Language Modeling

Daniel Jurafsky – Image from web

Language Model

Introduction

이전 챕터까지 우리는 단어 또는 문장을 입력으로 받아서, 어떤 값으로 맵핑(mapping)해주는 방법에 대해서 다루었습니다. 그 값을 통해 해당 단어 또는 문장을 분류하기도 하고 군집을 형성 할 수도 있었습니다. 이러한 방법들도 매우 쓰임새가 많고, 정말 중요합니다. 하지만, 우리는 여기서 더 나아가 컴퓨터로 하여금 필요에 따라 자연스러운 문장을 만들어내도록 하는 방법에 대해 다루어 보도록 하겠습니다.

언어 모델(Language Model)은 문장의 확률을 나타내는 모델 입니다. 즉, 우리는 언어 모델을 통해 문장 자체의 출현 확률을 예측 하거나, 이전 단어들이 주어졌을 때 다음 단어를 예측할 수 있으며, 결과적으로 우리는 주어진 문장이 얼마나 자연스러운 유창한(fluent) 표현인지 계산 할 수 있게 됩니다. 예를 들어 우리는 아래와 같은 문장이 주어졌을 때, 빈칸을 어렵지 않게 메꿀 수 있습니다.

- 버스 정류장에서 방금 버스를 ㅇㅇㅇ.

1. 사랑해
2. 고양이
3. 놓쳤다
4. 사고남

우리는 정답이 3번 '놓쳤다'라고 쉽게 맞출 수 있습니다. 4번 '사고남'의 경우에는 앞 단어가 '버스를' 대신에 '버스가' 였다면 정답이 될 수도 있었겠지요. 4번을 정답이라고 할 경우에는 뜻은 이해할 수 있지만 뭔가 어색함이 느껴지는 문장이 될 겁니다.

1. 저는 어제 점심을 먹었습니다.
2. 저는 2018년 4월 26일 점심을 먹었습니다.

위의 두 문장 중 1번 문장을 접할 기회는, 2번 문장을 접할 기회보다 훨씬 많을 겁니다. 수많은 단어들이 있고, 그 단어들간의 조합은 더욱 많이 존재 합니다. 그러한 조합들은 동등한 확률을 갖기보다는 자주 나타나는 단어나 표현(단어의 조합)이 훨씬 높은 확률로 나타날 겁니다.

이렇게 우리는 살아오면서 수많은 문장을 접하였고, 우리의 머릿속에는 단어와 단어 사이의 확률이 우리도 모르게 학습되어 있습니다. 덕분에 누군가와 대화를 하다가 한 단어 정도는 정확하게 알아듣지 못하여도, 애매한 경우를 제외하고는, 대화에 지장이 없을 겁니다. (물론 여기에는 문맥 정보를 이용 하는 것도 큰 도움이 됩니다.) 이와 같이 꼭 자연어처리 분야가 아니더라도, 음성인식(Speech Recognition)이나 문자인식(Optical Character Recognition, OCR)에 있어서도 언어모델은 큰 역할을 수행합니다. 우리는 인터넷이나 도서에서 많은 문장들을 (대개 수십만에서 수억까지) 수집하여 단어와 단어 사이의 출현 빈도를 세어 확률을 계산하고, 언어모델을 구성합니다. 궁극적인 목표는 우리가 일상 생활에서 사용하는 문장의 분포를 정확하게 파악하는데에 있습니다. 또한, 만약 특정한 목표를 가지고 있다면 (예를 들어 의료 분야의 음성인식기 제작) 해당 분야(domain)의 문장의 분포를 파악하기 위해서 해당 분야의 말뭉치(corpus)를 수집하기도 합니다.

Again, Korean is Hell

우리는 언어들의 구조적 특성에 따라서 여러 갈래로 언어를 분류합니다. 이전(Why Korean NLP is Hell)에 다루었듯이 한국어는 대표적인 **교착어**입니다. 그리고 영어는 **고립어(+굴절어)**의 특성을 띄고, 중국어는 **고립어**로 분류합니다. 교착어의 특성상, 단어의 의미 또는 역할은 어순에 의해 결정되기 보단, 단어에 부착되는 어미와 같은 접사에 의해 그 역할이 결정됩니다. 따라서 같은 의미의 단어라 할지라도 붙는 접사에 따라 단어의 형태가 달라지게 되어 단어의 수가 늘어나게 됩니다. (버스가, 버스를, 버스에, 버스로 등 같은 버스라도 뒤에 붙는 어미에 따라 다른 단어가 됩니다.) 다시 말해, 단어의 어순이 중요하지 않기 때문에 (또는 생략 가능하기 때문에), 단어와 단어 사이의 확률을 계산하는데 불리하게 작용할 수 있습니다.

- 나는 학교에 갑니다 버스를 타고 .
- 나는 버스를 타고 학교에 갑니다 .
- 버스를 타고 나는 학교에 갑니다 .
- (나는) 버스를 타고 학교에 갑니다 .

위의 세문장 모두 같은 의미의 표현이고 사용 된 단어들도 같지만, 어순이 다르기 때문에, 단어와 단어 사이의 확률을 정의하는데 있어서 혼란이 가중됩니다. 같은 의미의 문장을 표현하기 위해서 '타고' 다음에 나타날 수 있는 단어들은 '.', '학교에', '나는' 3개이기 때문에, 확률이 쉽게 말해 퍼지는 현상이 생기게 됩니다. 이에 반해 영어나 기타 라틴어 기반 언어들은 좀 더 어순에 있어서 규칙적이므로 좀 더 유리한 면이 있습니다.

더군다나, 한국어는 교착어의 특성상 어미가 부착되어 단어를 형성하기 때문에 같은 '학교'라는 단어가 '학교+에', '학교+로', '학교+를', '학교+가', … 등 수많은 단어로 파생되어 만들어질 수 있습니다. 따라서 사실 어미를 따로 분리해주지 않으면 어휘의 수가 기하급수적으로 늘어나게 되어 sparse함이 더욱 높아질 수 있어 더욱 문제 해결이 어려워질 수 있습니다.

Probability of Sentence

먼저 문장의 확률을 표현 해 보도록 하겠습니다. w_1, w_2라는 2개의 단어가 한 문장 안에 순서대로 나타났을 때, 이 문장의 확률로 표현하면 다음과 같습니다.

$$P(w_1, w_2)$$

우리는 이 수식을 베이즈 정리(Bayes Theorem)에 따라 조건부 확률(Conditional Probability)로 표현할 수 있습니다.

$$P(w_1, w_2) = P(w_1)P(w_2|w_1)$$
$$\text{because}$$
$$P(w_2|w_1) = \frac{P(w_1, w_2)}{P(w_1)}.$$

좀 더 나아가서 Chain Rule을 통해, $W = \{w_1, w_2, w_3, w_4\}$, 4개의 단어가 한 문장 안에 있을 때를 표현 해 보면 아래와 같습니다.

$$P(W) = P(w_1, w_2, w_3, w_4) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)P(w_4|w_1, w_2, w_3)$$

여기서 Chain Rule(연쇄 법칙)이란, 조건부 확률(conditional probability)을 사용하여 결합 확률(joint probability)를 계산 하는 방법으로, 아래와 같이 유도 할 수 있습니다.

$$\begin{aligned} P(A, B, C, D) &= P(D|A, B, C)P(A, B, C) \\ &= P(D|A, B, C)P(C|A, B)P(A, B) \\ &= P(D|A, B, C)P(C|A, B)P(B|A)P(A) \end{aligned}$$

우항을 해석해보면, w_1가 나타날 확률과 w_1가 주어졌을 때 w_2가 나타날 확률, w_1, w_2가 주어졌을 때 w_3가 주어졌을 확률, w_1, w_2, w_3가 주어졌을 때 w_4가 나타날 확률을 곱하는 것을 알 수 있습니다. 이로써 우리는 language model을 통해서 문장에 대한 확률 뿐만 아니라, 단어와 단어 사이의 확률도 정의 할 수 있습니다. 우리는 이를 일반화하여 아래와 같이 표현할 수 있습니다.

$$P(W) = \prod_{i=1}^{n} P(w_i|w_{<i})$$

또는 log확률로 표현하여 곱셈 대신 덧셈으로 표현할 수 있습니다. 참고로, 문장이 길어지게 된다면 당연히 확률에 대한 곱셈이 거듭되면서 확률이 매우 작아지게 되어 정확한 계산 또는 표현이 힘들어지게 됩니다. (또한, 곱셈 연산보다 덧셈 연산이 더 빠릅니다.) 따라서 우리는 log를 취하여 덧셈으로 바꾸어 더 나은 조건을 취할 수 있습니다.

$$\log P(W) = \sum_{i=1}^{n} \log P(w_i|w_{<i})$$

이제 우리는 실제 예제를 가지고 표현 해 보도록 하겠습니다. Corpus $\mathcal{C} = \{W_1, W_2, \cdots\}$에서 i번째 문장 $W_i = \{BOS, 나는, 학교에, 갑니다, EOS\}$에 대한 확률을 Chain rule을 통해 표현하면 아래와 같습니다.

$P(BOS, 나는, 학교에, 갑니다, EOS) = P(BOS)P(나는|BOS)P(학교에|BOS, 나는)P(갑니다|BOS, 나$

여기서 BOS는 Beginning of Sentence라는 의미의 토큰이고, EOS는 End of sentence라는 의미의 토큰 입니다. $P(BOS)$ 의 경우에는 항상 문장의 시작에 오게 되므로 상수가 될 것 입니다. $P(나는|BOS)$ 경우에는 문장의 첫 단어로 나는이 올 확률을 나타내게 됩니다.

