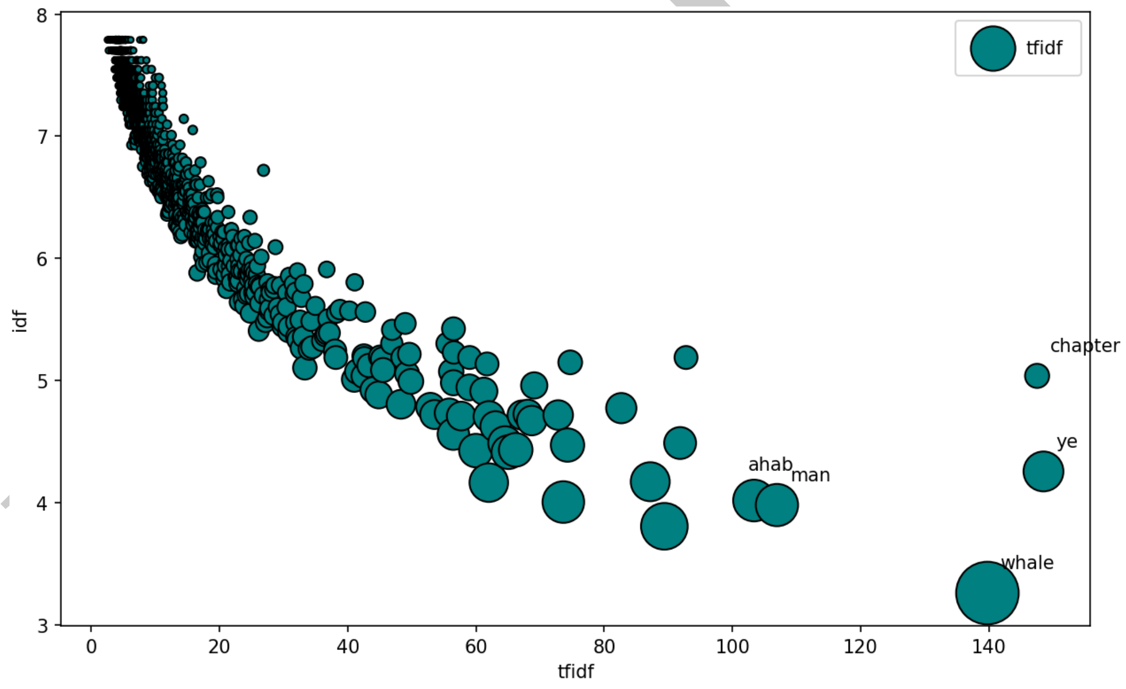

FIGURE 1. Scatter plot of TFIDF and IDF values, scaled by TF, for content words in Moby Dick

In Python, we can use the `TfidfVectorizer` in `sklearn`, which functions very much like the `CountVectorizer` we have already seen:

```python
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(analyzer='word', min_df=0.001,
    max_df=0.75, stop_words='english', sublinear_tf=True)

X = tfidf_vectorizer.fit_transform(documents)
```

CODE 1. Extracting TFIDF values from a corpus.

This returns a data matrix with a row for each document, and the transformed values in the cells. We can get the IDF values from the `idf_` property of the vectorizer. In order to get the TFIDF values of a certain feature for the entire corpus, we can sum over the all rows/documents. The `A1` property turns the sparse matrix into a regular array:

```python
idf = tfidf_vectorizer.idf_
tfidf = X.sum(axis=0).A1
```

CODE 2. computing IDF and TFIDF values for features.

DRAFT

DRAFT

CHAPTER I.3

Pointwise Mutual Information

Intuitively, we would like to treat terms like “social media”, “New York”, or “greenhouse emission” as a single concept, rather than treating them as individual units delimited by white space. However, just looking at the frequency of the two words together is not sufficient: the word pair “of the” is extremely frequent, but nobody would argue that this is a meaningful single concept. To complicate matters more, each part of these often also occurs by itself: think of the elements in “New York” (while “York” is probably not too frequent, “New” definitely is). Leaving these words apart or joining them will have a huge impact on our findings. How do we join the right words together?

