NPLM

- 임베딩 분야의 선구자적 모델
- 통계 기반의 전통적 언어 모델의 한계를 극복하는 과정에서 탄생
- 기존 언어 모델의 단점
  1. 학습데이터에 존재하지 않는 n-gram 포함된 문장 나타날 확률 0으로 부여
     - 백오프, 스무딩으로 보완 가능하나 불완전함
  2. 문장의 장기 의존성 포착하기 어려움 (n을 5이상으로 설정 불가능)
     - n 커질수록 등장확률 0인 시퀀스 급증
  3. 단어 / 문장 간 유사도 계산 불가능
- 위의 한계들을 일부 극복
- NPLM 자체적으로 단어 임베딩 역할 수행 가능

- NPLM의 기본 아이디어
  - 직전까지 등장한 n-1개의 단어들로 다음 단어를 맞추는 n-gram 모델

NPLM : Neural Probabilistic Language Model

NPLM 기본 구조

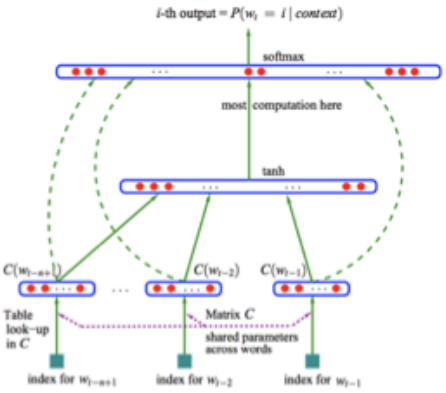


Figure 1: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

1. 행렬 C의 초기 값은 랜덤.

$$C = \begin{bmatrix} 11 & 18 & 25 \\ 10 & 12 & 19 \\ 4 & 6 & 13 \\ 23 & 5 & 7 \\ 17 & 24 & 1 \end{bmatrix} \begin{matrix} \text{발} \\ \text{없는} \\ \text{말} \\ \text{이} \\ \text{천리} \end{matrix}$$

2. 각 단어에 해당하는 원-핫 벡터를 행렬 C와 내적 (Look-up table)

$$\begin{bmatrix} 0 & 0 & 0 & \mathbf{1} & 0 \end{bmatrix} \times \begin{bmatrix} 11 & 18 & 25 \\ 10 & 12 & 19 \\ 4 & 6 & 13 \\ \mathbf{23} & \mathbf{5} & \mathbf{7} \\ 17 & 24 & 1 \end{bmatrix} = \begin{bmatrix} 23 & 5 & 7 \end{bmatrix}$$

Page 01

NPLM : Neural Probabilistic Language Model

NPLM 기본 구조

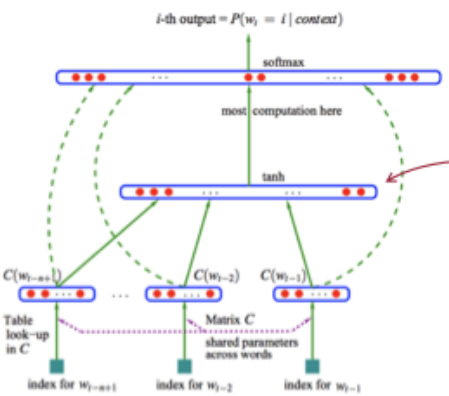


Figure 1: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

3. 단어들의 벡터를 붙임(concatenate).

$$x = [x_{t-1}, x_{t-2}, \ldots, x_{t-n+1}]$$

$$x = [\underbrace{10, 12, 19}_{\text{없는}}, \underbrace{4, 6, 13}_{\text{말}}, \underbrace{23, 5, 7}_{\text{이}}]$$

