Sentiment Analysis

– Sentiment Analysis on Movie Reviews from Rotten Tomato

Abstract

Sentiment analysis is the process of analyzing digital text to determine if the emotional tone of the message is positive, negative, or neutral. The assignment is to run sentiment analysis on movie reviews from Rotten Tomato data.

To extract features from textual data, three different models are implemented: Bag-of-word, Ngrams, and Word2Vec with pretrained models. Among which the pre-trained model has provided the most accurate classification with the shortest time cost. The advantages of using pre-trained models include the ability to leverage the knowledge and experience of others, the ability to save time and resources, and the ability to improve model performance. The accuracy score on the test set reaches 0.7597.

Word-based n-gram has the second best performance, whilst the byte-based n-gram has the worst performance. Byte-based n-grams usually require less memory to hold the vocabularies, and hold much less information. The limited vocabulary size leads to easy-overfitting and low generalization easily. Byte-based n-grams are usually utilized in such tasks as language identification, writer identification (i.e. fingerprinting), anomaly detection. This is why it does not have good performances here, since it is not suitable for the classification task for sentiment mining.

BOW is weaker than word-based n-grams, but better than the character-based ngrams. During the NLP application, n-gram is used to consider words in their real order so we can get an idea about the context of the particular word; BOW is used to build vocabulary for your text dataset. An assumption is that the word order and word-probability in regards to another word is important to our task, especially when the data itself has objectives such as the nature of reviews or comments. That is, for example, “he likes chocolate but hates vanilla” and “he likes vanilla but hates chocolate” have completely different meanings, but this type of difference can not be interpolated by BOW models.

Our current weight calculation when vectorizing texts is purely based on the occurrence count of a word. If time applies, we could try comparing TF-IDF for feature extraction, since TF-IDF not only focuses on the frequency of words present in the corpus but also provides the importance of the words. Also, we could test other pretrained models.

1 Introduction

Sentiment of a textual content is usually classified as positive, negative, or neutral. However, data used in the experiment only has two labels: positive and negative. The learning model is simplified to a basic binary classification problem.

A code framework has been provided. We re-wrote the framework for better readability and transferability. The classification is derived from a binary classification Hinge Loss, and the optimizer is a Stochastic gradient descent with the option to enable or disable momentum when updating model parameters. The

code implementation invokes Numpy for matrix calculation, and Gensim module for loading pretrained models.

2 Data Preparation

Text preprocessing is a method to clean the text data and make it ready to feed data to the model. Text data contains noise in various forms like emotions, punctuation, text in a different case. When we talk about Human Language then, there are different ways to say the same thing, And this is only the main problem we have to deal with because machines will not understand words, they need numbers so we need to convert text to numbers in an efficient manner[1]. Usual text preprocessing techniques are: removing punctuations/stopwords, converting all letters to lowercase etc..

However, such cleanup pipelines may vary for different tasks. Take the uppercase letters for example. Imagine that we have a review from someone really angry. In order to represent their anger, the person decided to write everything in capital letters. In this context, this style choice adds information to the analysis, since it represents anger and frustration[2]. Same logic could be applied to punctuations. Therefore, we will not remove any words from our textual data. In addition, we will only convert the first letter of the sentence start word to lowercase.

3 Feature Extraction.

A problem with modeling text is that it is messy, and techniques like machine learning algorithms prefer well defined fixed-length inputs and outputs. Machine learning algorithms cannot work with raw text directly; the text must be converted into numbers. Specifically, vectors of numbers[3].

As stated in the introduction, we will compare and contrast the performances of three different ways of converting texts to vectors of numbers: Bag-of-word, Ngrams, and Word2Vec. Below sections will present the code implementation for these three models

3.1 Bag-of-Words

The bag-of-words model is a model of text which uses a representation of text that is based on an unordered collection (or "bag") of words. It disregards word order (and thus any non-trivial notion of grammar[clarification needed]) but captures multiplicity[4].

```python
def build_vocab(data, min_occurane=500):
    """
    Build vocab from train data

    min_occurane: remove low frequency word
    """
    vocab_counter = Counter()
    for (line, label) in data:
        tokens = line.split(" ")
        vocab_counter.update(tokens)

    vocab = [word for word, count in vocab_counter.items() if count>min_occurane]

    # add <UNK> as a pad token for word not in the vocab
    vocab.insert(0, '<UNK>')

    return vocab
```

```python
def BOW_feature_extract(x, vocab):
    # initialize all zero array to hold the weight of tokens in the vocab
    vectorize = np.zeros((len(vocab),), dtype=float)

    # count word frequency and update the vectorization
    words = x.split(" ")
    for word in words:
        if word in vocab:
            vectorize[vocab.index(word)] += 1
        else:
            # <UNK> is always the first word in the vocabulary
            vectorize[0] += 1
    # normalization
    sum = vectorize.sum(axis=0)
    if sum!=.0:
        vectorize = vectorize / sum
    return vectorize
```

To implement the BOW model from scratch, we first need to build a vocabulary from all the training samples. We provide a feature to remove low-frequency words from the vocabulary.