Word is a squishy concept, and we have seen that whitespaces are more of an optional visual effect than a meaningful boundary. Chinese does not use them, and even the (in)famous German noun compounds (”Nahrungsmittelettikierungsgesetz”, or ”*law for the labeling of food stuffs*”) can be argued to be just a more whitespace-efficient way of writing the words, rather than individually separating them.

Even in English, we have concepts that include several words, but that we would not (anymore) analyze as separate components: place names like “Los Angeles” and “San Luis Obispo”, but also concepts like “social media”. (If you disagree on the latter, that is because the process, called **grammaticalization**, is gradual, and in this case not fully completed.) While it is still apparent what the parts mean, they have become inextricably linked with each other. These word pairs are called **collocations**.

grammaticalization

collocations

In many cases where we work with text, it is better to treat collocations as one entity, rather than as separate units. To do this, we usually just replace the white space between the words with an underscore (i.e., we would use `los_angeles`). While there are some “general” collocations (like “New York” or “Los Angeles”), there are usually specialized concepts that are specific to the context of our analysis (for example “quarterly earnings” in the context of reports, or “great service” if we are analyzing online reviews).

We can compute the probability of each word occurring on its own, but this does not help much: “New” is much more likely to occur than “York”. We need to relate these quantities to how likely the two words are to occur together. If we find that either (or all) of the words in the possible collocation occur predominantly in the context of the

DRAFT

other words (such as “Angeles” in the case of “Los Angeles”), we have reason to believe we have a collocation. If, on the other hand, either word is just as likely to occur independently, we probably do not have a collocation.

joint probability

To put numbers to this intuition, we use the **joint probability** of the entire n-gram occurring in a text, and normalize it by the individual probabilities (for more on probabilities, see Appendix A). Since the probabilities can get fairly small, we typically use the logarithm of the result. So for a bigram we have

$$PMI(x;y) = \log \frac{P(x,y)}{P(x)P(y)}$$

Joint probabilities can be rewritten as conditional probabilities, i.e.,

$$P(x,y) = P(y) \times P(x|y) = P(x) \times P(y|x)$$

The last two parts are the same because we don’t care about the order of x and y. So if it’s more convenient, we can transform our PMI calculation into

$$PMI(x;y) = \log \frac{P(x|y)}{P(x)}$$

or

$$PMI(x;y) = \log \frac{P(y|x)}{P(y)}$$

Essentially, the latter tells us for a bigram “xy” how likely we are to see x followed by y (i.e., $P(y|x)$), normalized by how likely we are to see y in general ($P(y)$).

Applying collocation detection to an example corpus in Python is much easier, since there are some functions implemented in `nltk` that save us from computing all the probability tables:

```python
from nltk.collocations import BigramCollocationFinder,
    BigramAssocMeasures
from nltk.corpus import stopwords

stopwords_ = set(stopwords.words('english'))

words = [word.lower() for document in documents for word in document.
    split()
          if len(word) > 2
          and word not in stopwords_]
finder = BigramCollocationFinder.from_words(words)
bgm = BigramAssocMeasures()
collocations = {bigram: pmi for bigram, pmi in finder.score_ngrams(
    bgm.mi_like)}
collocations
```

CODE 3. Finding collocations with `nltk`.

DRAFT

The code first creates a long list of strings from the corpus, removing all stopwords and words shorter than 2 characters (thereby removing punctuation). It then creates a `BigramCollocationFinder` object, initialized with the word list. From this object, we can extract the statistics. Here, we use a variant of PMI, called `mi_like`, which does not take the logarithm, but uses an exponent.

Table 1 shows the results for the text sample from Moby Dick. We can then set a threshold above which we concatenate the words.

RANK	COLLOCATION	$MI(x;y)$
1	moby_dick	83.000000
2	sperm_whale	20.002847
3	mrs_hussey	10.562500
4	mast_heads	4.391153
5	sag_harbor	4.000000
6	vinegar_cruet	4.000000
7	try_works	3.794405
8	dough_boy	3.706787
9	white_whale	3.698807
10	caw_caw	3.472222

TABLE 1. MI values and rank for a number of collocations in Moby Dick

DRAFT

DRAFT

Bibliography

Karen Spärck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21.

