Creating features from text

1. Looking at the character sets: Latin, Cyrillic, Greek, CJK (Chinese, Japanese, Korean), etc. can provide a first categorization into language families.
2. Looking at one-letter, two-letter or three-letter words and their frequency in a text.
3. Character n-grams and their frequency.
4. Word n-grams and their frequency.

What are n-grams?

Word n-grams deal with sentences. “I like red cherries” will have as bigrams: “I like”, “like red”, “red cherries”.

N-grams are a common model for representing language in the field of Natural Language Processing (a subfield of Artificial Intelligence).
Comparing character bigrams in different languages

To notice:
- The top 3 bigrams for English cannot be found at all in the list of Spanish bigrams.
- The two lists have 14 bigrams in common out of 30 (less than half).
- The bigrams that are in common have different frequency. E.g., EN is 2.27 in Spanish and 1.13 in English.

Note: These bigrams were calculated from a large set of news stories. Because the word "the" is the most common word in English speech, that explains why the two bigrams "th" and "he" are at the top. If we use only the vocabulary of English words, the list will change. The most common bigram becomes "in", because of the many words that start with "in" or that end in "ing".

English word length distribution

Problem: Given the dictionary of all English words, what is the distribution of words by length?

Solution 1 (requires two separate loops)
1. Iterate over the list of words to find the length of each word and store it into a new list. [Accumulation in a list via mapping]
2. Iterate over the list of lengths and store it into a dictionary to keep track of the number of times we encounter each length. [Accumulation via a dictionary]

Solution 2 (requires one loop)
1. Iterate over the list of words to find the length of each word and immediately store it into a dictionary. [Accumulation via a dictionary]

Back to CS 111

Question: How to build a program that identifies natural languages?

Answer: We create a “signature” for each known language by processing large amounts of text. This signature is composed of different features and their frequency distributions. Then, for new text, we compare its signature to that of known languages and pick the one that comes the closest.

Scenario 1
We are given the `englishwords` list from your psets, what features can we extract from it?
- The frequency distribution of word lengths.
- The frequency distribution of character n-grams.
- The frequency distribution of words starting with a particular letter.
and many more.
Fun with statistics

In Statistics, it is common to describe a dataset (e.g., the list of the lengths of all English words) in terms of descriptive statistics: the mean, the median, the mode (the value that occurs the most), the variance, the standard deviation, etc. All these statistics can be calculated with the operations we have been learning.

- The **mean** is the sum of all list elements divided by the length of the list. (\(\text{sum} = \text{accumulation to a number}\))
- The **median** is the middle element of a sorted list.
- The **mode** is the most frequent element (i.e., the max value from the frequency dictionary.)
- The **variance** is the sum of the squares of the difference of each item to the mean.
- The **standard deviation** is the square of the variance.

Try it out

Using the `lengthsList` and `lengthsDct` from the previous slide, you can practice calculating these statistics with Python code.

You should find that both the median and the mode are 8.

Accumulation Pattern

The bigram frequency distribution

```python
def bigrams(word):
    """Given a word return a list of its bigrams."""
    return ["".join(pair) for pair in zip(word, word[1:])]""

def createBigramFrequency()
    """Create and return the bigram frequency distribution of all words in 'englishwords'."""
    bigramsDct = {}  # accumulator dictionary

    for word in englishwords:
        bigramsList = bigrams(word)  # create ngrams as a list
        # add new bigrams or update counts of existing ones
        for ngram in bigramsList:
            bigramsDct[ngram] = bigramsDct.get(ngram, 0) + 1

    return bigramsDct
```

Accumulation Pattern

Building character n-grams

Unigrams:  
'b', 'o', 's', 't', 'o', 'n'

```python
word = 'boston'
list(word)
```

Bigrams:  
'bo', 'os', 'st', 'to', 'on'

```python
["".join(pair) for pair in zip(word, word[1:])]
```

Trigrams:  
'bos', 'ost', 'sto', 'ton'

```python
["".join(triple) for triple in zip(word, word[1:], word[2:])]
```

N-gram frequency distributions

- There are 66230 words in `englishwords`, we want to avoid iterating over them many times to create all n-gram distributions.
- We can create all three n-gram distributions in one single loop.
- Imagine we have three functions: `unigrams`, `bigrams`, `trigrams` that contain as bodies the statements in 16-10.
- Imagine also a function `storeNgrams` that takes a list of n-grams and a dictionary and adds the list elements to it, by updating their count.

```python
unigramsDct = {}
bigramsDct = {}
trigramsDct = {}

for word in englishwords:
    # create ngrams
    ngrams1 = unigrams(word)
    ngrams2 = bigrams(word)
    ngrams3 = trigrams(word)

    # store ngrams in freq dicts
    storeNgrams(ngrams1, unigramsDct)
    storeNgrams(ngrams2, bigramsDct)
    storeNgrams(ngrams3, trigramsDct)
```

Concepts in this slide:

How to avoid multiple iterations by creating helper functions?