4. 위에서 계산된 x가 모델의 입력으로 input layer, hidden layer, output layer를 거쳐 스코어 벡터 계산

$$y_{w_t} = b + Wx + Utanh(d + Hx)$$

Page 01

NPLM : Neural Probabilistic Language Model

NPLM 기본 구조

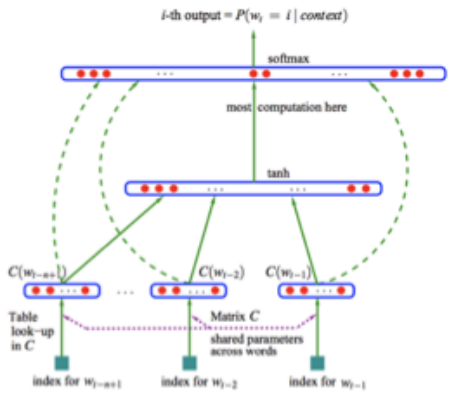


Figure 1: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

5. 계산된 스코어 벡터(y)에 소프트맥스 함수를 적용하여 원래 단어의 인덱스와 비교해 loss를 계산하여 역전파

$$y_{w_t} = b + Wx + Utanh(d + Hx)$$

↓ softmax

$$P(w_t|w_{t-1}, ..., w_{t-n+1}) = \frac{exp(y_{w_t})}{\sum_i exp(y_t)}$$

Page 01

- NPLM의 학습
  - 단어 시퀀스가 주어졌을 때 다음 단어가 무엇인지 맞추는 과정에서 학습

$$P(w_t|w_{t-1}, \ldots, w_{t-n+1}) = \frac{exp(Y_{wt})}{\sum_{i=1} exp(Y_i)}$$

> y_{w_t} 는 w_t라는 단어에 해당하는 점수 벡터 y_{w_t}의 크기는 |V| ; 말 뭉치 전체의 어휘단어 수

1. NPLM의 출력
   - |V|차원의 스코어벡터 y_{w_t}에 소프트맥스 함수를 적용한 |V|차원의 확률벡터
   - NPLM은 확률벡터에서 가장 높은 요소의 인덱스에 해당하는 단어가 실제 정답 단어에 일치하도록 학습
2. NPLM의 입력
   - 문장 내 t번째 단어(w_t)에 대응하는 단어 벡터 x_{w_t}를 만드는 과정

$$X_t = C(W_t)$$

   - |V|*m 크기의 행렬 C에서 w_t에 해당하는 벡터를 참조하는 형태

> |V| = 어휘 집합 크기
> m = Xt의 차원 수
> C 행렬의 원소 값 = 초기에 랜덤 설정
> - 참조의 의미 : 행렬 C와 w_t에 해당하는 원-핫 벡터를 내적한 결과

3. 예시
   - 3개의 단어 a, b, c가 주어졌을 때 d라는 단어를 예측해야 하는 상황
     A. 세 단어의 인덱스 값을 확인 - 세 단어에 해당하는 열벡터를 C에서 참조한 뒤 묶어서 NPLM의 입력벡터 X 제작
     B. NPLM의 스코어 벡터 계산

$$Y_{w_t} = b + W_x + U_{tanh}(d + H_x)$$

     C. y_{w_t}에 소프트맥스 함수 적용 - 정답 단어의 인덱스와 비교하여 역전파 방식으로 학습
     D. 학습 종료 후 - 행렬 C를 각 단어에 해당하는 m차원 임베딩으로 사용

학습 파라미터의 차원수

$$H \in R_{h(n-1)m}$$
$$X \in R_{(n-1)m}$$
$$d \in R_h$$
$$W \in R_{|V|(n-1)m}$$
$$U \in R_{|V|h}$$
$$b \in R_{|V|}$$
$$Y \in R_{|V|}$$
$$C \in R_{|V|m}$$

    - 다른 임베딩 기법에 비해 학습해야 하는 파라미터 종류 상당
    - 다른 임베딩 기법에 비해 파라미터 크기 또한 상당함
        - 이후의 단어 임베딩 기법은 추정 대상 파라미터 줄이고 품질은 높이는 방향으로 발전
- NPLM의 의미정보
    - 단어의 의미를 임베딩에 반영하는 방식

1. C행렬의 행 벡터들은 단어를 맞추는 과정에서 발생한 학습 손실을 최소화하는 그래디언트를 받아 동일하게 업데이트 함
2. 동일 단어를 맞추는 n-gram에 포함되는 단어들의 벡터 - 벡터 공간에서 같은 방향으로 움직임
3. 문장 내 모든 단어들을 하나씩 훑으면서 말뭉치 전체를 학습함
    - 결과적으로 NPLM 모델의 C행렬에 각 단어의 문맥정보 내재

- NPLM의 장점
    - 말뭉치에 있는 문맥이 비슷한 다른 문장을 참고하여 확률을 부여함
    - 기존 통계 기반 모델 - 학습 데이터에 한 번도 등장하지 않은 패턴에 대해서는 그 등장확률을 0으로 부여