DRAFT

DRAFT

APPENDIX A

Probabilities

In some cases, we will represent words as probabilities. In the case of topic models or language models, this allows us to reason under uncertainty. However, there is an even simpler reason to use probabilities: Keeping track of frequencies is often not too practical for our purposes. In one of our previous example, some quarterly reports might be much longer than others, so seeing "energy" mentioned more often in those longer reports than in shorter ones is not too informative by itself. We need to somehow account for the length of the documents.

For this, normalized counts, or probabilities, are a much better indicator. Of course, probabilities are nothing more than counts normalized by a **Z-score**. Typically, those Z-scores are the counts of all words in our corpus.

Z-score

$$P(w) = \frac{count(w)}{N}$$

where N is the total count of words in our corpus.

A **probability distribution** over a vocabulary therefore assigns a probability between 0.0 and 1.0 to each word in the vocabulary. And if we summed up all probabilities in the vocabulary, the result would be 1.0 (otherwise, it is not a probability distribution).

probability distribution

Technically, probabilities are continuous values. However, since each probability is a discrete feature (i.e., it "means" something) in a feature vector, we cover them here under discrete representations.

Say we have a corpus of 1000 documents, x is "natural" and occurs in 20 documents, and y is "language" and occurs in 50 documents.

The probability $P(x)$ of seeing a word x if we reached into a big bag with all words in our vocabulary, is therefore simply the number of documents that contain x, say "natural" (here, 20), divided by the number of all documents in our corpus (1000).

$P(x)$

$$P(natural) = \frac{count(documents\ w.\ \text{“}natural\text{”})}{number\ of\ all\ documents} = \frac{20}{1000} = \frac{1}{50} = 0.02$$

$$P(language) = \frac{count(documents\ w.\ \text{“}language\text{”})}{number\ of\ all\ documents} = \frac{50}{1000} = \frac{1}{20} = 0.05$$

Note that probabilities are not percentages (even though they are often used that way in everyday use)! In order to express a probability in percent, you need to multiply it by 100. So we have a 2% probability of randomly drawing the word "natural", and a 5% chance of drawing "language".

DRAFT

As we have seen, though, word frequencies follow a Zipf distribution, so the most frequent words get a lot of the probability mass, whereas most of the words get a very small probability. Since this can lead to vanishingly small numbers that computers sometimes can not handle, it is common to take the logarithm of the probabilities (also called **log-probabilities**). Note that in log-space, multiplication becomes addition, and division becomes subtraction. So another way to compute the (log) probability of a word is to subtract the log-count of the total number of words from the log-count of the word count.

log-probabilities

$$\log P(w) = \log count(w) - \log N$$

To get a normal probability back, we have to exponentiate, but this can cause a loss of precision. It is therefore usually better to store raw counts and convert them to log-probabilities as needed.

$$\log P(natural) = \log 20 - \log 1000 = \log 1 - \log 50 = -3.912023005428146$$

1. Joint Probability

joint probability

$P(x, y)$ is a **joint probability**, i.e., how likely is it that we see x and y together, that is the words in one document ("natural language", "language natural", or separated by other words, since order does not matter in this case). Say we have 10 documents that contain both words, then

$$P(natural, language) = \frac{count(documents\ w.\ \text{``}natural\text{''}\ and\ \text{``}language\text{''})}{number\ of\ all\ documents}$$

$$= \frac{10}{1000} = \frac{1}{100} = 0.01$$

2. Conditional Probability

Words do not occur in isolation, but in context, and in certain contexts, they are more likely than in others. If you hear the sentence "Most of the constituents mistrusted the ..." you expect only a small number of words to follow. "Politician" is much more likely than, say "handkerchief". That is the probability of "politician" in this context is much higher than that of "handkerchief". Or, to put it in mathematical notation:

$$P(politician|CONTEXT) > P(handkerchief|CONTEXT)$$

conditional probability

$P(y|x)$ is a **conditional probability**, i.e., how likely are we to see y after having seen x.