이제 문장을 어떻게 확률로 나타내는지 알았으니, 확률을 직접 구하는 방법에 대해서 알아보겠습니다. 우리는 앞 챕터에서 문장을 수집하는 방법에 대해서 논의 했습니다. 수집한 말뭉치 내에서 직접 단어들을 카운트 함으로써 우리가 원하는 확률을 구할 수 있습니다. 예를 들어 아래의 확률은 다음과 같이 구할 수 있습니다.

$$P(갑니다|BOS, 나는, 학교에) = \frac{COUNT(BOS, 나는, 학교에, 갑니다)}{COUNT(BOS, 나는, 학교에)}$$

N-gram

Sparseness

이전 섹션에서 언어모델에 대해 소개를 간략히 했습니다. 하지만 만약 그 수식대로라면 우리는 확률들을 거의 구할 수 없을 것 입니다. 왜냐하면 비록 우리가 수천만~수억 문장을 인터넷에서 긁어 모았다고 하더라도, 애초에 출현할 수 있는 단어의 조합의 경우의 수는 무한대에 가깝기 때문입니다. 문장이 조금만 길어지더라도 Count를 구할 수 없어 분자가 0이 되어 확률이 0이 되거나, 심지어 분모가 0이 되어 정의할 수가 없어져버릴 것이기 때문입니다. 이렇게 너무나도 많은 경우의 수 때문에 생기는 문제를 희소성(sparseness or sparsity)문제라고 표현합니다.

Markov Assumption

따라서 희소성(sparseness) 문제를 해결하기 위해서 Markov Assumption을 도입합니다.

$$P(x_i|x_1, x_2, \cdots, x_{i-1}) \approx P(x_i|x_{i-k}, \cdots, x_{i-1})$$

Markov Assumption을 통해, 다음 단어의 출현 확률을 구하기 위해서, 이전에 출현한 모든 단어를 볼 필요 없이, 앞에 k 개의 단어만 상관하여 다음 단어의 출현 확률을 구하도록 하는 것 입니다. 이렇게 가정을 간소화 하여, 우리가 구하고자 하는 확률을 근사(approximation) 하겠다는 것 입니다. 보통 k는 0에서 3의 값을 갖게 됩니다. 즉, $k = 2$ 일 경우에는 앞 단어 2개를 참조하여 다음 단어(x_i)의 확률을 근사하여 나타내게 됩니다.

$$P(x_i|x_{i-2}, x_{i-1})$$

이를 이전 chain rule(연쇄법칙) 수식에 적용하여, 문장에 대한 확률도 다음과 같이 표현 할 수 있습니다.

$$P(x_1, x_2, \cdots, x_n) \approx \prod_{j=1}^{n} P(x_j|x_{j-k}, \cdots, x_{j-1})$$

이것을 log확률로 표현하면,

$$\log P(x_1, x_2, \cdots, x_n) \approx \sum_{j=1}^{n} logP(x_j|x_{j-k}, \cdots, x_{j-1})$$

우리는 이렇게 전체 문장 대신에 바로 앞 몇 개의 단어만 상관하여 확률 계산을 간소화 하는 방법을 $n = k + 1$으로 n-gram 이라고 부릅니다.

k	n-gram	명칭
0	1-gram	uni-gram
1	2-gram	bi-gram
2	3-gram	tri-gram

위 테이블과 같이 3-gram 까지는 tri-gram이라고 읽지만 4-gram 부터는 그냥 four-gram 이라고 읽습니다. 앞서 설명 하였듯이, n이 커질수록 우리가 가지고 있는 훈련 corpus내에 존재하지 않을 가능성이 많기 때문에, 오히려 확률을 정확하게 계산하는데 어려움이 있을 수도 있습니다. (예를 들어 훈련 corpus에 존재 하지 않는다고 세상에서 쓰이지 않는 문장 표현은 아니기 때문 입니다.) 따라서 당연히 훈련 corpus의 양이 적을수록 n의 크기도 줄어들어야 합니다. 보통은 대부분 어느정도 훈련 corpus가 적당히 있다는 가정 하에서, 3-gram을 가장 많이 사용하고, 훈련 corpus의 양이 많을 때는 4-gram을 사용하기도 합니다. 하지만 이렇게 4-gram을 사용하면 언어모델의 성능은 크게 오르지 않는데 반해, 단어 조합의 경우의 수는 지수적(exponential)으로 증가하기 때문에, 사실 크게 효율성이 없습니다.

$$P(x_i|x_{i-2}, x_{i-1}) = \frac{C(x_{i-2}, x_{i-1}, x_i)}{C(x_{i-2}, x_{i-1})}$$

이제 위와 같이 3개 단어의 출현 빈도와, 앞 2개 단어의 출현 빈도만 구하면 x_i의 확률을 근사할 수 있습니다. 즉, 아래와 같은 문장 전체의 확률에 대해서

$$P(x_1, x_2, \cdots, x_n)$$

비록 훈련 corpus 내에 해당 문장이 존재 한 적이 없더라도, Markov assumption을 통해서 우리는 해당 문장의 확률을 근사(approximation)할 수 있게 되었습니다.

Generalization

머신러닝의 힘은 보지 못한 case에 대한 대처 능력, 즉 일반화(generalization)에 있습니다. 따라서, n-gram도 Markov assumption을 통해서 희소성(sparseness or sparsity)를 해결하는 일반화 능력을 갖게 되었다고 할 수 있습니다. 이제 우리는 이것을 좀 더 향상시킬 수 있는 방법을 살펴 보도록 하겠습니다.

Smoothing (Discounting)

출현 횟수를 단순히 확률 값으로 이용 할 경우 문제점이 무엇이 있을까요? 바로 훈련 corpus에 출현하지 않는 단어 조합에 대한 대처 방법 입니다. 비록, markov assumption을 적용하여 우리는 희소성 문제를 훨씬 줄였고, 문장의 모든 단어 조합에 대해서 출현 횟수를 세지 않아도 되지만, 어찌되었든 그래도 훈련 corpus에 등장하지 않는

경우는 결국 존재 할 것인데, 그 경우인 훈련 셋에서 보지 못한 단어 조합(unseen word sequence)이라고 해서 등장 확률이 0이 되면 맞지 않습니다. 따라서 출현 횟수(counting) 값 또는 확률 값을 좀 더 다듬어 줘야 할 필요성이 있습니다. 아래 파란색 막대기와 같이 들쭉날쭉한 출현 횟수 값을 주황색 선으로 부드럽게(smooth) 바꾸어 주기 때문에 smoothing 또는 discounting이라고 불립니다. 그럼 그 방법에 대해 살펴보겠습니다.

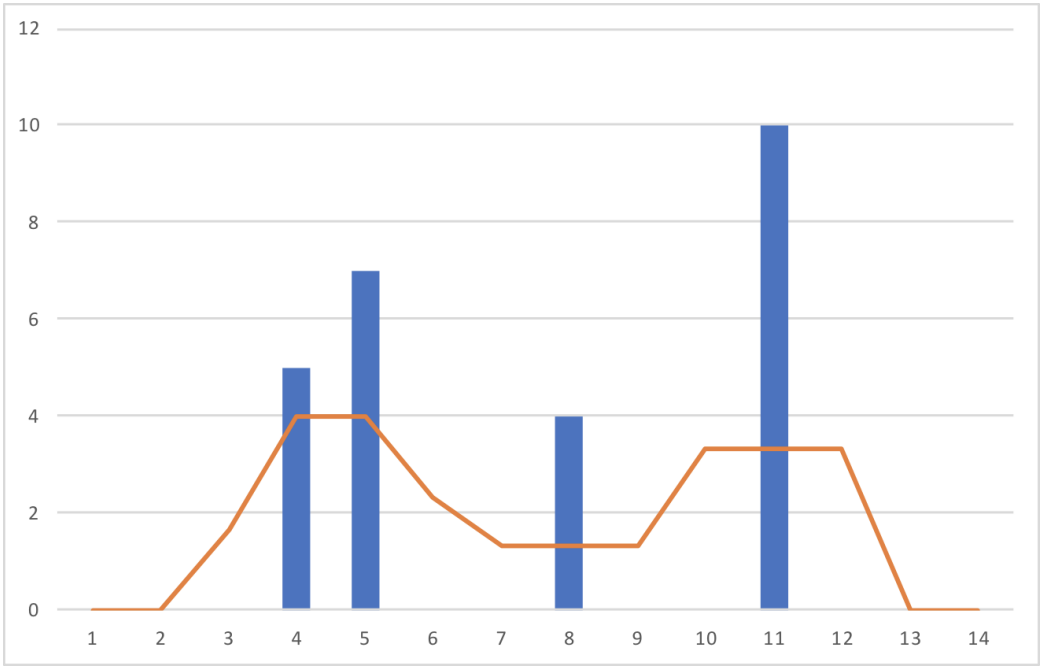


Figure 1:

Add one Smoothing

먼저 우리가 생각 해 볼 수 있는 가장 간단한 방법은, 모든 n-gram에 1을 더하는 것입니다. 그렇다면 훈련셋에 출현하지 않은 n-gram의 경우에도 작은 확률이나마 가질 수 있을 것 입니다. 이를 수식으로 나타내면 아래와 같습니다.

$$P(w_i|w_{<i}) \approx \frac{C(w_{<i}, w_i) + 1}{C(w_{<i}) + V}$$

이처럼 1을 더하여 smoothing을 통해 $P(w_i|w_{<i})$를 근사할 수 있습니다. 이 수식을 좀 더 일반화하여 표현하면 아래와 같이 쓸 수 있습니다.

$$P(w_i|w_{<i}) \approx \frac{C(w_{<i}, w_i) + k}{C(w_{<i}) + kV}$$
$$\approx \frac{C(w_{<i}, w_i) + (m/V)}{C(w_{<i}) + m}$$

이처럼, 1보다 작은 상수값을 더하여 smoothing을 구현 해 볼 수도 있을 것 입니다. 그렇다면 여기서 또 한발 더 나아가 1-gram prior 확률을 이용하여 좀 더 동적으로 대처 해 볼 수도 있을 것 입니다.

$$P(w_i|w_{<i}) \approx \frac{C(w_{<i}, w_i) + mP(w_i)}{C(w_{<i}) + m}$$

이처럼 add-one smoothing은 매우 간단하지만, 사실 언어모델처럼 희소성(sparseness) 문제가 큰 경우에는 부족합니다. 따라서 언어모델에 쓰이기에는 알맞은 방법은 아닙니다.