A common process in NLP is to remove high and low frequency words before going into training. Frequent words are almost always devoid of meaning. And rare words are removed because the association between them and other words is dominated by noise[5].

A “<UNK>” padding is added to the built vocabulary. It is used for feature extraction, that is, to calculate the weight of a word in the vocabulary based on its count or frequency in the document data. A word as such is known as “unregistered”. This occurs when we remove words from the vocabulary due to its frequency, or simply because the word never appears in the train set. This is also a method to maintain the generalization ability of a model.

To calculate the weights of words in the vocabulary, we here simply used the count of appearance of the word within one sample. A better approach is TF-IDF, since it not only focuses on the frequency of words present in the corpus but also provides the importance of the words. However, due to the time restriction, we did not implement or compare TF-IDF in this assignment.

3.2 N-gram

An n-gram is a sequence of n adjacent symbols in particular order. The symbols may be ‘n’ adjacent letters (including punctuation marks and blanks), syllables, or rarely whole words found in a language dataset[6]. We enable word-based n-gram and byte-based n-gram in the method definitions.

```python
def build_ngrams(data, ngram=[2], min_occurane=500, byte_base=False):
    """
    n-gram: list(int), supports multiple n-gram mixing
    byte_base: character-based n-gram or word-based n-gram
    """
    ngram_counter = Counter()

    # word-based
    if not byte_base:
        for (line, label) in data:
            words = line.split(" ")
            for ngram_v in ngram:
                if ngram_v>1:
                    words.insert(0, '<PAD>')
                    words.insert(-1, '<PAD>')
                temp = zip(*[words[i:] for i in range(ngram_v)])
                ngram_tokens = [' '.join(ngram) for ngram in temp]
                ngram_counter.update(ngram_tokens)
    # character-based
    else:
        for (line, label) in data:
            for ngram_v in ngram:
                temp = zip(*[line[i:] for i in range(ngram_v)])
                ngram_tokens = [' '.join(ngram) for ngram in temp]
                ngram_tokens = ['<PAD>'+line[0], line[-1]+'<PAD>'] + ngram_tokens
                ngram_counter.update(ngram_tokens)


    ngrams = [word for word, count in ngram_counter.items() if count>min_occurane]

    # add <UNK>, <PAD> as a pad token for word not in the vocab or padding to the start/end word in a line
    ngrams.insert(0, '<UNK>')

    return ngrams
```

```python
def Ngram_feature_extract(x, ngrams, ngram=[2], byte_base=False):
    # initialize all zero array to hold the weight of tokens in the vocab
    vectorize = np.zeros((len(ngrams),), dtype=float)

    # count word frequency and update the vectorization

    # if is word-based
    if not byte_base:
        words = x.split(" ")
        for ngram_v in ngram:
            if ngram_v>1:
                words.insert(0, '<PAD>')
                words.insert(-1, '<PAD>')
            temp = zip(*[words[i:] for i in range(ngram_v)])
            tokens = [' '.join(ngram) for ngram in temp]

            for token in tokens:
                if token in ngrams:
                    vectorize[ngrams.index(token)] += 1
                else:
                    # <UNK> is always the first word in the vocabulary
                    vectorize[0] += 1

    # if is character based
    else:
        for ngram_v in ngram:
            temp = zip(*[line[i:] for i in range(ngram_v)])
            tokens = [' '.join(ngram) for ngram in temp]
            tokens = ['<PAD>'+line[0], line[-1]+'<PAD>'] + tokens

            for token in tokens:
                if token in ngrams:
                    vectorize[ngrams.index(token)] += 1
                else:
                    # <UNK> is always the first word in the vocabulary
                    vectorize[0] += 1

    # normalization
    sum = vectorize.sum(axis=0)
    if sum!=.0:
        vectorize = vectorize / sum

    return vectorize
```

The vocabulary build and weight calculation are similar to the BOW model described in Section 3.1, except this time, the vocabulary does not hold per-word base tokens, but tokens that are formed by adjacent elements.