Question

Can you hypothesize why the function `storeNgrams` doesn't return a value?
Mutating Dictionaries via aliasing

A dictionary can be mutated via aliasing.

```python
def storeNgrams(ngramsList, ngramsDict):
    """Given a list of items and a dictionary, update the counts of the dictionary keys. """
    for ngram in ngramsList:
        ngramsDict[ngram] = ngramsDict.get(ngram, 0) + 1
```

Function Call Frames

```
storeNgrams(ngramsList, ngramsDict)
```

Concepts in this slide:

- Dictionary comprehension is very similar to list comprehension.
- We can use dictionary comprehension in situations when we want to start accumulation with a complex data structure (see next slide).

Dictionary Comprehension

Very much like list comprehension: use `{}` instead of `[]` and create pairs with the colon syntax, e.g., `aKey: aValue`.

**Syntax:** `{ aKey: aValue for aKey in sequence }`

**Example:** Write a dictionary comprehension that pairs words with their lengths.

```
In [1]: wordsList = 'the autumn is dragging its feet'.split()
In [2] {word: len(word) for word in wordsList}
Out[2]: {'autumn': 6, 'dragging': 8, 'feet': 4, 'is': 2, 'its': 3, 'the': 3}
```

Important

We can use dictionary comprehension in situations when we want to start accumulation with a complex data structure (see next slide).

Analyzing the Results

- Predict what will be the max lengths for the `unigramsDict`, `bigramsDict`, and `trigramsDict`: ____, ____, ____
- Do you expect that all dictionaries will have that max length? Explain.
- Predict the top 3 unigrams, top 3 bigrams, and top 3 unigrams.
- How to write a function `sortItemsInFreqDict` that given a frequency dictionary will return the sorted list (in descending order) of its items, based on the value of each (key/value) item? Hint: use `lambda`.
- Which will be more frequent (have the highest values): the top unigrams, the top bigrams, or the top trigrams?

Accumulating in a dictionary of dictionaries

**Problem:** How can we create a dictionary that has two level of keys? In the first level, each key is a unigram, in the second level the keys are bigrams that start with the unigram. [See example on the right.]

**Solution 1:** Assume we already have `bigramsDict`: ```python
{'a': {'aa': 19, 'ab': 1665, 'ac': 2387, 'ad': 1685, ...},
'b': {'ba': 1431, 'bb': 417, 'bc': 25, 'bd': 35, ...
```

```python
from string import ascii_lowercase as lowercase
# 'abcdefghijklmnopqrstuvwxyz'

# create the dict with unigrams as keys and empty dict as values
bigramsByFirstLetter = {char: {} for char in lowercase}

for bigram in bigramsDict:
    unigram = bigram[0]
    # assign the second level of keys
    bigramsByFirstLetter[unigram][bigram] = bigramsDict[bigram]
```
Accumulating in a dictionary of dictionaries [2]

Solution 2: We don’t have bigrams, we create them as we iterate over the list of words.
from string import ascii_lowercase as lowercase
# 'abcdefghijklmnopqrstuvwxyz'

# create the dict with unigrams as keys and empty dict as values
bigramsByFirstLetter = {char: {} for char in lowercase}

for word in englishwords:
    # create list of bigrams from word
    bigramsLst = bigrams(word)
    # iterate over bigrams
    for bigram in bigramsLst:
        unigram = bigram[0]  # access the nested bigram dict for easy reference
        bigramsDct = bigramsByFirstLetter[unigram]
        # increase frequency counter
        bigramsDct[bigram] = bigramsDct.get(bigram, 0) + 1

Summary
1. Lists and dictionaries are powerful data structures that are used routinely to perform complex data analysis tasks such as transforming data from one form to another.
2. Accumulation is a very common pattern in problem solving; we accumulate frequencies (counts) as we encounter new data; or we organize data as nested dictionaries of dictionaries or dictionaries of lists.
3. When we need to accumulate into nested structures, first always draw a picture of what the structure would look like, in order to visualize what needs to be created through code.
4. Dictionaries are mutable and they can be changed via aliasing (two different variables pointing to the same dictionary object).
5. The nested structures would need double subscripting operations (e.g., see last statement in slide 16-16). If this is conceptually difficult, you can store the inner structure into a temporary variable and work with that instead. Because of aliasing, this temporary variable will be directly mutating the entire dictionary. [See second from last statement in 16-17.]
6. Use a dictionary comprehension statement whenever you need to create a dictionary of dictionaries or a dictionary of lists in the case that the keys of the outer dictionary are known.

Accumulating in a dictionary of lists.

Problem: Group words from englishwords based on their ending: words ending with 'ed', 'ly', 'es', etc.

Solution Algorithm:
1. Create an empty dictionary
2. Iterate over words and get the ending of each word.
3. Check to see if the key/value for ending is already in the dictionary using the method get with the default value an empty list.
4. Append the word to the list associated with its ending.

wordsByEnding = {}
for word in englishwords:
    ending = word[-2:]
    wordsByEnding[ending] = wordsByEnding.get(ending, [])
    wordsByEnding[ending].append(word)