Introduction to Natural Language Processing

Assignment 3

WORD VECTORIZATION

Hardik Sharma

Introduction

The purpose of this assignment is to find the differences between the `word2Vec` and the `SVD` methods to generate dense word representations from a given corpus. Finally, from the obtained embeddings, I train a classifier on the Downstream task of sentence label prediction using a bidirectional LSTM, which helps in providing a clear analysis of the various methods and their corresponding pros and cons.

SVD

The first method consists of finding the top-k vectors from the **Singular Vector Decomposition** of the *co-occurence* matrix. To create the *co-occurence* matrix, we take the frequency of other words in the context (defined by the `contextSize`) and compute the SVD to get the top dominant eigenvectors, computing the word representations from the same.

Architecture

I have computed the `SVD` using ***scipy.sparse.linalg.svds***, with the sparse represntations from the ***scipy.sparse.lil_matrix***. The entire code for the same in present in the `src/parser.py`, which also contains the code for parsing the entire dataset and storing all the different embeddings and data for the same.

Code

```
    def __getSparseCoocurenceMatrix__(self):
        """
            Returns a sparse co-occurence matrix
            as a list of lists.
        """
        if not hasattr(self, "mapping") or self.mapping is None:
            self.__constructBidict__()

        print("Creating sparse co-occurence matrix...")
        matrixDimension = len(self.vocab)
        sparseMatrix = LilMatrix((matrixDimension, matrixDimension), dtype = "float64")

        with alive_bar(len(self.tokenizedData), force_tty = True) as bar:
            for sentence in self.tokenizedData:
                for index, token in enumerate(sentence):
                    windowStart = max(0, index - SVDConfig.contextWindow)
                    for windowToken in sentence[windowStart : index]:
                        sparseMatrix[self.mapping[token], self.mapping[windowToken]] += 1
                        sparseMatrix[self.mapping[windowToken], self.mapping[token]] += 1
                bar()
        return sparseMatrix
```

In the end, I perform SVD using ***scipy.sparse.linalg.svds***. The reason for performing sparse SVD is to speed up the computation.Whereas the reason for using a sparse matrix is to improve the efficiency of using the space, for common words such as *the*, *a* will have a lot of words in their context, but for most of the words, there context would not be populated heavily. Therefore, using a sparse representation pays off.

```
    def performSVD(self, sparse = True):
        """
            Performs SVD on the co-occurence matrix
```

```
        @param sparse: Set to True if you want a sparse matrix representation.
    """

    try :
        if sparse:
            if ( not hasattr(self, 'SparseCoOcMatrix') ):
                self.getCoocurenceMatrix(sparse = True)

            print("Computing SVD on sparse co-occurence matrix...")
            U,S,Vt = SparseSVD(A = self.SparseCoOcMatrix, k = min(SVDConfig.EmbeddingSize,
                len(self.vocab) - 1) )

            self.U = U
            self.S = S
            self.Vt = Vt
        else:
            if ( not hasattr(self, 'StochasticCoOcMatrix') ):
                self.getCoocurenceMatrix(sparse = False)

            U,S,Vt = DenseSVD(self.StochasticCoOcMatrix)

            self.U = U
            self.S = S
            self.Vt = Vt

    except:
        import traceback
        traceback.print_exc()
```

Word2Vec

Here, I have implemented the `skip-gram` technique for doing ***Word2Vec***, to reduce the computational cost, I have also implemented the *negative-sampling* approach. The `contextSize` is set thoughh the `src/Config.py` file.

Architecture

The architecture of the word2vec is straightforward, I use two embedding layers, one for the `targetEmbedding` for the word and one for the `contextEmbedding` of the model. Finally, I add these pair wise to get the final embedding for the word.

The positive samples are pre-computed for the model whereas the negative samples are constructed on the fly during the training. This is done through a somewhat unusual use of the custom Dataset class.

The number of negative samples per positive sample are also set from the `src/Config.py` file. It contains other controllable parameters such as learning rate and embedding size.

Code

Datapoints preparation.

```
def getWord2VecDataPoints(self):
    self.word2VecDataPoints = []
    print(f"Preparing data for Word2Vec...")
    with alive_bar(len(self.tokenizedData), force_tty=True) as bar:
```

```
            for sentence in self.tokenizedData:
                for index, token in enumerate(sentence):
                    window = sentence[max(0,index - Word2VecConfig.contextWindow) : index] +
                        sentence[index + 1 : index + 1 + Word2VecConfig.contextWindow]
                    for word in window:
                        if word in self.vocab:
                            self.word2VecDataPoints.append( (self.mapping[token],
                                self.mapping[word], True) )
                bar()

    def getWordEmbedding(self, word : str) -> list:
```

The Custom Dataset class :

```
class CustomDataset(TorchDataset):
    def __init__(self, word2VecDataPoints : list, vocabLength : int) -> None:
        self.word2VecDataPoints = word2VecDataPoints
        self.vocabLength = vocabLength

    def __len__(self):
        return len(self.word2VecDataPoints)*(Word2VecConfig.negativeSamples + 1)

    def __getitem__(self, index):
        actualIndex = index // (Word2VecConfig.negativeSamples + 1)

        if index % (Word2VecConfig.negativeSamples + 1) == 0 :
            dataPoint = self.word2VecDataPoints[actualIndex]
            return dataPoint
        else :
            dataPoint = self.word2VecDataPoints[actualIndex]
            randomWord = random.randint(0, self.vocabLength - 1)
            return ( dataPoint[0], randomWord, False )
```

This helps in stochastically genrating the negative samples during training.