Absolute Smoothing

[Church et al.1991]은 bigram에 대해서 실험을 한 결과를 제시하였습니다. 훈련용 corpus에서 n번 나타난 2-gram에 대해서, held-out corpus (validation or development set)에서 나타난 횟수를 세어 평균을 낸 것 입니다. 그 결과는 아래와 같습니다.

재미있게도, 0번과 1번 나타난 2-gram을 제외하면, 2번부터 9번 나타난 2-gram의 경우에는 held-out corpus에서의 출현 횟수는 훈련용 corpus 출현 횟수보다 약 0.75번 정도 적게 나타났다는 것 입니다. 즉, 출현 횟수(counting)에서 상수 d를 빼주는 것과 같다는 것입니다.

Kneser-Ney Smoothing

[Kneser et al.1995]은 여기에서 한발 더 나아가, KN discount를 제시하였습니다.

KN discounting의 주요 아이디어는 단어 w가 누군가(v)의 뒤에서 출현 할 때, 얼마나 다양한 단어 뒤에서 출현하는지를 알아내는 것 입니다. 그래서 다양한 단어 뒤에 나타나는 단어일수록, 훈련셋에서 보지 못한 단어 조합(unseen word sequence)으로써 나타날 확률이 높다는 것 입니다.

- learning의 앞에 출현한 단어: machine, deep, supervised, unsupervised, generative, discriminative, …

Bigram count in training set	Bigram count in heldout set
0	0.0000270
1	0.448
2	1.25
3	2.24
4	3.23
5	4.21
6	5.23
7	6.21
8	7.21
9	8.26

Figure 4.8 For all bigrams in 22 million words of AP newswire of count 0, 1, 2,...,9, the counts of these bigrams in a held-out corpus also of 22 million words.

Figure 2:

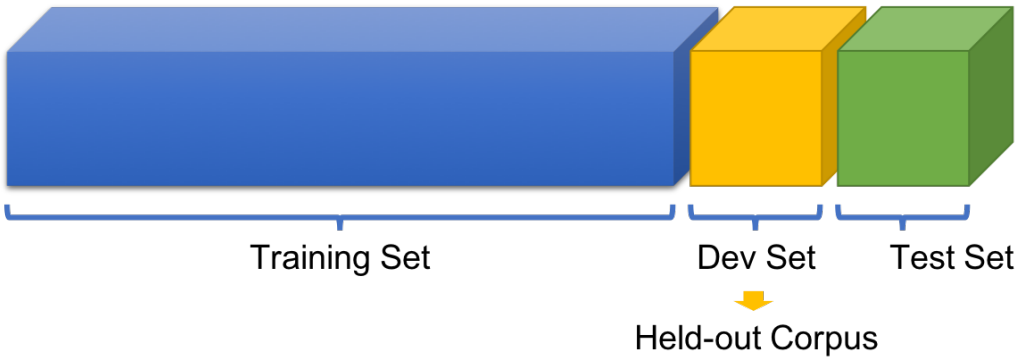


Figure 3:

- laptop의 앞에 출현한 단어: slim, favorite, fancy, expensive, cheap, ···

예를 들어, 우리 책은 machine learning(기계학습)과 deep learning(딥러닝)을 다루는 책 이므로, 책 내에서 learning이라는 keyword의 빈도는 굉장히 높을 것 입니다. 하지만, 해당 단어는 주로 machine과 deep뒤에서만 나타났다고 해 보죠. learning이라는 단어에 비해서, laptop이라는 표현의 빈도는 낮을 것 입니다. 하지만 learning과 같이 특정 단어의 뒤에서 대부분 나타나기 보단, 자유롭게 나타났을 것 같습니다. KN discounting은 이 경우, laptop이 unseen word sequence에서 나타날 확률이 더 높다고 가정 하는 것 입니다. 한마디로 낮을 덜 가리는 단어를 찾아내는 것 입니다.

KN discounting은 $P_{continuation}$을 아래와 같이 모델링 합니다. 즉, w와 같이 나타난 v들의 집합의 크기가 클 수록 $P_{continuation}$은 클 것이라고 가정 합니다.

$$P_{continuation}(w) \propto |\{v : C(v, w) > 0\}|$$

위의 수식은 이렇게 나타내 볼 수 있습니다. w와 같이 나타난 v들의 집합의 크기를, v, w'가 함께 나타난 집합의 크기의 합으로 나누어 줍니다.

$$P_{continuation}(w) = \frac{|\{v : C(v, w) > 0\}|}{\sum_{w'} |\{v : C(v, w') > 0\}|}$$

이렇게 우리는 P_{KN}를 정의 할 수 있습니다.

$$P_{KN}(w_i|w_{i-1}) = \frac{\max\left(C(w_{i-1}, w_i) - d, 0\right)}{C(w_{i-1})} + \lambda(w_{i-1}) P_{continuation}(w_i),$$
$$where\ \lambda(w_{i-1}) = \frac{d}{\sum_v C(w_{i-1}, v)} \times |\{w : c(w_{i-1}, w) > 0\}|.$$

Interpolation

Interpolation에 의한 generalization(일반화)을 살펴 보도록 하겠습니다. Interpolation은 두 다른 언어모델을 선형적으로 일정 비율(λ)로 섞어 주는 것 입니다. Interpolation을 통해 얻을 수 있는 효과는, 일반 영역의 corpus를 통해 구축한 언어모델을, 필요에 따라 다른 특정 영역(domain)의 양이 적은 corpus를 통해 구축한 영역 특화(domain specific or adapted) 언어모델과 섞어 주는 것 입니다. 이를 통해 일반적인 언어 모델을 해당 영역에 특화 된 언어 모델로 만들 수 있습니다.

$$\tilde{P}(w_n|w_{n-k},\cdots,w_{n-1}) = \lambda P_1(w_n|w_{n-k},\cdots,w_{n-1}) + (1-\lambda)P_2(w_n|w_{n-k},\cdots,w_{n-1})$$

예를 들어 의료 쪽 음성인식(ASR) 또는 기계번역(MT) 시스템을 구축한다고 가정 해 보겠습니다. 그렇다면 기존의 일반 영역 corpus를 통해 생성한 language model의 경우, 의료 용어 표현이 낯설 수도 있습니다. 하지만 만약 특화 영역의 corpus만 사용하여 언어모델을 생성할 경우, generalization 능력이 너무 떨어질 수도 있습니다.

- 일반 영역(general domain)
  - $P(|,) = 0.00001$
  - $P(|,) = 0.01$
- 특화 영역(specialized domain)
  - $P(|,) = 0.09$
  - $P(|,) = 0.04$
- Interpolated
  - $P(|,) = 0.5 * 0.09 + (10.5) * 0.00001 = 0.045005$

따라서 일반적인 대화에서와 다른 의미를 지닌 단어가 나올 수도 있고, 일반적인 대화에서는 희소(rare)한 표현(word sequence)가 훨씬 더 자주 등장 할 수 있습니다. 이런 상황들에 잘 대처하기 위해서 해당 영역 corpus로 생성한 언어모델을 섞어주어 해당 영역(domain)에 특화 시킬 수 있습니다.

Back-off

너무 긴 단어 조합(word sequence)은 실제 훈련 corpus에서 굉장히 희소(rare, sparse)하기 때문에, 우리는 Markov assumption을 통해서 일반화(generalization) 할 수 있었습니다. 그럼 그것을 응용해서 좀 더 나아가 보도록 하겠습니다.

아래 수식을 보면 특정 n-gram의 확률을 n보다 더 작은 시퀀스에 대해서 확률을 구하여 선형결합(linear combination)을 계산 하는 것을 볼 수 있습니다. 아래와 같이 n보다 더 작은 시퀀스에 대해서도 확률을 가져옴으로써 smoothing을 통해 일반화 효과를 좀 더 얻을 수 있습니다. 사실 따라서 이 방법도 interpolation의 일종이라고 볼 수 있습니다.

$$\begin{aligned}\tilde{P}(w_n|w_{n-k},\cdots,w_{n-1}) =& \lambda_1 P(w_n|w_{n-k},\cdots,w_{n-1}) \\ &+ \lambda_2 P(w_n|w_{n-k+1},\cdots,w_{n-1}) \\ &+ \cdots \\ &+ \lambda_k P(w_n),\end{aligned}$$

$$where \sum_i \lambda_i = 1.$$

또한, 다음 단어를 예측 해 내는 추론(inference)에서도, 훈련 corpus에 존재하지 않는 (unseen) n-gram에 대해서도 훈련 corpus에 나타났던 단어 시퀀스(word sequence)가 있을 때까지 back-off하여, 다음 단어의 확률을 예측 해 볼 수 있습니다.

Conclusion

n-gram 방식은 출현 횟수(count)를 통해 확률을 근사하기 때문에 굉장히 쉽고 간편합니다. 대신에 단점도 명확합니다. 훈련 corpus에 등장하지 않은 단어 조합은 확률을 정확하게 알 수 없습니다. 따라서 Markov assumption을 통해서 단어 조합에 필요한 조건을 간소화 시키고, 더 나아가 Smoothing과 Back-off 방식을 통해서 남은 단점을 보완하려 했습니다만, 이 또한 근본적인 해결책은 아니므로 실제로 음성인식이나 통계기반 기계번역에서 쓰이는 언어모델 어플리케이션 적용에 있어서 큰 난관으로 작용하였습니다. 하지만, 워낙 간단하고 명확하기 때문에 성공적으로 음성인식, 기계번역 등에 정착하였고 십 수년 동안 널리 사용되어 왔습니다. # How to evaluate Language Model?

