Chapter 1

SVD for word representation

1.1 Singular Value Decomposition

Any given rectangular matrix A of dimensions $m \times n$ can be written as $A = U\Sigma V^T$
where U is the matrix containing eigenvectors of AA^T , V is a matrix containing eigenvectors of A^TA and Σ is a diagonal matrix containing square roots of eigen values of AA^T or A^TA

1.2 Truncated SVD Process

To find a rank-r approximation of a matrix $A_{m\times n}$ using Truncated SVD. Here are the steps to be followed.

- Calculate A^TA

```
ATA=np.dot(A,A.T)
```

- Calculate Eigenvalues λ_R of A^TA
- Calculate Eigenvectors V of A^TA using λ_R

```
eigenvalues,V=np.linalg.eig(ATA)
```

- Sort these eigenvalues λ_R and then sort V according the sorted eigenvalues.

```
idx = eigenvalues.argsort()[::-1]
eigenvalues = eigenvalues[idx]
V = V[:, idx]
```

- Calculate Singular value σ_i for every eigenvalue λ_{R_i} as $\sigma_i = \sqrt{\lambda_{R_i}}$. If an $\lambda_{R_i} \leq 0$ then $\sigma_i = 0$
- Calculate Σ diagonal matrix using these singular values and all other entries as 0.

```
eigenvalues[eigenvalues < 0] = 0
Sigma = np.sqrt(np.diag(eigenvalues))
```

- Using each of these positive non-zero singularvalue σ_i and corresponding v_i eigenvector, calculate i^{th} column of U using

$$A = U\Sigma V^T$$

$$AV = U\Sigma$$

$$U = AV\Sigma^{-1}$$

$$\begin{bmatrix} \uparrow & \uparrow & & \uparrow \\ u_1 & u_2 & \dots & u_k \\ \downarrow & \downarrow & & \downarrow \end{bmatrix}_{m\times k} = A \begin{bmatrix} \uparrow & \uparrow & & \uparrow \\ v_1 & v_2 & \dots & v_k \\ \downarrow & \downarrow & & \downarrow \end{bmatrix}_{n\times k} \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \dots & 0 \\ 0 & \frac{1}{\sigma_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{\sigma_k} \end{bmatrix}_{k\times k}$$

$$u_i = \frac{1}{\sigma_i} A v_i$$

- Find each u_i for each non-zero positive σ_i

```
U = np.zeros((A.shape[0], len(eigenvalues)))
for i in range(len(eigenvalues)):
       if eigenvalues[i] > 0:
           U[:, i] = (1 / np.sqrt(eigenvalues[i])) * np.dot(A, V[:, i])
```

- Assemble all these u_i as columns of U
- Hence we got U, Σ and V
- Now we can find SVD of $A = U\Sigma V^T$

```
def SVD(A):
  # Step 1: Compute A^TA
  ATA = np.dot(A.T, A)

  # Step 2: Calculate eigenvalues and eigenvectors of A^TA
  eigenvalues, V = np.linalg.eig(ATA)

  # Sort eigenvalues in descending order
  idx = eigenvalues.argsort()[::-1]
  eigenvalues = eigenvalues[idx]
  V = V[:, idx]
  eigenvalues[eigenvalues < 0] = 0

  # Construct Sigma matrix
  Sigma = np.sqrt(np.diag(eigenvalues))

  # Step 3: Calculate U matrix
  U = np.zeros((A.shape[0], len(eigenvalues)))
  for i in range(len(eigenvalues)):
      if eigenvalues[i] > 0:
          U[:, i] = (1 / np.sqrt(eigenvalues[i])) * np.dot(A, V[:, i])

  return U, Sigma, V
```

- If we need a rank-r approximation of A
- We can Use only first r columns of U and V and then first r columns and first r rows of Σ

```
def rank_k_approximation(A, k):
 # Compute SVD
 U, Sigma, V = SVD(A)
 # Rank-k approximation
 approx_U = U[:, :k]
 approx_Sigma = Sigma[:k, :k]
 approx_Vt = V.T[:k, :]

 return approx_U,approx_Sigma,approx_Vt
```

1.3 Word representation using SVD

- Get a text corpus.

```
corpus = """Dogs are wonderful companions .
 They enjoy playing fetch and running in the park.
 A well-trained dog can learn many tricks.
 Cats are independent creatures, often preferring solitude.
 Their agility and grace are admired by many.
 Some people love both dogs and cats equally, while others have a strong preference for one over the othe
 Dogs require regular exercise, whereas cats are more low-maintenance.
 Dog owners often take their pets for walks, while cats enjoy lounging indoors.
 The debate between dog lovers and cat enthusiasts is never-ending, each having valid reasons for their p
 Nevertheless, both dogs and cats bring joy and comfort to countless households.
"""
```

- Pre-process the corpus i.e., Punctuations removal, Stop words removal, Lemmatization etc.

```
def corpus_prepare(corpus):
 corpus=corpus.lower()
 corpus=corpus_remove_punc_symb(corpus)
 corpus=corpus_remove_stop_words(corpus)
 corpus=corpus_lemmatize(corpus)
 return corpus

corpus=corpus_prepare(corpus)
```