Word2Vec

- Skip-Gram, CBOW모델을 근간으로 네거티브 샘플링 등 학습 최적화 기법을 제안
- 모델 기본 구조

1. CBOW
   - 문맥 단어들을 통해 타깃 단어를 맞추는 과정에서 학습
   - {입력학습데이터, 출력학습데이터} = {문맥 단어 4개, 타깃단어}
2. Skip-Gram
   - 타깃 단어를 가지고 주변 문맥 단어가 무엇일지 예측하는 과정에서 학습
   - 학습 데이터 = {타깃 단어, 타깃 직전 2번째 단어}, {타깃 단어, 타깃 직전 단어}, {타깃 단어, 타깃 다음 단어}, {타깃 단어, 타깃 다음 2번째 단어}
   - 동일 말뭉치로 더 많은 학습 데이터 확보 가능 - CBOW보다 임베딩 품질 좋은 경향

- 학습데이터 구축
  - 포지티브 샘플 : 타깃단어(t)와 주변에 실제로 등장한 문맥단어(c) 쌍
  - 네거티브 샘플 : 타깃단어와 주변에 등장하지 않은, 말뭉치 전체에서 랜덤 추출한 단어 쌍
    - 윈도우 ; 타깃 단어 앞, 뒤로 고려할 단어의 개수
  - Skip-gram모델 : 같은 말뭉치를 두고 많은 수의 학습데이터 쌍 만들 수 있음
    1. 처음 제안시
       - 타깃 단어가 주어진 경우 문맥단어가 무엇일지 맞추는 과정에서 학습
         - 정답 문맥 단어의 등장확률 높이고 나머지 단어의 확률 낮춰야 함
         - 어휘 집합 단어의 수는 대략 수십만개에 달하는데 모두 계산할 경우 비효율적
           - 소프트맥스로 인해 계산량 큼
    2. 이후 제안
       - 타깃단어 - 문맥단어 쌍이 주어진 경우 해당 쌍을 포지티브/네거티브 샘플로 이진분류하는 과정에서 학습 ; 네거티브 샘플링
         - 1개의 포지티브 샘플 & k개의 네거티브 샘플 계산 - 기존 방식보다 계산량 급감
           - 현재 방식) 매 단계 2차원의 시그모이드를 k+1회 계산
           - 기존 방식) 매 단계 (어휘 집합 수)차원의 소프트맥스를 1회 계산
         - 적절한 k 값의 범위

> 작은 데이터의 경우 : 5 < k < 20
> 큰 데이터의 경우 : 2 < k < 5

  - 네거티브 샘플
    - 말뭉치에 자주 등장하지 않는 희귀한 단어가 샘플로 잘 뽑힐 수 있도록 설계
    - 네거티브 샘플 확률

$$\frac{U(w_i)^{3/4}}{\sum_{j=0}^{n} U(w_j)^{3/4}}$$

      - U(Wi) = 해당 단어의 유니그램 확률 (해당 단어 빈도 / 전체 단어 수)
  - 서브샘플링
    - 자주 등장하는 단어 학습에서 제외
    - Skip-Gram모델 - 말뭉치에서 많은 수의 학습데이터 쌍을 제작할 수 있으므로 고빈도 단어를 등장 횟수만큼 모두 학습시키는 경우 매우 비효율적
    - 서브 샘플링 확률

$$1 - \sqrt{\frac{t}{f(w_i)}}$$

      - t = 하이퍼파라미터 (논문에서는 10의 -5승으로 설정)
      - f(Wi) = 단어 Wi의 빈도
      - 등장확률이 0.01로 높은 단어

        - 서프샘플링 확률 대략 0.968로 100번의 학습 기회 중 약 96번은 학습에서 제외됨
    - 등장확률이 낮아 서브샘플링확률이 0에 가까운 경우
        - 해당 단어 등장 시 항상 학습
    - 학습량을 효과적으로 줄여 계산량을 감소시키는 전략
- 모델 학습
    - Skip-Gram 모델