좋은 언어모델이란 실제 우리가 쓰는 언어에 대해서 최대한 비슷하게 확률 분포를 근사하는 모델(또는 파라미터, θ)이 될 것입니다. 많이 쓰이는 문장 또는 표현일수록 높은 확률을 가져야 하며, 적게 쓰이는 문장 또는 이상한 문장 또는 표현일수록 확률은 낮아야 합니다. 즉, 테스트 문장을 잘 예측 해 낼 수록 좋은 언어모델이라고 할 수 있습니다. 문장을 잘 예측 한다는 것은, 다르게 말하면 주어진 테스트 문장이 언어모델에서 높은 확률을 가진다고 할 수 있습니다. 또는 문장의 앞부분이 주어지고, 다음에 나타나는 단어의 확률 분포가 실제 테스트 문장의 다음 단어에 대해 높은 확률을 갖는다면 더 좋은 언어모델이라고 할 수 있을 것 입니다. 그럼 이번 섹션은 언어모델의 성능을 평가하는 방법에 대해서 다루도록 하겠습니다.

Perplexity

Perplexity 측정 방법은 explicit evaluation 방법의 하나입니다. 줄여서 PPL이라고 부르기도 합니다. PPL을 이용하여 테스트 문장에 대해서 언어모델을 이용하여 점수를 구하고 언어모델의 성능을 측정합니다. 문장에 대한 확률을 구하고 문장의 길이에 대해서 normalization 합니다. 수식은 아래와 같습니다.

$$\begin{aligned} PPL(w_1, w_2, \cdots, w_n) =& P(w_1, w_2, \cdots, w_N)^{-\frac{1}{N}} \\ =& \sqrt[N]{\frac{1}{P(w_1, w_2, \cdots, w_N)}} \end{aligned}$$

문장이 길어지게 되면 문장의 확률은 당연히 굉장히 작아지게 됩니다. 따라서 우리는 문장의 길이(N)로 제곱근을 취해주어 기하평균을 구하고, 문장 길이에 대해서 normalization을 해 주는 것을 볼 수 있습니다. 수식을 보면 문장의 확률이 분모에 들어가 있기 때문에, 확률이 높을수록 PPL은 작아지는 것을 쉽게 예상할 수 있습니다.

다시말해서 테스트 문장에 대해서 확률을 높게 예측할 수록 좋은 언어모델인 만큼, 해당 테스트 문장에 대한 PPL이 작을 수록 좋은 언어모델이라고 할 수 있습니다. 즉, PPL은 수치가 낮을수록 좋습니다. 또한, n-gram의 n이 클 수록 보통 더 낮은 PPL을 보여주기도 합니다.

위 PPL의 수식은 다시한번 Chain rule에 의해서

$$= \sqrt[N]{\frac{1}{\prod_{i=1}^{N} P(w_i|w_1, \cdots, w_{i-1})}}$$

라고 표현 될 수 있고, 여기에 n-gram이 적용 될 경우,

$$= \sqrt[N]{\frac{1}{\prod_{i=1}^{N} P(w_i|w_{i-n+1}, \cdots, w_{i-1})}}$$

로 표현 될 수 있습니다.

Perplexity의 해석

Perplexity의 개념을 좀 더 짚고 넘어가도록 해 보겠습니다. 이것은 Perplexity의 수치를 해석하는 방법에 대해서라고 할 수 있습니다. 예를 들어 우리가 6면 주사위를

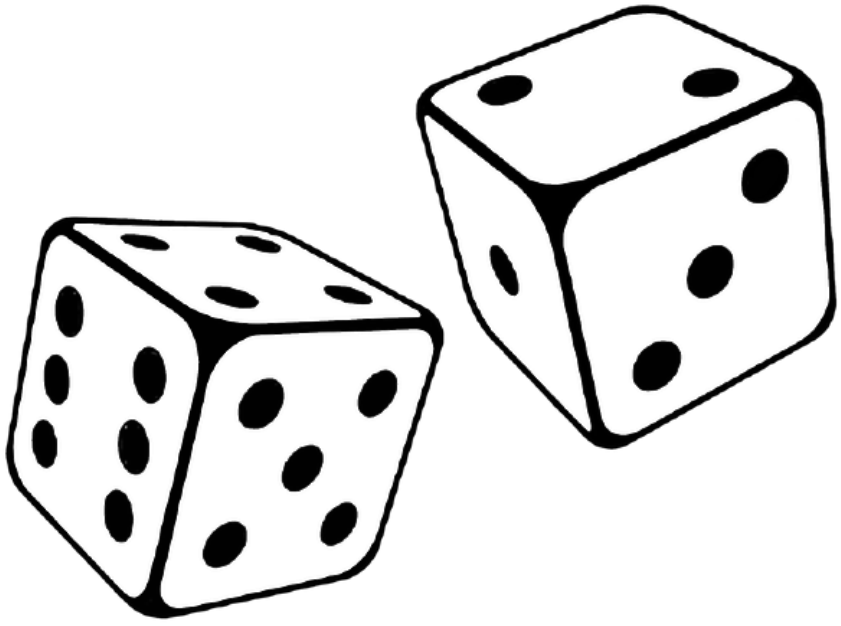


Figure 4:

던져서 나오는 값을 통해 수열을 만들어낸다고 해 보겠습니다. 따라서 1부터 6까지 숫자의 출현 확률은 모두 같다(uniform distribution)고 가정하겠습니다. 그럼 N번 주사위를 던져 얻어내는 수열에 대한 perplexity는 아래와 같습니다.

$$PPL(x) = (\frac{1}{6}^{N})^{-\frac{1}{N}} = 6$$

매 time-step 가능한 경우의 수인 6이 PPL로 나왔습니다. 즉, PPL은 우리가 뻗어나갈 수 있는 branch(가지)의 숫자를 의미하기도 합니다. 다른 예를 들어 만약 20,000개의 어휘로 이루어진 뉴스 기사에 대해서 PPL을 측정한다고 하였을 때, 단어의 출현 확률이 모두 동일하다면 PPL은 20,000이 될 것입니다. 하지만 3-gram을 사용한 언어모델을 만들어 측정한 PPL이 30이 나왔다면, 우리는 이 언어모델을 통해 해당 신문기사에서 매번 기사의 앞 부분을 통해 다음 단어를 예측 할 때, 평균적으로 30개의 후보 단어 중에서 선택할 수 있다는 얘기가 됩니다. 따라서 우리는 perplexity를 통해서 언어모델의 성능을 단순히 측정할 뿐만 아니라 실제 어느정도인지 가늠 해 볼 수도 있습니다.

Entropy와 Perplexity의 관계

엔트로피(Entropy)는 물리학에서 중요하게 사용되는 개념이지만, 정보이론(Information Theory)에서도 굉장히 중요하게 사용 됩니다. 정보이론에서 엔트로피는 어떤 정보의 불확실성을 나타내는 수치로 사용 될 수 있습니다. 정보량과 불확실성은 어떤 관계를 가질까요? 불확실성은 일어날 것 같은 사건(likely event)의 확률로 정의할 수 있습니다. 따라서 불확실성과 정보량은 아래와 같은 관계를 가집니다.

- 자주 발생하는(일어날 확률이 높은) 사건은 낮은 정보량을 가진다.
- 반대로 드물게 발생하는(일어날 확률이 낮은) 사건은 높은 정보량을 가진다.

위의 예를 실제 사례를 들어 적용 해 보겠습니다.

- 내일 아침에는 해가 동쪽에서 뜬다.
- 내일 아침에는 해가 서쪽에서 뜬다.
- 대한민국 올 여름의 평균 기온은 섭씨 28도로 예상 된다.
- 대한민국 올 여름의 평균 기온은 섭씨 5도로 예상 된다.

누군가에게 위와 같은 말을 들었을 때, 어떤 말이 우리에게 큰 도움이 될까요? 몇십억년 동안 반복되온 아침 해가 뜨는 위치가 내일은 바뀐다는 정보는 정말 천지개벽의 정보가 될 겁니다. 따라서 우리는 이처럼 확률이 낮을 수록 그 안에 포함된 정보량은 높다고 생각 할 수 있습니다.

섀넌(Claude Elwood Shannon)은 위와 같이 정보 엔트로피라는 개념과 함께 아래의 수식을 제시하였습니다. 확률 변수(Random Variable) X의 값이 x인 경우의 정보량은 아래와 같이 표현 할 수 있습니다.

$$I(x) = -\log P(x)$$

$-\log$를 취하였기 때문에, 0과 1 사이로 표현되는 확률은 0에 가까워질수록 지수적으로 높은 정보량을 가짐을 알 수 있습니다. 언어모델 관점에서 적용시켜 보면, 흔히 나올 수 없는 문장(확률이 낮은 문장)일수록 더 높은 정보량을 가질 것이라고 생각 할 수 있습니다.

우리는 이러한 정보량의 기대값을 엔트로피(entropy)라고 부릅니다.

$$H(P) = -E_{X\sim P}[\log P(x)] = -\sum_{\forall x} P(x)\log P(x)$$

위 식을 해석 해 보면, 우리는 확률 분포 P로부터 발생한 사건 X의 정보량에 대한 기대값을 구하는 것이라고 할 수 있습니다.

