各グローバル変数の役割

・number

n-gram の n の値を決定する。1 であれば unigram を、2 であれば bigram を意味する。

・discountin

discount の値を定義 Kneser-Ney smoothing の計算時に活用する。

・dirname = os.getcwd()

os ライブラリを活用して、本プログラムがあるフォルダのパスを取得、保存する。train および test データセットをロードする際に活用する。

・counts = {}

train データセットにおける各単語の出現回数を保存するための辞書。”単語名”:”出現回数” の形式で保存される。$C(w_{i-n+1} \dots w_i)$の計算に用いる。

・context_counts = {}

train データセットにおける、ある単語に先行する単語ごとの出現回数を保存するための辞書。$C(w_{i-n+1} \dots w_{i-1})$ の計算に用いる。

・preceding_words_types = {}

train データセットにおける、ある単語もしくは単語群に後行する単語の種類数を保存するための辞書。Kneser-Ney smoothing における $\sum_{w'}\{v: C(v\, w') > 0\}$ ， $\sum_{w'}\{v: C(v\, w_{i-n+1} \dots w_{i-1}\, w') > 0\}$ ，$\{v: C(v\, w_{i-n+1} \dots w_i) > 0\}$の計算に用いる。

・following_words_types = {}

train データセットにおける、ある単語もしくは単語群に先行する単語の種類数を保存するための辞書。Kneser-Ney smoothing におけるλの$\{v: C(w_{i-n+1} \dots w_{i-1}\, v) > 0\}$ の計算に用いる。

・test_vocaburary = {}

test データの語彙数を把握するための辞書。

各関数の役割

・def n_gram(str, n)

与えられた文章から n-gram を作成し、返す関数。

引数: (str: n-gram にかける文章、n: gram 数 上記の number と同等)

・def load_train(tgtfile, n)

対象となる train ファイルに対して処理を施し、各単語および単語の組み合わせに対して確率を算出する関数。

引数: (tgtfile: train 指定ファイル、n: gram 数)

具体的な処理:

1. train データセットの各文に対し、カンマ,とピリオド.を除去、先頭と末尾に"<s>"と"</s>"を追加する処理を施す。
2. 各文を後述の all_context にかける。
3. counts = {}から各単語および単語の組み合わせを取り出し、それぞれに対して counts と context_counts における出現回数を用いて確率を計算する。

・def all_context(words, n)

上記の counts, context_counts, preceding_words_types, following_words_types に単語と出現回数・種類数を保存する関数。

引数: (words: 対象となる文章、n: gram 数)

具体的な処理:

1. 文章を unigram〜number-gram にかけ、分割された文章を一度リストに保存する。
2. リストの各要素に対して、' '.join で結合した箇所における counts の値を+1 する。例えば、['machine-learning', 'paradigm']という要素に対しては、counts['machine-learning paradigm']に+1 される。また、各要素の$[w_0 \dots w_{i-1}]$までを' '.join で結合した箇所における context_counts の値を+1 する。先程の例では、context_counts['machine-learning']に+1 される。
3. 各単語もしくは単語の組み合わせが初めて出現した時、その$[w_0 \dots w_{i-1}]$までを' '.join で結合した箇所における following_words_types の値を+1 する。
   さらに、bigram の際は preceding_words_types['']に+1 ($\sum_{w'}\{v: C(v\ w') > 0\}$に相当)、
   bigram 以上の際は preceding_words_types[' '.join(リストの要素$[w_1 \dots w_i]$) + '¥t' + str(現在の n-gram の n の数 - 1)]に+1 ($\{v: C(v\ w_{i-n+1} \dots w_i) > 0\}$に相当)、
   trigram 以上の際は preceding_words_types[' '.join(リストの要素$[w_1 \dots w_{i-1}]$) + '¥t' + str(現在の n-gram の n の数 - 1)]に+1 ($\sum_{w'}\{v: C(v\ w_{i-n+1} \dots w_{i-1}\ w') > 0\}$に相当)する。

・def savefile_en(list, n)

np.savetxt でパラメーターをファイルとして保存する関数。

引数: (list: 保存対象リスト、n: gram 数)

・def load_test(test_file, gram, d):
対象となる test ファイルに対して処理を施し、全体のエントロピーを算出する関数。
引数: (test_file: test 指定ファイル、gram: gram 数、d: discounting)
具体的な処理:

1. test_vocaburary = {}を用いて test データセットの（"<s>"と"</s>"も加えた）語彙を保存し、その総数を算出する。
2. test データセットの各文に対し、カンマ,とピリオド.を除去、先頭と末尾に"<s>"と"</s>"を追加する処理を施す。
3. 各文を後述の probability_KN にかけ、1 文ごとの確率を算出、合計する。
4. 確率の合計を test データセットの単語数で除算し、エントロピーを算出する。

・def probability_KN(words, gram, pro_KN, total, d)
各文に対し、Kneser-Ney smoothing を実行して確率を算出する関数。
引数: (test_file: test 指定ファイル、gram: gram 数、d: discounting)
具体的な処理:

1. 文章を unigram〜number-gram にかけ、分割された文章を一度リストに保存する。
2. gram が 2 より大きい時（すなわち bigram 以上である時):
   この場合はさらに以下に分けられる。
   gram = number の時（すなわち the highest order である時):
   $P_{KN}(w_i|w_{i-n+1:i-1}) = \frac{\max(C(w_{i-n+1:i})-d,0)}{\sum_v C(w_{i-n+1:i-1}\ v)} + \lambda(w_{i-n+1:i-1})P_{KN}(w_i|w_{i-n+2:i-1})$を再帰を利用して返す。
   gram ≠ number の時（すなわち lower order である時):
   $P_{KN}(w_i|w_{i-n+1:i-1}) = \frac{\max(\{v:C(v\ w_{i-n+1}\dots w_i)>0\}-d,0)}{\sum_{w'}\{v:C(v\ w_{i-n+1}\dots w_{i-1}\ w')>0\}} + \lambda(w_{i-n+1:i-1})P_{KN}(w_i|w_{i-n+2:i-1})$ を再帰を利用して返す。
   gram が 1 の時（すなわち unigram である時):
   $P_{KN}(w_i) = \frac{\max(\{v:C(vw_i)>0\}-d,0)}{\sum_{w'}\{v:C(vw')>0\}} + \frac{d}{total}$を返す。
   なお、$\lambda(w_{i-n+1:i-1})$は後述の normalizing_constant で算出する。
3. 上記過程で、対象とするリストの要素が未知語であった場合、一律で$\frac{d}{total}$を pro_KN に加算する。

・def normalizing_constant(str_words, gram, d)
Kneser-Ney smoothing におけるλを計算し、返す関数。
引数: (words: 対象となる文章、gram: gram 数、pro_KN: 現在の確率の値、total: test データの語彙数、d: discounting)

test データセットに対する結果:
entropy = 0.11281205931015247