1. 타깃 단어와 문맥 단어 쌍이 실제 포지티브 샘플일 경우
    - 아래의 조건부확률 최대화하여 포지티브 샘플이 입력 된 경우 포지티브 샘플이라고 맞춰야 함

$$P(+|t,c) = \frac{1}{1 + exp(-U_t V_c)}$$

    - P(+|t,c) = t,c가 포지티브 샘플일 확률 (t 주변에 c가 존재할 확률)
    - 학습 파라미터 U, V 행렬
    - 크기 = {어휘집합크기 (|V|)} x {임베딩 차원 수 (d)}
        - U, V는 각각 타깃 단어와 문맥 단어에 대응
        - NPLM에 비해 학습 파라미터의 크기와 종류 대폭 줄어듦
    - P(+|t,c) 최대화 위해서

> 분모를 줄여야 함
> exp(-UtVc)를 줄여야 함
> Ut와 Vc의 내적값을 키워야 함
> Ut와 Vc의 코사인 유사도 향상

2. 타깃 단어와 문맥 단어 쌍이 실제 네거티브 샘플일 경우
    - 아래의 조건부확률 최대화하여 네거티브 샘플이 입력 된 경우 네거티브 샘플이라고 맞춰야 함

$$P(-|t,c) = 1 - P(+|t,c) = \frac{exp(-U_t V_c)}{1 + exp(-U_t V_c)}$$

    - P(-|t,c) = t,c가 네거티브 샘플일 확률 (c가 t와 무관한 랜덤 샘플일 확률)
    - P(-|t,c) 최대화 위해서

> 분자 최대화 필요
> exp(-UtVc)를 키워야 함
> Ut와 Vc의 내적값을 줄여야 함
> Ut와 Vc의 코사인 유사도 감소

3. 로그우도 함수 - 최대화 필요

$$L(\theta) = \log P(+|t_p, c_p) + \sum_{i=1}^{k} P(-|t_{n_i}, c_{n_i})$$

    - 모델 파라미터 θ를 한 번 업데이트할 때 1쌍의 포지티브 샘플과 k개의 네거티브 샘플이 학습됨을 의미함
    - 최대화 하는 과정에서 결과적으로 말뭉치의 분포 정보를 단어 임베딩에 함축
    - 타깃 단어 벡터 ut는 행렬 U, 문맥 단어 벡터 vc는 행렬 V에서 참조
        - 모델 학습 완료 후
            A. U만 d차원의단어임베딩으로활용
            B. $U + V^T$행렬을임베딩으로활용
            C. U와 V^T를이어붙여$2d$차원의단어임베딩으로활용

- 튜토리얼
    - model = Word2Vec(corpus, size = 100, workers = 4, sg = 1)

corpus = 말뭉치
size = Word2Vec 임베딩의 차원 수
workers = CPU 스레드 개수
sg = 0 ; CBOW 모델 / 1 ; Skip-Gram 모델

In [ ]:

```
# 모델 학습

corpus_fname = "/notebooks/embedding/data/tokenized/corpus_mecab.txt"
model_fname = "/notebooks/embedding/data/word-embeddings/word2vec/word2vec"

from gensim.models import Word2Vec
corpus = [sent.strip().split(" ") for sent in open(corpus_fname, 'r').readlines
()]
model = Word2Vec(corpus, size = 100, worders = 4, sg = 1)
model.save(model_fname)
```

In [ ]:

```
# 코사인 유사도 상위 단어 목록 체크 코드

from models.word_eval import WordEmbeddingEvaluator
model = WordEmbeddingEvaluator("/notebooks/embedding/data/word-embeddings/word2v
ec/word2vec",
                               method = "word2vec", dim = 100,
                               tokenizer_name = "mecab")
model.most_similar("희망", topn = 5)
```

FastText

- 페이스북에서 개발, 공개한 단어 임베딩 기법
- 각 단어를 문자 단위 n-gram으로 표현 (이외 내용은 Word2Vec과 동일)
  - 예) 시나브로 - 문자 단위 n-gram(n=3)
    - 임베딩 = 5개의 문자단위 n-gram 벡터의 합

$$U_{시나브로} = Z_{<시나} + Z_{시나브} + Z_{나브로} + Z_{브로>} + Z_{<시나브로>}$$

$$\therefore U_t = \sum_{g \in G_t} Z_g$$

  - 네거티브 샘플링 기법 활용
  - 타깃단어(t), 문맥단어(c) 쌍을 학습할 때 타깃 단어에 속한 문자단위 n-gram 벡터(Z)들을 모두 업데이트
- 모델 기본 구조
  1. 포지티브 샘플 주어진 경우 아래의 조건부확률 최대화하여 포지티브 샘플이라고 맞춰야 함

$$P(+|t,c) = \frac{1}{1+exp(-U_t V_c)} = \frac{1}{1+exp(-\sum_{g \in G_t} Z_g^T V_c)}$$