Cross Entropy Loss

크로스 엔트로피(Cross entropy)는 entropy로부터 한 걸음 더 나아가, 우리가 구하고자 하는 ground-truth 확률분포 P를 통해 우리가 학습중인 확률분포 Q의 정보량의 기대값을 이릅니다. 크로스 엔트로피의 수식은 아래와 같습니다.

$$H(P,Q) = -E_{X\sim P}[\log Q(x)] = -\sum_{\forall x} P(x)\log Q(x)$$

이것을 Q 대신, 우리의 모델(P_θ)을 최적화 하기 위해 최소화(minimize)해야 하는 loss(손실)함수로 적용하여 보면 아래와 같습니다.

$$L = H(P,P_\theta) = -E_{X,Y\sim P}[\log P_\theta(y|x)]$$

여기서 우리는 N번의 Sampling 하는 과정을 통해 expectation을 제거할 수 있습니다.
– 강화학습(Reinforcement Learning) 챕터 참고

$$H(P, P_\theta) = -E_{X,Y\sim P}[\log P_\theta(w_i|w_{<i})]$$
$$\approx -\frac{1}{N}\sum_{i=1}^{N} \log P_\theta(w_i|w_{<i})$$

여기서 $Y = \{y_1, y_2, \cdots, y_N\}$는 N개의 단어로 이루어진 문장(word sequence)로 생각하고, 한 문장에 대한 cross entropy는 아래와 같이 표현할 수 있습니다.

$$L = -\frac{1}{N}\sum_{i=1}^{N} \log P_\theta(w_i|w_{<i})$$
$$= \log\left((\prod_{i=1}^{N} P_\theta(w_i|w_{<i}))^{-\frac{1}{N}}\right)$$
$$= \log\left(\sqrt[N]{\frac{1}{\prod_{i=1}^{N} P_\theta(w_i|w_{<i})}}\right)$$

여기에 PPL 수식을 다시 떠올려 보겠습니다.

$$PPL(W) = P(w_1, w_2, \cdots, w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1, w_2, \cdots, w_N)}}$$

$$By\ chain\ rule,$$

$$PPL(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1, \cdots, w_{i-1})}}$$

앞서 정리했던 Cross Entropy와 수식이 비슷한 형태임을 알 수 있습니다. 따라서 PPL과 Cross Entropy의 관계는 아래와 같습니다.

$$PPL = \exp(Cross\ Entropy)$$

따라서, 우리는 반복적인(iterative) 학습을 통해 parameter θ를 배울 때, cross entropy를 통해 얻은 (P_θ의 로그 확률 값) loss 값에 exponential을 취함으로써, perplexity를 얻어 언어모델의 성능을 나타낼 수 있습니다. # n-gram Exercise with SRILM

SRILM은 음성인식, segmentation, 기계번역 등에 사용되는 통계 언어 모델 (n-gram language model)을 구축하고 적용 할 수 있는 toolkit입니다. 이 책에서 다루는 다른 알고리즘이나 기법들에 비하면, SRI speech research lab에서 1995년부터 연구/개발 해 온 유서깊은(?) toolkit 입니다.

Install SRILM

다운로드: http://www.speech.sri.com/projects/srilm/download.html

위의 주소에서 SRILM은 간단한 정보를 기입 한 후, 다운로드 가능합니다. 이후에 아래와 같이 디렉터리를 만들고 그 안에 압축을 풀어 놓습니다.

```
$ mkdir srilm
$ cd ./srilm
$ tar -xzvf ./srilm-1.7.2.tar.gz
```

디렉터리 내부에 Makefile을 열어 7번째 라인의 SRILM의 경로 지정 후에 주석을 해제 하여 줍니다. 그리고 make명령을 통해 SRILM을 빌드 합니다.

```
$ vi ./Makefile
$ make
```

빌드가 정상적으로 완료 된 후에, PATH에 SRILM/bin 내부에 새롭게 생성된 디렉터리를 등록 한 후, export 해 줍니다.

```
PATH={SRILM_PATH}/bin/{MACHINE}:$PATH
#PATH=/home/khkim/Workspace/nlp/srilm/bin/i686-m64:$PATH
export PATH
```

그리고 아래와 같이 ngram-count와 ngram이 정상적으로 동작하는 것을 확인 합니다.

```
$ source ~/.profile
$ ngram-count -help
$ ngram -help
```

Prepare Dataset

이전 preprocessing 챕터에서 다루었던 대로 tokenize가 완료된 파일을 데이터로 사용합니다. 그렇게 준비된 파일을 training set과 test set으로 나누어 준비 합니다.

Basic Usage

아래는 주로 SRILM에서 사용되는 프로그램들의 주요 arguments에 대한 설명입니다.

- ngram-count: LM을 훈련
  - vocab: lexicon file name
  - text: training corpus file name
  - order: n-gram count
  - write: output countfile file name
  - unk: mark OOV as
  - kndiscountn: Use Kneser-Ney discounting for N-grams of order n
- ngram: LM을 활용
  - ppl: calculate perplexity for test file name
  - order: n-gram count
  - lm: language model file name

Language Modeling

위에서 나온 대로, language model을 훈련하기 위해서는 ngram-count 모듈을 이용합니다. 아래의 명령어는 예를 들어 kndiscount를 사용한 상태에서 tri-gram을 훈련하고 LM과 거기에 사용된 vocabulary를 출력하도록 하는 명령 입니다.

```
$ time ngram-count -order 3 -kndiscount -text <text_fn> -lm <output_lm_fn> -write_
```

Sentence Generation

아래의 명령은 위와 같이 ngram-count 모듈을 사용해서 만들어진 lm을 활용하여 문장을 생성하는 명령입니다. 문장을 생성 한 이후에는 preprocessing 챕터에서 다루었듯이 detokenization을 수행해 주어야 하며, pipeline을 통해 sed와 regular expression을 사용하여 detokenization을 해 주도록 해 주었습니다.

```
$ ngram -lm <input_lm_fn> -gen <n_sentence_to_generate> | sed "s/ //g" | sed "s/ /
```

위와 같이 매번 sed와 regular expression을 통하는 것이 번거롭다면, preprocessing 챕터에서 구현한 **detokenization.py** python script를 통하여 detokenization을 수행할 수도 있습니다.

Evaluation

이렇게 language model을 훈련하고 나면 test set에 대해서 evaluation을 통해 얼마나 훌륭한 langauge model이 만들어졌는지 체크 할 필요가 있습니다. Language model에 대한 성능평가는 아래와 같은 명령을 통해 수행 될 수 있습니다.

```
$ ngram -ppl <test_fn> -lm <input_lm_fn> -order 3 -debug 2
```

아래는 위의 명령에 대한 예시입니다. 실행을 하면 OOVs(Out of Vocabularies)와 해당 test 문장들에 대한 perplexity가 나오게 됩니다. 주로 문장 수에 대해서 normalize를 수행한 (ppl1이 아닌) ppl을 참고 하면 됩니다.

```
$ ngram -ppl ./data/test.refined.tok.bpe.txt -lm ./data/ted.aligned.en.refined.tok
...
...
file ./data/test.refined.tok.bpe.txt: 1000 sentences, 13302 words, 32 OOVs
0 zeroprobs, logprob= -36717.49 ppl= 374.1577 ppl1= 584.7292
```

위의 evaluation에서는 1,000개의 테스트 문장에 대해서 13,302개의 단어가 포함되어 있었고, 개중에 32개의 OOV가 발생하였습니다. 결국 이 테스트에 대해서는 약 374의 ppl이 측정되었습니다. 이 ppl을 여러가지 hyper-parameter 튜닝 또는 적절한 훈련데이터 추가를 통해서 낮추는 것이 관건이 될 것 입니다.

그리고 -debug 파라미터를 2를 주게 되면 아래와 같이 좀 더 자세한 각 문장과 단어 별 log를 볼 수 있습니다. 실제 language model 상에서 어떤 n-gram이 hit되었는지와 이에 대한 확률을 볼 수 있습니다. 3-gram이 없는 경우에는 2-gram이나 1-gram으로 back-off 되는 것을 확인 할 수 있고, back-off 시에는 확률이 굉장히 떨어지는 것을 볼 수 있습니다. 따라서 back-off를 통해서 unseen word sequence에 대해서 generalization을 할 수 있었지만, 여전히 성능에는 아쉬움이 남는 것을 알 수 있습니다.

```
 I  m  pleased  with  the  way  we  handled  it  .
    p(  I | <s> )  = [2gram] 0.06806267 [ -1.167091 ]
    p(  m |  I ...)   = [1gram] 6.597231e-06 [ -5.180638 ]
    p(  pleased |  m ...)     = [1gram] 6.094323e-06 [ -5.215075 ]
    p(  with |  pleased ...)  = [2gram] 0.1292281 [ -0.8886431 ]
    p(  the |  with ...)  = [2gram] 0.05536767 [ -1.256744 ]
    p(  way |  the ...)   = [3gram] 0.003487763 [ -2.457453 ]
    p(  we |  way ...)    = [3gram] 0.1344272 [ -0.8715127 ]
    p(  handled |  we ...)    = [1gram] 1.902798e-06 [ -5.720607 ]
```

```
    p(  it |  handled ...)    = [1gram] 0.002173233 [ -2.662894 ]
    p(  . |  it ...)   = [2gram] 0.05907027 [ -1.228631 ]
    p( </s> |  . ...)   = [3gram] 0.8548805 [ -0.06809461 ]
1 sentences, 10 words, 0 OOVs
0 zeroprobs, logprob= -26.71738 ppl= 268.4436 ppl1= 469.6111
```

위의 결과에서는 10개의 단어가 주어졌고, 5번의 back-off이 되어 3-gram은 3개만 hit되었고, 4개의 2-gram과 4개의 1-gram이 hit되었습니다. 그리하여 -26.71의 log-probability가 계산되어, 268의 PPL로 환산되었음을 볼 수 있습니다.

Interpolation

SRILM을 통해서 단순한 smoothing(or discounting) 뿐만이 아니라 interpolation을 수행 할 수도 있습니다. 이 경우에는 완성된 두 개의 별도의 language model이 필요하고, 이를 섞어주기 위한 hyper parameter lambda가 필요합니다. 아래와 같이 명령어를 입력하여 interpolation을 수행할 수 있습니다.

```
$ ngram -lm <input_lm_fn> -mix-lm <mix_lm_fn> -lambda <mix_ratio_between_0_and_1>
```

Neural Network Language Model

Against to Sparseness

앞서 설명한 것과 같이 기존의 n-gram 기반의 언어모델은 간편하지만 훈련 데이터에서 보지 못한 단어의 조합에 대해서 상당히 취약한 부분이 있었습니다. 그것의 근본적인 원인은 n-gram 기반의 언어모델은 단어간의 유사도를 알 지 못하기 때문입니다. 예를 들어 우리에게 훈련 corpus로 아래와 같은 문장이 주어졌다고 했을 때,

- 고양이는 좋은 반려동물 입니다.