- Create Vocabulary (List of unique words) from the processed corpus.

```
def vocabulary_prepare(corpus):
 vocabulary=list(set(corpus_tokenise(corpus)))
 return vocabulary
vocabulary=vocabulary_prepare(corpus)
```

- Create Co-occurrence matrix using count of number of times a context word comes in a w window-size of a word.

```
def co_matrix_prepare(corpus,vocabulary,window_size):
 matrix={}
 context={}
 for i in range(0,len(vocabulary)):
  context[vocabulary[i]]=0

 for i in range(0,len(vocabulary)):
  matrix[vocabulary[i]]=context

 matrix = pd.DataFrame.from_dict(matrix)
 corpus_list=corpus_tokenise(corpus)

 for i in range(0,len(corpus_list)):
  windowwords=[]
  if i+window_size<len(corpus_list):
  for j in range(0,window_size):
   windowwords.append(corpus_list[i+j])
   matrix[corpus_list[i]][corpus_list[i+j]]+=1
  else:
  break

 for i in matrix.keys():
  matrix[i][i]=0

 return matrix

co_matrix=co_matrix_prepare(corpus,vocabulary,3)
```

- Update Co-occurrence matrix using Positive Pointwise mutual Information as a metric between 2 words as a context and word.

  $$PPMI(word(w), context(c)) = max(log\frac{count(w,c)*N}{count(w)*count(c)}, 0)$$

```
 def countOccurrences(str, word):
  wordslist = list(str.split())
  return wordslist.count(word)

 def PPMI(count_wc,count_c,count_w,N,eps=1e-6):
  if count_wc*N!=0 and count_c*count_w!=0:
   return max(eps,math.log((count_wc*N)/(count_c*count_w)))
  else:
   return 0

 def PPMI_co_matrix(matrix,corpus):
  for word in matrix.keys():
   for context in matrix[word].keys():
    matrix[word][context] = PPMI(matrix[word][context],
    countOccurrences(corpus,context),countOccurrences(corpus,word),len(matrix.keys()))
  return matrix


 co_matrix=PPMI_co_matrix(co_matrix,corpus)
```

- We can use Columns of this matrix as the word embedding but its of large dimensions
- If X is the co-occurrence matrix then let $\hat{X} = U\Sigma V^T$ be the low rank approximation of X matrix.
- $\hat{X}\hat{X}^T$ is the matrix containing Cosine similarity between each word in the low rank co-occurrence matrix.
- We can easily show that $\hat{X}\hat{X}^T = U\Sigma(U\Sigma)^T$

$$\begin{aligned}
\hat{X}\hat{X}^T &= U\Sigma V^T(U\Sigma V^T)^T \\
&= U\Sigma V^T V\Sigma^T U^T \\
&= U\Sigma\Sigma^T U^T \\
&= U\Sigma(U\Sigma)^T
\end{aligned} \tag{1.1}$$

And let $W_{word} = U\Sigma$

- This means $\hat{X}\hat{X}^T$ captures the same cosine similarity as $W_{word}(W_{word})^T$
- But $\hat{X} \in R^{m\times n}$ and $W_{word} = U\Sigma \in R^{m\times k}$ and $k < n$
- This means W_{word} can hold word embeddings with less dimensions and $W_{context} = V_{n\times k}$

```
def SVD(A):
 # Step 1: Compute A^TA
 ATA = np.dot(A.T, A)

 # Step 2: Calculate eigenvalues and eigenvectors of A^TA
 eigenvalues, V = np.linalg.eig(ATA)

 # Sort eigenvalues in descending order
 idx = eigenvalues.argsort()[::-1]
 eigenvalues = eigenvalues[idx]
 V = V[:, idx]


 # Construct Sigma matrix
 Sigma = np.sqrt(np.diag(eigenvalues))

 # Step 3: Calculate U matrix
 U = np.zeros((A.shape[0], len(eigenvalues)))
 for i in range(len(eigenvalues)):
  if eigenvalues[i] > 0:
   U[:, i] = (1 / np.sqrt(eigenvalues[i])) * np.dot(A, V[:, i])

 return U, Sigma, V

def rank_k_approximation(A, k):
 # Compute SVD
 U, Sigma, V = SVD(A)

 # Rank-k approximation
 approx_U = U[:, :k]
 approx_Sigma = Sigma[:k, :k]
 approx_Vt = V.T[:k, :]

 return approx_U,approx_Sigma,approx_Vt

def co_matrix_rank_k_SVD_word_context_embeddings(co_matrix,k):
 co_matrix_np = co_matrix.to_numpy()
 approx_U,approx_Sigma,approx_Vt = rank_k_approximation(co_matrix_np, k)
 word_matrix=approx_U @ approx_Sigma
 word_matrix = pd.DataFrame(word_matrix, index=co_matrix.index)
 context_matrix=pd.DataFrame(approx_Vt.T,index=co_matrix.index)

 return word_matrix,context_matrix
```

```
word_embeddings,context_embeddings=co_matrix_rank_k_SVD_word_context_embeddings(co_matrix,20)
```