     - FastText 모델에서 t,c가 포지티브 샘플일 확률
     - P(+|t,c) 최대화 위해서
       - 분모를 최소화
       - Z와 Vc 간 내적값을 높여야 함
       - 문자단위 n-gram과 문맥단어의 코사인 유사도 높여야 함
  2. 네거티브 샘플 주어진 경우 아래의 조건부확률 최대화하여 네거티브 샘플이라고 맞춰야 함

$$P(-|t,c) = 1 - P(+|t,c) = \frac{exp(-U_t V_c)}{1+exp(-U_t V_c)} = \frac{exp(-\sum_{g \in G_t} Z_g^T V_c)}{1+exp(-\sum_{g \in G_t} Z_g^T V_c)}$$

     - FastText 모델에서 t,c가 네거티브 샘플일 확률
     - P(-|t,c) 최대화 위해서
       - 분자를 최대화
       - Z와 Vc 간 내적값을 낮춰야 함
       - 문자단위 n-gram과 문맥단어 네거티브 샘플의 코사인 유사도 낮춰야 함
  3. 로그우도 함수 - 최대화해야 함

$$L(\theta) = \log P(+|t_p, c_p) + \sum_{i=1}^{k} P(-|t_{n_i}, c_{n_i})$$

     - 모델을 한 번 업데이트할 때 1개의 포지티브 샘플, k개의 네거티브 샘플 학습함
- 튜토리얼
  - FastTest 모델의 기본 임베딩 차원 수 = 100
  - 조사, 어미가 발달한 한국어에 좋은 성능 발휘
    - 용언의 활용이나 관계된 어미들이 벡터 공간 상 가깝게 임베딩됨
    - 오타, 미등록 단어에 강함
    - 각 단어의 임베딩을 문자단위 n-gram 벡터의 합으로 표현하기 때문
    - 예) '서울'이 어휘집합에 포함되어 있다면 미등록단어인 '서울특별시'의 임베딩 추정 가능
    - 다른 단어 임베딩 기법 ; 미등록 단어 벡터 추출 불가능

In [ ]:

```
# bash
# 데이터 다운로드
  git pull origin master
  bash preprocess.sh dump-tokenized

# 데이터 합치기
  cat data/tokenized/wiki_ko_mecab.txt data/tokenzied/ratings_mecab.txt data/tok
enized/korquad_mecab.txt > data/tokenized/corpus_mecab.txt
# FastText Skip-gram 모델 학습

mkdir -p data/word-embeddings/fasttext
models/fastText/fasttext skip-gram -input data/tokenized/corpus_mecap.txt -outpu
t data/word-embeddings/fasttext/fasttext
```

In [ ]:

```
# python
# FastText Skip-gram 모델의 코사인 유사도 상위 단어 목록 체크

from models.word_eval import WordEmbeddingEvaluator
model = WordEmbeddingEvaluator(
    vecs_txt_fname = "data/word-embeddings/fasttext/fasttext.vec",
    vecs_bin_fname = "data/word-embeddings/fasttext/fasttext.bin",
    method = "fasttext", dim = 100, tokenizer_name = "mecab")

model.most_similar("희망", topn = 5)
```

- 한글 자소와 FastText
  - FastText ; 문자 단위 n-gram 사용 - 한글에 잘 어울림
  - 한글 ; 자소 단위 분해 가능 - 자소 각각을 하나의 문자로 보고 FastText 시행 가능
  - 자소단위 FastText 사용 시
    - 쿼리 단어의 자소 변환 & 상위 유사 단어의 음절 원복 과정 추가됨

In [ ]:

```
# 한글 자소분해 예시
from preprocess import jama_sentence, get_tokenizer
tokenizer = get_tokenizer("mecab")
tokens = " ".join(tokenizer.morphs("나는 학교에 간다"))
print(jamo_sentence(tokens))
# 결과
# ㄴ ㅏ-ㄴ ㅡ ㄴ ㅎ ㅏ ㄱ ㄱ ㅛ-ㅇ ㅔ ㄱ ㅏ ㄴ ㄷ ㅏ-
```