사람은 단어간의 유사도를 알기 때문에 다음 중 어떤 확률이 더 큰지 알 수 있습니다. 하지만, 컴퓨터는 훈련 corpus에 해당 n-gram이 존재하지 않으면, count를 할 수 없기 때문에 확률을 구할 수 없고, 따라서 확률 간 비교를 할 수도 없습니다.

- P(반려동물|강아지는, 좋은)
- P(반려동물|자동차는, 좋은)

비록 강아지가 개의 새끼이고 포유류에 속하는 가축에 해당한다는 깊고 해박한 지식이 없을지라도, 강아지와 고양이 사이의 유사도가 자동차와 고양이 사이의 유사도보다 높은 것을 알기 때문에 자동차 보다는 강아지에 대한 반려동물의 확률이 더 높음을 유추할 수 있습니다. 하지만 n-gram 방식의 언어모델은 단어간의 유사도를 구할 수 없기 때문에, 이와 같이 훈련 corpus에서 보지 못한 단어(unseen word sequence)의 조합(n-gram)에 대해서 효과적으로 대처할 수 없습니다.

하지만 Neural Network LM은 word embedding을 사용하여 단어를 벡터화(vectorize) 함으로써, 강아지와 고양이를 비슷한 vector로 학습하고, 자동차와 고양이 보다 훨씬 높은 유사도를 가지게 합니다. 따라서 NNLM이 훈련 corpus에서 보지 못한 단어의 조합을 보더라도, 비슷한 훈련 데이터로부터 배운 것과 유사하게 대처할 수 있습니다.

Neural Network LM은 많은 형태를 가질 수 있지만 우리는 가장 효율적이고 흔한 형태인 Recurrent Neural Network(RNN)의 일종인 Long Short Term Memory(LSTM)을 활용한 방식에 대해서 짚고 넘어가도록 하겠습니다.

Recurrent Neural Network LM

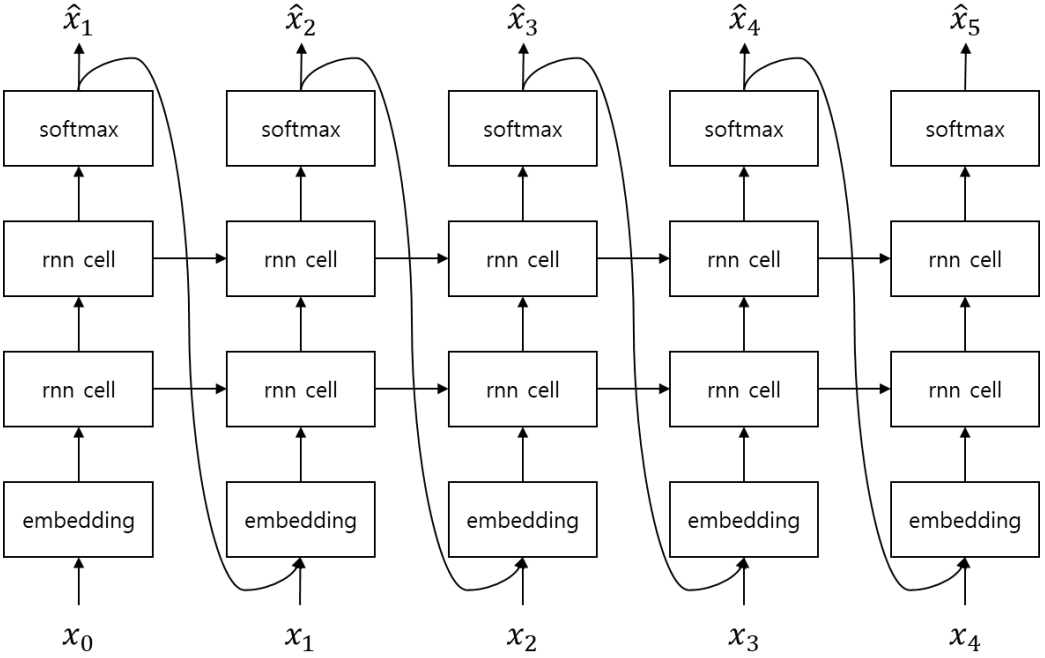


Figure 5:

Recurrent Neural Network Lauguage Model (RNNLM)은 위와 같은 구조를 지니고 있습니다. 기존의 언어모델은 각각의 단어를 descrete한 존재로써 처리하였기 때문에, 문장(word sequence)의 길이가 길어지면 희소성(sparseness)문제가 발생하여 어려운 부분이 있었습니다. 따라서, $n-1$ 이전까지의 단어만 (주로 $n=3$) 조건부로 잡아 확률을 근사(approximation) 하였습니다. 하지만, RNN LM은 단어를 embedding하여 벡터화(vectorize)함으로써, 희소성 문제를 해소하였기 때문에, 문장의 첫 단어부터 모두 조건부에 넣어 확률을 근사 할 수 있습니다.

$$P(w_1, w_2, \cdots, w_k) = \prod_{i=1}^{k} P(w_i | w_{<i})$$

로그를 취하여 표현 해보면 아래와 같습니다.

$$\log P(w_1, w_2, \cdots, w_k) = \sum_{i=1}^{k} \log P(w_i | w_{<i})$$

Implementation

이제 RNN을 활용한 언어모델을 구현 해 보도록 하겠습니다. PyTorch로 구현하기에 앞서, 이를 수식화 해보면 아래와 같습니다. - language_model.py 가 이를 구현 한 코드 입니다.

$$\begin{aligned} X &= \{x_0, x_1, \cdots, x_n, x_{n+1}\} \\ &\quad where\ x_0 = BOS\ and\ x_{n+1} = EOS. \end{aligned}$$

$$\begin{aligned} \hat{x}_{i+1} &= Softmax(Linear_{hidden \to |V|}(RNN(Emb(x_i)))) \\ \hat{X}[1:] &= Softmax(Linear_{hidden \to |V|}(RNN(Emb(X[:-1])))), \\ &\quad where\ |V|\ is\ size\ of\ vocabulary. \end{aligned}$$

위의 수식을 따라가 보면, 문장 X를 입력으로 받아 각 time-step 별(x_i)로 Emb(embedding layer)에 넣어 정해진 차원(dimension)의 embedding vector를 얻습니다. RNN은 해당 embedding vector를 입력으로 받아, hidden size의 vector 형태로 반환 합니다. 이 RNN 출력 vector를 linear layer를 통해 어휘(vocabulary)수 dimension의 vector로 변환 한 후, softmax를 취하여 $\hat{x}_{i+1}$을 구합니다.

여기서 우리는 LSTM을 사용하여 RNN을 대체 할 것이고, LSTM은 여러 층(layer)로 구성되어 있으며, 각 층 사이에는 dropout이 들어가 있습니다. 이 결과($\hat{X}$)를 이전 섹션에서 perplexity와 엔트로피(entropy)와의 관계를 설명하였듯이, cross entropy loss를 사용하여 모델(θ) 최적화를 수행 합니다.

Code

아래의 PyTorch 코드는 저자의 github에서 다운로드 할 수 있습니다. (업데이트 여부에 따라 코드가 약간 달라질 수 있습니다.)

- github repo url: https://github.com/kh-kim/OpenNLMTK

language_model.py

```
import torch
import torch.nn as nn

class LanguageModel(nn.Module):

    def __init__(self, vocab_size, word_vec_dim = 512,
                                    hidden_size = 512,
                                    n_layers = 4,
                                    dropout_p = .2,
                                    max_length = 255
                                    ):
        self.vocab_size = vocab_size
        self.word_vec_dim = word_vec_dim
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.dropout_p = dropout_p
        self.max_length = max_length

        super(LanguageModel, self).__init__()

        self.emb = nn.Embedding(vocab_size, word_vec_dim, padding_idx = 0)
        self.rnn = nn.LSTM(word_vec_dim, hidden_size, n_layers, batch_first = True
```

```
        self.out = nn.Linear(hidden_size, vocab_size, bias = True)
        self.log_softmax = nn.LogSoftmax(dim = 2)

    def forward(self, x):
        # |x| = (batch_size, length)
        x = self.emb(x)
        # |x| = (batch_size, length, word_vec_dim)
        x, (h, c) = self.rnn(x)
        # |x| = (batch_size, length, hidden_size)
        x = self.out(x)
        # |x| = (batch_size, length, vocab_size)
        y_hat = self.log_softmax(x)

        return y_hat
```

data_loader.py

```
from torchtext import data, datasets

BOS = 2
EOS = 3

class DataLoader():

    def __init__(self, train_fn, valid_fn, batch_size = 64,
                                           device = -1,
                                           max_vocab = 99999999,
                                           max_length = 255,
                                           fix_length = None,
                                           use_bos = True,
                                           use_eos = True,
                                           shuffle = True
                                           ):
        super(DataLoader, self).__init__()

        self.text = data.Field(sequential = True,
                               use_vocab = True,
```

```
                                    batch_first = True,
                                    include_lengths = True,
                                    fix_length = fix_length,
                                    init_token = '<BOS>' if use_bos else None,
                                    eos_token = '<EOS>' if use_eos else None
                                    )

            train = LanguageModelDataset(path = train_fn,
                                         fields = [('text', self.text)],
                                         max_length = max_length
                                         )
            valid = LanguageModelDataset(path = valid_fn,
                                         fields = [('text', self.text)],
                                         max_length = max_length
                                         )

            self.train_iter = data.BucketIterator(train,
                                                  batch_size = batch_size,
                                                  device = device,
                                                  shuffle = shuffle,
                                                  sort_key=lambda x: -len(x.text),
                                                  sort_within_batch = True
                                                  )
            self.valid_iter = data.BucketIterator(valid,
                                                  batch_size = batch_size,
                                                  device = device,
                                                  shuffle = False,
                                                  sort_key=lambda x: -len(x.text),
                                                  sort_within_batch = True
                                                  )

            self.text.build_vocab(train, max_size = max_vocab)

    class LanguageModelDataset(data.Dataset):
        """Defines a dataset for machine translation."""

        def __init__(self, path, fields, max_length=None, **kwargs):
            if not isinstance(fields[0], (tuple, list)):
```

```
            fields = [('text', fields[0])]

        examples = []
        with open(path) as f:
            for line in f:
                line = line.strip()
                if max_length and max_length < len(line.split()):
                    continue
                if line != '':
                    examples.append(data.Example.fromlist(
                        [line], fields))

        super(LanguageModelDataset, self).__init__(examples, fields, **kwargs)


if __name__ == '__main__':
    import sys
    loader = DataLoader(sys.argv[1], sys.argv[2])

    for batch_index, batch in enumerate(loader.train_iter):
        print(batch.text)

        if batch_index > 1:
            break
```