Forward pass for the model :

```
    def forward(self, word, context):
        out1 = self.contextEmbedding(context)
        out2 = self.wordEmbedding(word)
        out = torch.bmm(out1.unsqueeze(1), out2.unsqueeze(2)).squeeze()
        return Sigmoid(out)
```

The embeddings for both the models are stored in the `./corpus/` directory.

Results

Below is the graph for the model accuracy on the test data, plotted vs the time taken for training the model. From the raw data itself.

Model Accuracies :

Model	Context Size	Accuracy
SVD	1	82.88%
SVD	2	85.24%
SVD	3	88.65%
Word2Vec	1	88.66%
Word2Vec	2	90.22%
Word2Vec	3	90.39%

Confusion Matrices

SVD

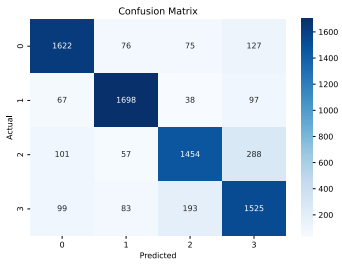

Figure 1: CS = 1

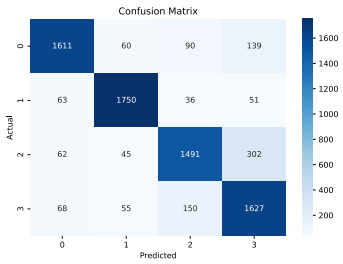

Figure 2: CS = 2

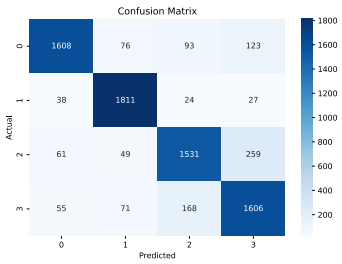

Figure 3: CS = 3

Word2Vec

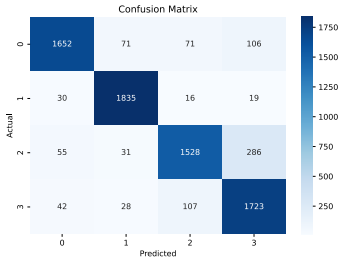

Figure 4: CS = 1

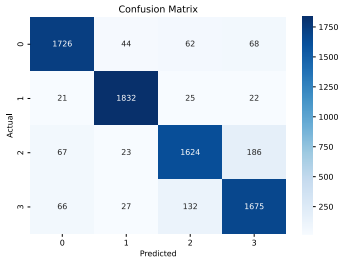

Figure 5: CS = 2

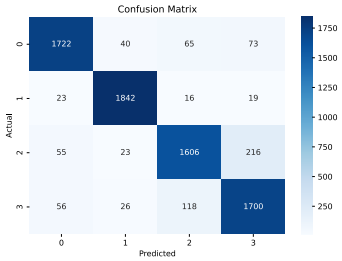

Figure 6: CS = 3

Conclusion

Overall, we can see that `word2Vec` outperforms `SVD` on the downstream task. It is also interesting to see that `word2Vec` with context size 1, is similar to performance with `SVD`, with context size 3.

Despite the clear domination in accuracy, it is not straightforward to choose between the two. For example, `word2Vec` has a much higher accuracy, but also takes significantly more time to train. As seen from the following visualization :

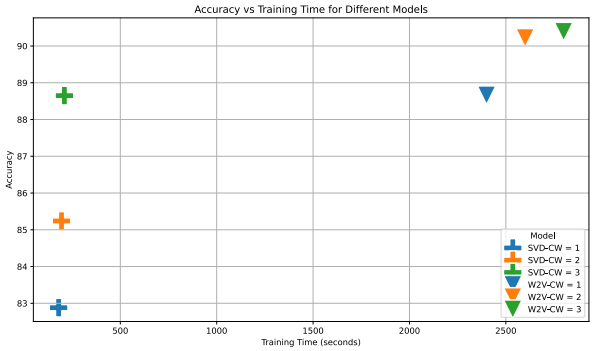

Figure 7: Accuracy Comparison

The graph shows the time taken to prepare the two embeddings using same hardware, and it shows that although the `word2Vec` model is much more accurate than `SVD`, it is almost 10 times as slow. Whereas SVD has an inherent scaling issue that it scales quadratically with the vocab size, even with sparse matrix performance becomes an issue, so both these things need to be taken into account before procedding with the choice of embeddings.