In [ ]:

```
# 미등록 단어에 대한 자소단위 FastText 임베딩 체크
model._is_in_vocabulary("서울특별시")
model.get_word_vector("서울특별시")
model.most_similar("서울특별시", topn = 5)
```

잠재의미분석 (LSA)

- 단어-문서 행렬, TF-IDF 행렬, 단어-문맥 행렬 등 큰 행렬에 대한 차원 축소 방법의 일종
  - 특이값 분해 수행하여 데이터 차원 수 감소시킴
  - 계산의 효율성 향상
  - 행간에 숨어있는 잠재의미 도출하기 위한 방법론
- 단어-문서 행렬, 단어-문맥 행렬 등에 특이값 분해 시행 후 도출되는 행 벡터 - 단어 임베딩으로 사용 가능
- GloVe, Swibel과 더불어 행렬 분해 기반의 기법으로 분류됨
- PPMI행렬
  - 점별 상호 정보량 (PMI) 행렬의 특수한 버전

$$PMI(A, B) = \log \frac{P(A, B)}{P(A) * P(B)}$$

    - 두 확률변수 간 상관성을 계량화한 지표
    - 자연어 처리 분야
      - 두 단어의 등장이 독립을 가정했을 때 대비 얼마나 자주 같이 등장하는지 수치화
  - 분자 < 분모 일 경우
    - PMI < 0 (= A, B 동시에 등장할 확률이 두 단어가 독립일 때 보다 작을 때)
      - 이 경우 보유한 말뭉치의 크기가 충분히 크지 않은 한 신뢰하기 어려움
        - 보통의 말뭉치에서 단어 하나의 등장확률 =< 0.001
        - A, B의 등장확률이 0.0001로 동일한 경우 PMI < 0 이려면 두 단어 동시 등장 확률이 10억 분의 1 로 적어야 함
        - A, B 단 한번도 동시에 등장하지 않는 경우
          - $PMI(A, B) = log0 = -\infty$
  - PMI의 한계(음의 경우)로 인해 자연어 처리 분야에서는 양의 점별 상호 정보량 (PPMI) 활용
    - PMI가 음수일 경우 그 값을 신뢰하기 어려우 0으로 치환하여 무시함