trainer.py

```
import time
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.utils as torch_utils

import utils
```

```
def get_loss(y, y_hat, criterion, do_backward = True):
    batch_size = y.size(0)

    loss = criterion(y_hat.contiguous().view(-1, y_hat.size(-1)), y.contiguous().v
    if do_backward:
        # since size_average parameter is off, we need to devide it by batch siz
        loss.div(batch_size).backward()

    return loss

def train_epoch(model, criterion, train_iter, valid_iter, config):
    current_lr = config.lr

    lowest_valid_loss = np.inf
    no_improve_cnt = 0

    for epoch in range(1, config.n_epochs):
        optimizer = optim.SGD(model.parameters(), lr = current_lr)
        print("current learning rate: %f" % current_lr)
        print(optimizer)

        sample_cnt = 0
        total_loss, total_word_count, total_parameter_norm, total_grad_norm = 0, 0
        start_time = time.time()
        train_loss = np.inf

        for batch_index, batch in enumerate(train_iter):
            optimizer.zero_grad()

            current_batch_word_cnt = torch.sum(batch.text[1])
            # Most important lines in this method.
            # Since model takes BOS + sentence as an input and sentence + EOS as
            # x(input) excludes last index, and y(index) excludes first index.
            x = batch.text[0][:, :-1]
            y = batch.text[0][:, 1:]
            # feed-forward
            y_hat = model(x)
```

```
# calcuate loss and gradients with back-propagation
loss = get_loss(y, y_hat, criterion)

# simple math to show stats
total_loss += float(loss)
total_word_count += int(current_batch_word_cnt)
total_parameter_norm += float(utils.get_parameter_norm(model.parameter
total_grad_norm += float(utils.get_grad_norm(model.parameters()))

if (batch_index + 1) % config.print_every == 0:
    avg_loss = total_loss / total_word_count
    avg_parameter_norm = total_parameter_norm / config.print_every
    avg_grad_norm = total_grad_norm / config.print_every
    elapsed_time = time.time() - start_time

    print("epoch: %d batch: %d/%d\t|param|: %.2f\t|g_param|: %.2f\tlos
```

```
    total_loss, total_word_count, total_parameter_norm, total_grad_nor
    start_time = time.time()

    train_loss = avg_loss

# Another important line in this method.
# In orther to avoid gradient exploding, we apply gradient clipping.
torch_utils.clip_grad_norm_(model.parameters(), config.max_grad_norm)
# Take a step of gradient descent.
optimizer.step()

sample_cnt += batch.text[0].size(0)
```

```
        if sample_cnt >= len(train_iter.dataset.examples) * config.iter_ratio
            break

    sample_cnt = 0
    total_loss, total_word_count = 0, 0

    model.eval()
    for batch_index, batch in enumerate(valid_iter):
        current_batch_word_cnt = torch.sum(batch.text[1])
        x = batch.text[0][:, :-1]
        y = batch.text[0][:, 1:]
        y_hat = model(x)

        loss = get_loss(y, y_hat, criterion, do_backward = False)

        total_loss += float(loss)
        total_word_count += int(current_batch_word_cnt)

        sample_cnt += batch.text[0].size(0)
        if sample_cnt >= len(valid_iter.dataset.examples):
            break

    avg_loss = total_loss / total_word_count
    print("valid loss: %.4f\tPPL: %.2f" % (avg_loss, np.exp(avg_loss)))

    if lowest_valid_loss > avg_loss:
        lowest_valid_loss = avg_loss
        no_improve_cnt = 0
    else:
        # decrease learing rate if there is no improvement.
        current_lr /= 10.
        no_improve_cnt += 1

    model.train()

    model_fn = config.model.split(".")
    model_fn = model_fn[:-1] + ["%02d" % epoch, "%.2f-%.2f" % (train_loss, np
```

```
        # PyTorch provides efficient method for save and load model, which uses
        torch.save({"model": model.state_dict(),
                    "config": config,
                    "epoch": epoch + 1,
                    "current_lr": current_lr
                    }, ".".join(model_fn))

        if config.early_stop > 0 and no_improve_cnt > config.early_stop:
            break
```

utils.py

```
import torch

def get_grad_norm(parameters, norm_type = 2):
    parameters = list(filter(lambda p: p.grad is not None, parameters))

    total_norm = 0

    try:
        for p in parameters:
            param_norm = p.grad.data.norm(norm_type)
            total_norm += param_norm ** norm_type
        total_norm = total_norm ** (1. / norm_type)
    except Exception as e:
        print(e)

    return total_norm

def get_parameter_norm(parameters, norm_type = 2):
    total_norm = 0

    try:
        for p in parameters:
            param_norm = p.data.norm(norm_type)
            total_norm += param_norm ** norm_type
```

```
        total_norm = total_norm ** (1. / norm_type)
    except Exception as e:
        print(e)

    return total_norm
```

train.py

```
import argparse

import torch
import torch.nn as nn

from data_loader import DataLoader
import data_loader
from language_model import LanguageModel as LM
import trainer

def define_argparser():
    p = argparse.ArgumentParser()

    p.add_argument('-model', required = True)
    p.add_argument('-train', required = True)
    p.add_argument('-valid', required = True)
    p.add_argument('-gpu_id', type = int, default = -1)

    p.add_argument('-batch_size', type = int, default = 64)
    p.add_argument('-n_epochs', type = int, default = 20)
    p.add_argument('-print_every', type = int, default = 50)
    p.add_argument('-early_stop', type = int, default = 3)
    p.add_argument('-iter_ratio_in_epoch', type = float, default = 1.)

    p.add_argument('-dropout', type = float, default = .1)
    p.add_argument('-word_vec_dim', type = int, default = 256)
    p.add_argument('-hidden_size', type = int, default = 256)
    p.add_argument('-max_length', type = int, default = 80)
```

```
    p.add_argument('-n_layers', type = int, default = 4)
    p.add_argument('-max_grad_norm', type = float, default = 5.)
    p.add_argument('-lr', type = float, default = 1.)
    p.add_argument('-min_lr', type = float, default = .000001)

    config = p.parse_args()

    return config

if __name__ == '__main__':
    config = define_argparser()

    loader = DataLoader(config.train,
                        config.valid,
                        batch_size = config.batch_size,
                        device = config.gpu_id,
                        max_length = config.max_length
                        )
    model = LM(len(loader.text.vocab),
                word_vec_dim = config.word_vec_dim,
                hidden_size = config.hidden_size,
                n_layers = config.n_layers,
                dropout_p = config.dropout,
                max_length = config.max_length
                )

    # Let criterion cannot count EOS as right prediction, because EOS is easy to
    loss_weight = torch.ones(len(loader.text.vocab))
    loss_weight[data_loader.EOS] = 0
    criterion = nn.NLLLoss(weight = loss_weight, size_average = False)

    print(model)
    print(criterion)

    trainer.train_epoch(model,
                        criterion,
                        loader.train_iter,
                        loader.valid_iter,
```

```
config
)
```

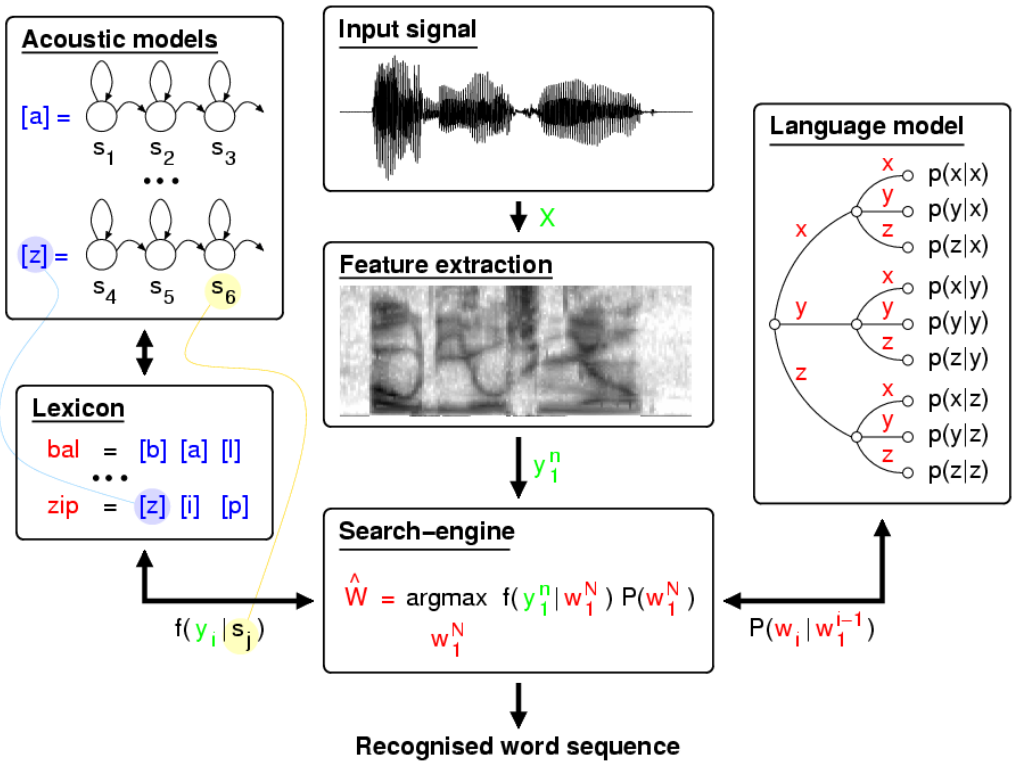
Conclusion

NNLM은 word embedding vector를 사용하여 희소성(sparseness)을 해결하여 큰 효과를 보았습니다. 따라서 훈련 데이터셋에서 보지 못한 단어(unseen word sequence)의 조합에 대해서도 훌륭한 대처가 가능합니다. 하지만 그만큼 연산량에 있어서 n-gram에 비해서 매우 많은 대가를 치루어야 합니다. 단순히 table look-up 수준의 연산량을 필요로 하는 n-gram방식에 비해서 NNLM은 다수의 matrix 연산등이 포함된 feed forward 연산을 수행해야 하기 때문입니다. 그럼에도 불구하고 GPU의 사용과 점점 빨라지는 하드웨어 사이에서 NNLM의 중요성은 커지고 있고, 실제로도 많은 분야에 적용되어 훌륭한 성과를 거두고 있습니다. # Applications