DRAFT

Let's reduce the context to one word for now and look at the words in our previous example, i.e., "language" in a document that contains "natural". We can compute this as

$$P(language|natural) = \frac{P(x,y)}{P(x)} = \frac{\frac{count(documents\ w.\ \text{``}natural\text{''}\ and\ \text{``}language\text{''})}{number\ of\ all\ documents}}{\frac{count(documents\ w.\ \text{``}natural\text{''})}{number\ of\ all\ documents}}$$

$$= \frac{count(documents\ w.\ \text{``}natural\text{''}\ and\ \text{``}language\text{''})}{count(documents\ w.\ \text{``}natural\text{''})}$$

$$= \frac{10}{20} = \frac{1}{2} = 0.5$$

Note that order *does* matter in this case! So the corresponding probability formulation for seeing "natural" in a document that contains "language", or $P(x|y) = \frac{P(x,y)}{P(y)}$ is something completely different (namely 0.2).

3. Probability Distributions

So far, we have treated probabilities as normalized counts, and for the most part, that works well. However, there is a more general way of looking at probabilities, and that is as a functions. They take as input an event (a word or some other thing), and returns a number between 0 and 1. In the case of normalized counts, the function itself was a lookup table, or maybe a division. However, we can also have probability functions that are defined by an equation, and that take a number of parameters other than the event we are interested in.

The general form of these probability functions is

$$f(x;\theta)$$

where x is the event we are interested in, and θ is just a shorthand for any of the additional parameters.

What makes these functions probability distributions are the facts that they

(1) cover the entire sample space (for example all words)
(2) sum to 1.0 (if we added up the probabilities of all words)

In the case of texts, we mostly work with **discrete probability distributions**, i.e., there is a defined number of events, for example words. When we graph these distributions, we usually use bar graphs. There are also continuous probability distributions, the most well known of which is the normal or Gaussian distribution. However, for the sake of this book, we will not need them. IN the following, we will look at some of the most common discrete probability distributions.

discrete probability distributions

DRAFT

3.1. Uniform distribution. The simplest distribution is the one where all events are equally likely, so all we have to return is 1 divided by the number of outcomes.

$$U(x; N) = \frac{1}{N}$$

A common example is a die roll: all numbers are equally likely to come up. This is also a good distribution to use if we do not know anything about the data and do not want to bias our model unduly.

3.2. Bernoulli distribution. This is the simplest possible discrete distribution: it only has two outcomes, and can be described with a single parameter q, which is the chance of success. The other outcome is simply $1 - q$

$$Pr(x; q) = \begin{cases} 1 - q & if x = 0 \\ q & if x = 1 \end{cases}$$

If $q = 0.5$, the Bernoulli distribution is also a uniform distribution.

3.3. Multinomial distribution. Most often, we will deal with distributions that are much larger than two outcomes, and much more irregular than uniform. Formally, it is parametrized by a single parameter θ, which is a vector with all probabilities. The mathematical definition is

$$P(x; \theta) = \prod_{j=1}^{1} \theta_j^{I(x_j)=1}$$

which is a very complicated way of writing "use the value at the vector position for x"

A good example is the distribution over characters in a sample of text. In Figure 1, we can see how different characters have very different probabilities.

DRAFT

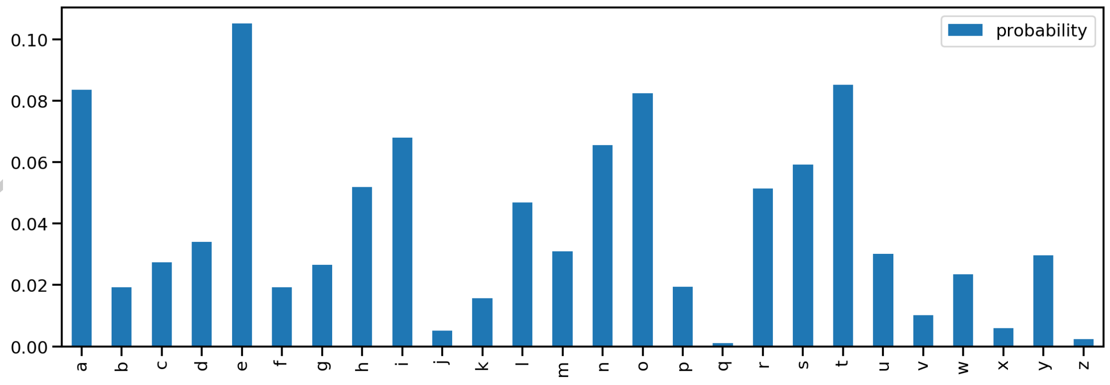

FIGURE 1. Multinomial distribution over lower-case characters.