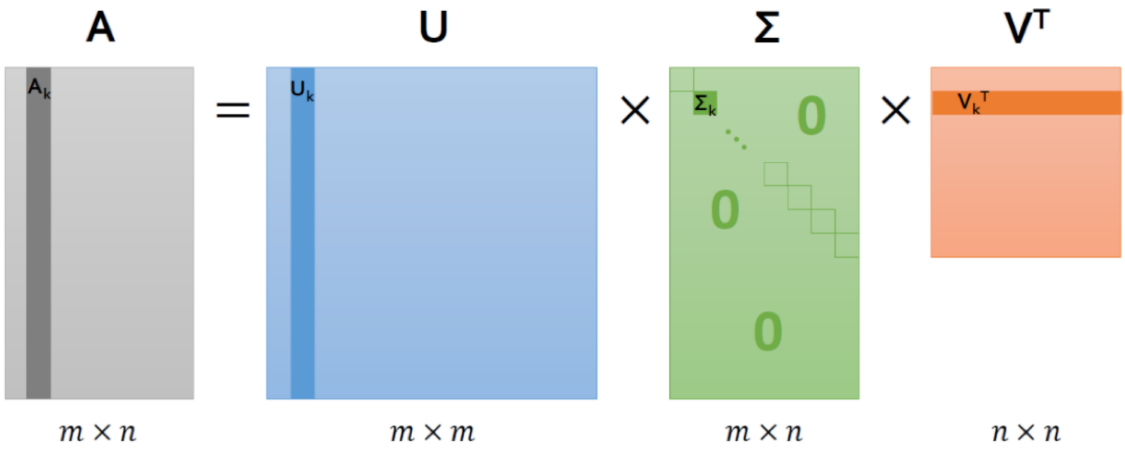
$$PPMI(A, B) = max(PMI(A, B), 0)$$

  - Shifted PMI
    - PMI에서 log k 를 빼준 값

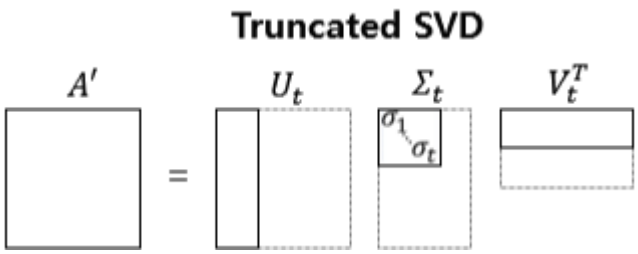
$$SPMI(A, B) = PMI(A, B) - \log k$$

k = 임의의 양의 상수

- 행렬 분해로 이해하는 잠재의미분석 (LSA)
  - 특이값 분해
    - m*n 크기의 임의의 사각행렬 A를 아래와 같이 분해하는 것



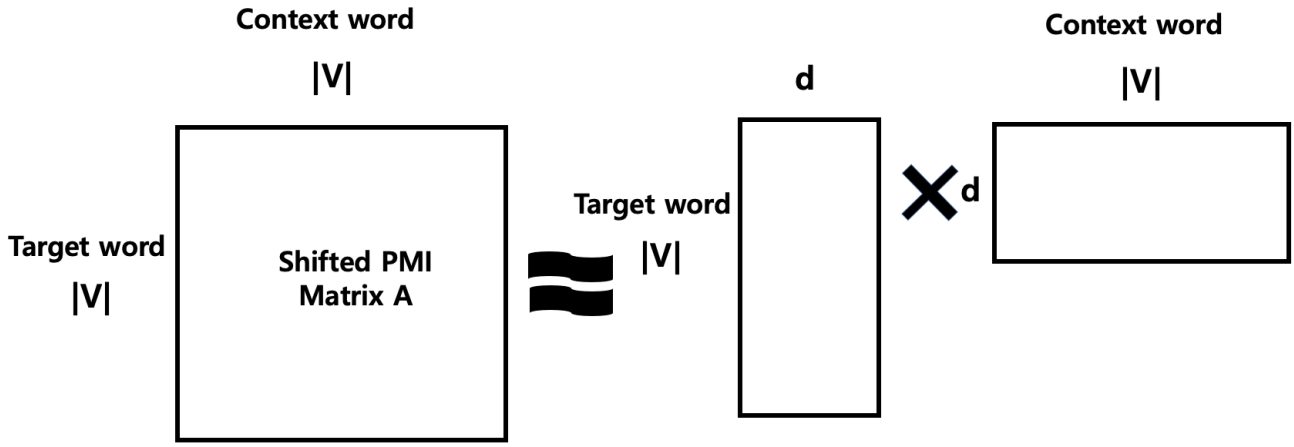
  - truncated SVD
    - 특이값(Σ의 대각성분) 중 가장 큰 d개만 가지고 해당 특이값에 대응하는 특이벡터들로 원래 행렬 A를 근사



A' = m *n*
U' = *m* d
Σ = d *d*
V' = *d* n

예) m개 단어, n개 문서로 이뤄진 단어-문서 행렬에 truncated SVD(LSA)수행 가정
- U' = 단어 임베딩, V' = 문서 임베딩
  - n개 문서로 표현된 단어 벡터들이 U'에서 d차원만으로 표현 가능해짐
  - m개 단어로 표현된 문서 벡터들이 V'에서 d차원만으로 표현 가능해짐

- LSA 적용시
  1. 단어, 문맥 간 내재적인 의미 효과적으로 보존 가능
     - 문서 간 유사도 측정 등 모델 성능 향상에 도움
  2. 입력 데이터의 노이즈, 희소성 줄일 수 있음
     - 희소성 : 행렬의 대부분 요소가 0으로 이뤄져 있는 경우
- 행렬 분해로 이해하는 Word2Vec



- Word2Vec의 학습 = Shifted PMI행렬을 U와 V로 분해하는 것

$$A_{ij}^{SGNS} = U_i \bullet V_j = PMI(i,j) - \log_k$$
$$A_{ij}^{SGNS} = SPMI\text{행렬}A\text{의}i\text{번째행},\ j\text{번째열의원소}$$
$$U_i = \text{타깃단어}(t)\text{의임베딩행렬}$$
$$V_j = \text{문맥단어}(c)\text{의임베딩행렬}$$
$$k = Skip-Gram\text{모델의네거티브샘플수}$$