사실 언어모델 자체를 단독으로 사용하는 경우는 굉장히 드뭅니다. 그렇다면 왜 언어모델이 중요할까요? 하지만, 자연어생성(Natural Language Generation, NLG)의 가장 기본이라고 할 수 있습니다. NLG는 현재 딥러닝을 활용한 NLP 분야에서 가장 큰 연구주제 입니다. 기계번역에서부터 챗봇까지 모두 NLG의 영역에 포함된다고 할 수 있습니다. 이러한 NLG의 기본 초석이 되는 것이 바로 언어모델(LM) 입니다. 따라서, LM 자체의 활용도는 그 중요성에 비해 떨어질지언정, LM의 중요성과 그 역할은 부인할 수 없습니다. 대표적인 활용 분야는 아래와 같습니다.

Automatic Speech Recognition (ASR)

음성인식(Speech recognition) 시스템을 구성할 때, 언어모델은 중요하게 쓰입니다. 사실 실제 사람의 경우에도 말을 들을 때 언어모델이 굉장히 중요하게 작용합니다. 어떤 단어를 발음하고 있는지 명확하게 알아듣지 못하더라도, 머릿속에 저장되어 있는 언어모델을 이용하여 알아듣기 때문입니다. 예를 들어, 우리는 갑자기 쌩뚱맞은 주제로 대화를 전환하게 되면 보통 한번에 잘 못알아듣는 경우가 많습니다. 컴퓨터의 경우에도 음소별 분류(classification)의 성능은 이미 사람보다 뛰어납니다. 다만, 사람에 비해 주변 문맥(context) 정보를 활용할 수 있는 능력, **눈치**가 없기 때문에 음성인식률이 떨어지는 경우가 많습니다. 따라서, 그나마 좋은 언어모델을 학습하여 사용함으로써, 음성인식의 정확도를 높일 수 있습니다.



(Traditional ASR System based on WFST, Image from web)

아래는 음성인식 시스템의 수식을 개략적으로 나타낸 것 입니다. 음성 신호 X가 주어졌을 때 확률을 최대로 하는 문장 $\hat{Y}$를 구하는 것이 목표 입니다.

$$\hat{Y} = argmax P(Y|X) = argmax \frac{P(X|Y)P(Y)}{P(X)},$$

where X is an audio signal and Y is a word sequence, $Y = \{y_1, y_2, \cdots, y_n\}$.

그럼 베이즈 정리(Bayes Theorem)에 의해서 수식은 AM과 LM 부분으로 나뉠 수 있습니다. 그리고 밑변(evidence) $P(X)$는 날려버릴 수 있습니다.

$$argmax \frac{P(X|Y)P(Y)}{P(X)} = argmax P(X|Y)P(Y)$$

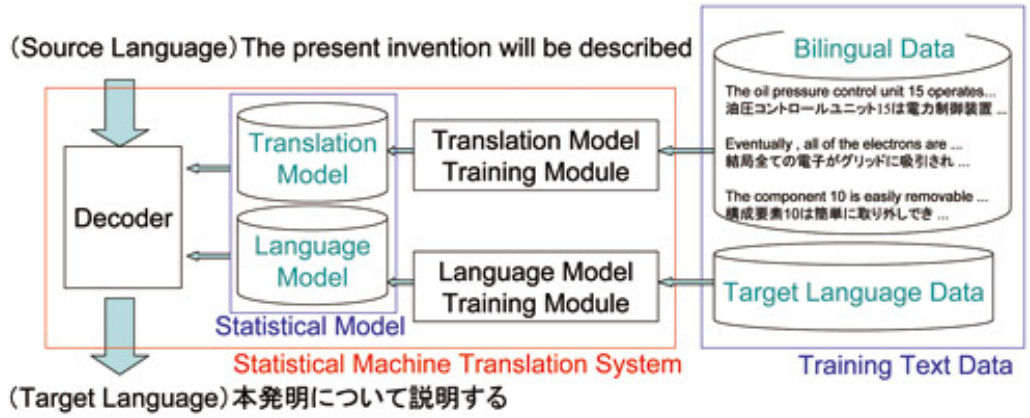
여기서 $P(X|Y)$가 음향모델(Acoustic Model, AM)을 의미하고, $P(Y)$는 언어모델(LM)을 의미 합니다. 즉, $P(Y)$는 문장의 확률을 의미하고, $P(X|Y)$는

문장(단어 시퀀스 또는 음소 시퀀스)이 주어졌을 때, 해당 음향 시그널이 나타날 확률을 나타냅니다.

이처럼 LM은 AM과 결합하여, 문맥 정보에 기반하여 다음 단어를 예측하도록 함으로써, 음성인식의 성능을 더울 향상 시킬 수 있고, 매우 중요한 역할을 차지하고 있습니다.

Machine Translation (MT)

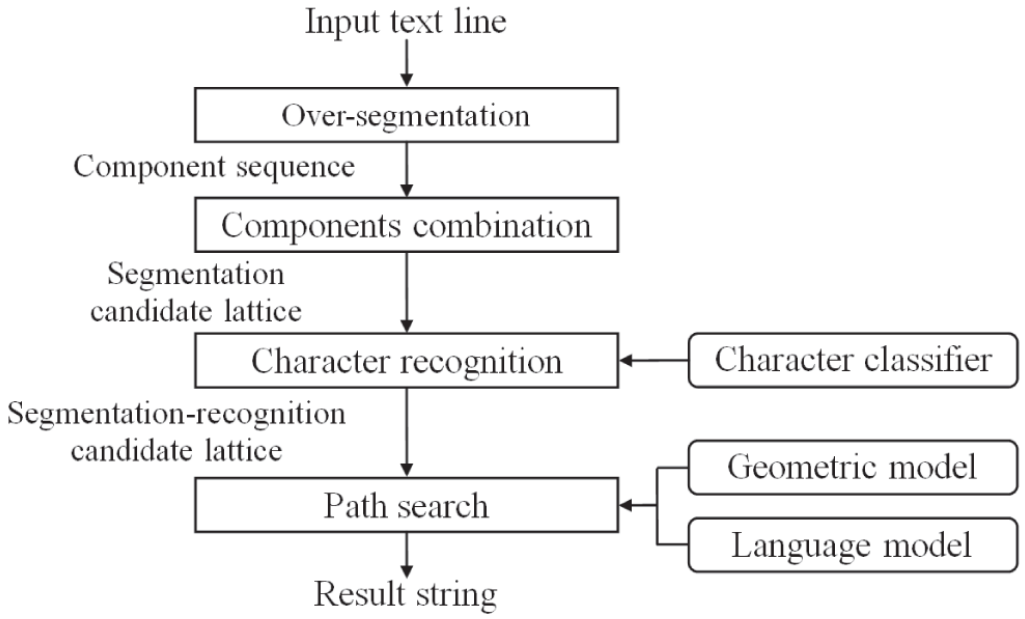
번역 시스템을 구성 할 때에도 언어모델은 중요한 역할을 합니다. 기존 통계기반번역(Statistical Machine Translation, SMT) 시스템에서는 위의 음성인식 시스템과 매우 유사하게 LM이 번역모델(translation model)과 결합하여 자연스러운 문장을 만들어내도록 동작 하였으며, 신경망기계번역(Neural Machine Translation, NMT)의 경우에도 LM이 마찬가지로 매우 중요한 역할을 차지 합니다. 더 자세한 내용은 다음 챕터에서 다루도록 하겠습니다.



(Statistical Machine Translation, SMT, system, Image from web)

Optical Character Recognition (OCR)

광학문자인식 시스템을 만들 때에도 언어모델이 사용 됩니다. 사진에서 추출하여 글자를 인식 할 때에 각 글자(character) 간의 확률을 정의하면 훨씬 더 높은 성능을 얻어낼 수 있습니다. 따라서, OCR에서도 언어모델의 도움을 받아 글자나 글씨를 인식합니다.



(OCR system, Image from web)

Natural Language Generation (NLG)

사실 위에 나열한 ASR, MT, OCR도 주어진 정보를 바탕으로 문장을 생성해내는 NLG에 속한다고 볼 수 있습니다. 이외에도 NLG가 적용 될 수 있는 영역은 굉장히 많습니다. 기계학습의 결과물로써 문장을 만들어내는 작업은 모두 NLG의 범주에 속한다고 볼 수 있습니다. 예를 들어, 주어진 정보를 바탕으로 뉴스 기사를 쓸 수도 있고, 주어진 뉴스 기사를 요약하여 제목을 생성 해 낼 수도 있습니다. 또한, 사용자의 응답에 따라 대답을 생성 해 내는 chatbot도 생각 해 볼 수 있습니다.

Others

이외에도 여러가지 영역에 정말 다양하게 사용됩니다. 검색엔진에서 사용자가 검색어를 입력하는 도중에 밑에 drop-down으로 제시되는 검색어 완성 등에도 언어모델이 사용 될 수 있습니다.