In fact, if we sort them by their likelihood, we can see that they rapidly decline. This is another example of Zipf's Law, and can be modeled with a Zeta distribution, which we will not go into here.

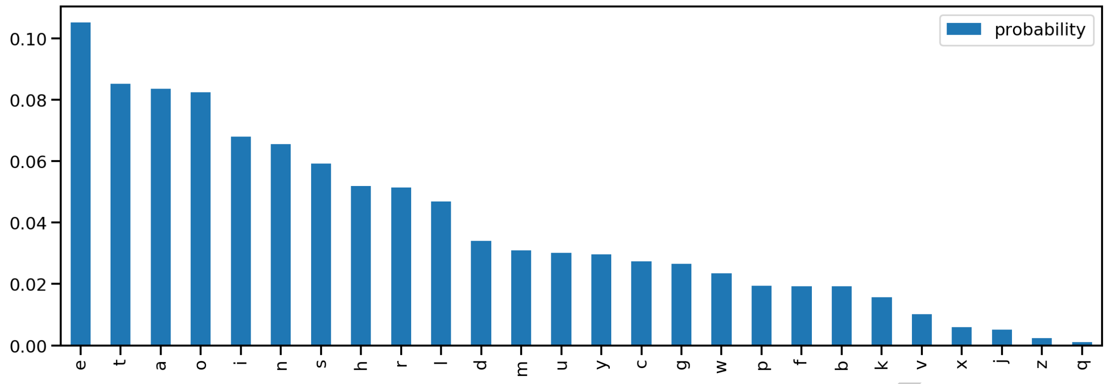

FIGURE 2. Sorted multinomial distribution over lower-case characters.

3.4. Dirichlet distribtution. A **Dirichlet distribution** is a distribution over multinomial distributions, and has only one parameter, called α. We can imagine a Dirichlet distribution as a generator function that gives us multinomial distributions over k outcomes for each document. The parameter α controls how peaked or uniform the multinomial distributions are: if α is close to 0, the distributions are very peaked, i.e., one outcome will have (almost) all the probability mass. The larger α gets, the more uniform the distributions become. Figure 3 shows the effect over 20 draws from a Dirichlet over 3 outcomes with different values for α.

Dirichlet distribution

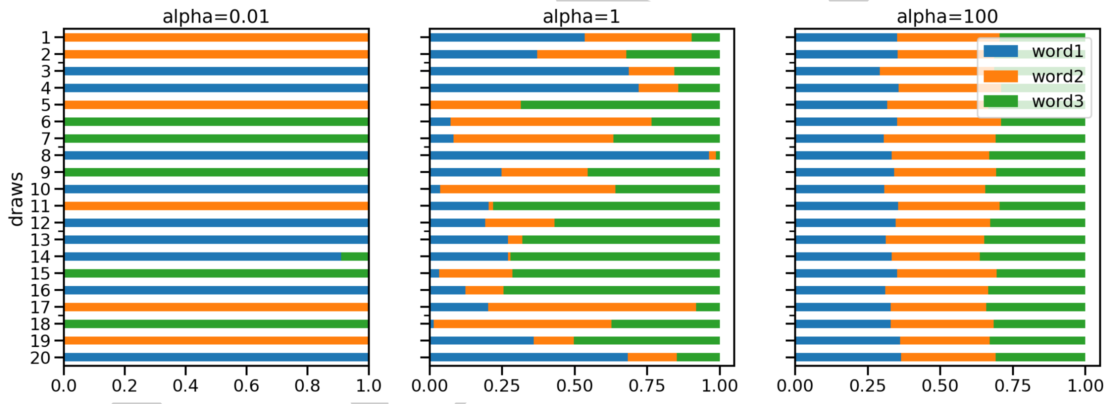

FIGURE 3. 20 multinomial distributions over three outcomes drawn from Dirichlets differing in α

DRAFT