  - [네거티브 샘플 수(k) = 1] 인 Skip-Gram 모델 = PMI 행렬 분해(log 1 = 0)
- Skip-Gram 모델
  - 말뭉치를 단어별로 슬라이딩하며 타깃 단어의 포지티브 샘플과 네거티브 샘플을 이진분류하는 과정에서 학습
    - 학습 완료시 ; U, V 행렬 얻을 수 있음
    - U, V의 내적 = SPMI 행렬

        - i번째 타깃 단어에 해당하는 U_i 벡터와 j번째 문맥단어에 해당하는 V_j 벡터의 내적
        - i,j의 PMI 값에서 log_k 를 빼준 값

    1. 말뭉치에서 i, j 두 단어가 자주 같이 등장 = SPMI 높음
        - 두 단어가 의미 상 관련 있을 가능성 높음
    2. 말뭉치에서 i, j 두 단어 동시 등장 적음 = SPMI 낮음
    - i, j 단어가 의미상 얼마나 관련 있는지
        - U_i, V_j의 내적값으로 표현됨
        - 관련성 높을수록 내적 값 높음
        - 코사인 유사도 높음
  - Skip-Gram과 그 변종(FastText)은 단어 임베딩에 말뭉치 전체적인 글로벌한 분포 정보 함축 가능
- 튜토리얼
  - soynlp

    > window = 3 ; 타깃 단어 앞,뒤 3개 단어를 문맥적으로 고려
    > dynamic_weight = True ; 타깃 단어에서 멀어질수록 카운트하는 동시 등장 점수를 조금씩 깎는다는 의미
    > dynamic_weight = Falst ; 윈도우 내에 포함된 문맥단어들의 동시 등장 점수는 타깃 단어와의 거리와 관계없이 모두 1로 일정하게 계산하겠다는 의미

    - 결과) 단어-문맥 행렬의 차원수 : (어휘 수)x(어휘 수) ; 정방행렬
      - 말뭉치 단어 수는 보통 10만개를 넘기에 매우 큰 행렬에 해당
    - sklearn이 제공하는 TruncatedSVD활용 시 : (어휘 수)x(100) 크기의 단어 임베딩 행렬
  - soynlp

    > pmi함수 : min_pmi = 0 설정 시 ; ppmi와 동일 pmi_matrix 의 차원수 : (어휘 수)x(어휘 수) ; 정방행렬

In [ ]:

```
# 단어-문맥 행렬을 활용한 LSA

from sklearn.decomposition import TruncatedSVD
from soynlp.vectorizer import sent_to_word_contexts_matrix

corpus_fname = "/notebooks/embedding/data/tokenized/for-lsa-mecab.txt"

corpus = [sent.replace('\n','').strip() for sent in open(corpus_fname,'r').readl
ines()]
input_matrix, idx2vocab = sent_to_word_contexts_matrix(corpus,
                                                       windows = 3,
                                                       min_tf = 10,
                                                       dynamic_weight = True,
                                                       verbose = True)
cooc_svd = TruncatedSVD(n_components=100) # 어휘 수 X 100 차원으로 축소
cooc_vecs = cooc_svd.fit_transform(input_matrix)
```

In [ ]:

```
# 코사인 유사도 상위 단어 목록 체크
# 단어-문맥 행렬 + LSA

from models.word_eval import WordEmbeddingEvaluator
model = WordEmbeddingEvaluator("data/word-embeddings/lsa/lsa-cooc.vecs",
                               method = "lsa",
                               dim = 100,
                               tokenizer_name = "mecab")
model.most_similar("희망", topn = 5)
```

In [ ]:

```
# PPMI 행렬을 활용한 LSA

from soynlp.word import pmi
ppmi_matrix, _, _ = pmi(input_matrix, min_pmi = 0)
ppmi_svd = TruncatedSVD(n_components = 100)
ppmi_vecs = ppmi_svd.fit_transform(input_matrix)
```

In [ ]:

```
# 코사인 유사도 상위 단어 목록 체크
# PPMI + LSA

from models.word_eval import WordEmbeddingEvaluator
model = WordEmbeddingEvaluator("data/word-embeddings/lsa/lsa-pmi.vecs",
                               method = "lsa",
                               dim = 100,
                               tokenizer_name = "mecab")
model.most_similar("희망", topn = 5)
```