Another difference is that we added “<PAD>” tokens in the extraction process. For n-gram with n greater than 1, we use “<PAD>” to do left padding and right padding to a sentence. Padding ensures that each symbol of the actual string occurs at all positions of the ngram.

3.3 Word2Vec with Pretrained Models

Word2Vec, is a model only for learning vectors for individual words. As an algorithm, word2vec says nothing about what a vector for a multi-word text should be.

Many people choose to use the average of all a multi-word text's individual words as a simple vector for the text as a whole. It's quick & easy to calculate, but fairly limited in its power. Still, for some applications, especially broad topical-classifications that don't rely on any sort of grammatical/ordering understanding, such text-vectors work OK – especially as a starting baseline against which to compare additional techniques[7].

An optimization for the limitation stated above could be the doc2vec model. However, due to the limitation of time, we will not test this model. Below are the code snippets to use pretrained models.

```python
def pretrain_extract_features(x, model):
    """
    model : pretrained model
    Pretrained model list could be found at: https://radimrehurek.com/gensim/models/word2vec.html
    """

    # clean up texts (all lower case)
    word_list = x.lower().split(" ")
    multiword_average_vector = model.get_mean_vector(word_list)
    return multiword_average_vector
```

4 Learning Model and SGD Optimizer

In this section, we implement a learning model with Hinge Loss, and a SGD optimizer with momentum.

```python
class HingeLoss():
    def __init__(self, num_input, num_output, trainable=True):
        self.num_input = num_input
        self.num_output = num_output
        self.trainable = trainable

        self.XavierInit()
        self.delta_W = 0.0

    def XavierInit(self):
        """
        Initialize the weigths

        Args:
            num_input: size of each input sample(feature amount)
            num_output: size of each output sample(1 -> binary classification)
        """
        raw_std = (2 / (self.num_input + self.num_output))**0.5
        init_std = raw_std * (2**0.5)
        self.W = np.random.normal(0, init_std, (self.num_input, self.num_output))

    def forward(self, x, y):
        # x is one sample point
        prob = np.dot(x, self.W)
        loss = max(0, 1-y*prob)
        grad = self.gradient_computing(y, prob, x)

        # update weight gradient
        #self.grad_W = grad[:,0]
        if self.trainable:
            self.grad_W = grad.reshape(self.num_input, self.num_output)
        #print(self.grad_W.shape)
        return loss, prob

    def make_predict(self, prob):
        y_predict = 1 if prob>=0 else -1
        return y_predict

    def gradient_computing(self, y, prob, x):
        grad = np.zeros((self.num_input, self.num_output), dtype=float)
        if y*prob<=1:
            grad = -x*y
        return grad
```

```python
class SGD():
    def __init__(self, learning_rate, momentum=0.9, use_momentum=False):
        self.learning_rate = learning_rate
        self.momentum = momentum
        self.use_momentum = use_momentum

    def step(self, model):
        if model.trainable:
            if self.use_momentum:
                model.delta_W = self.momentum*model.delta_W - self.learning_rate*model.grad_W
                model.W = model.W + model.delta_W
            else:
                model.W = model.W - self.learning_rate*model.grad_W
```

The formula for derivation of Hinge Loss gradient descent(inspired by [9]) is:

$$l_{\text{hinge}} = \max(0, 1 - y\,\boldsymbol{x} \cdot \boldsymbol{w})$$

We will rewrite the loss function above as:
$f(g(w))$ where $f(z) = max(0, 1 - yz)$ and $g(w) = x * w$.

To get the derivation of $f(g(w))$ we use the chain rule:

$$\frac{\partial}{\partial w_i} f(g(w)) = \frac{\partial f}{\partial z}\frac{\partial g}{\partial w_i}$$

The first derived item is evaluated at $g(w) = x * w$ becoming -y when $x * w < 1$ and 0 when $x * w \geq 1$

Therefore, the weight gradient by Hinge Loss is:

$$\frac{\partial f(g(w))}{\partial w_i} = \begin{cases} -y\,x_i & \text{if } y\,\mathbf{x} \cdot \mathbf{w} < 1 \\ 0 & \text{if } y\,\mathbf{x} \cdot \mathbf{w} > 1 \end{cases}$$

5 Hyperparameter Tuning

We run hyperparameter tuning for each feature extraction method introduced in Section 3.

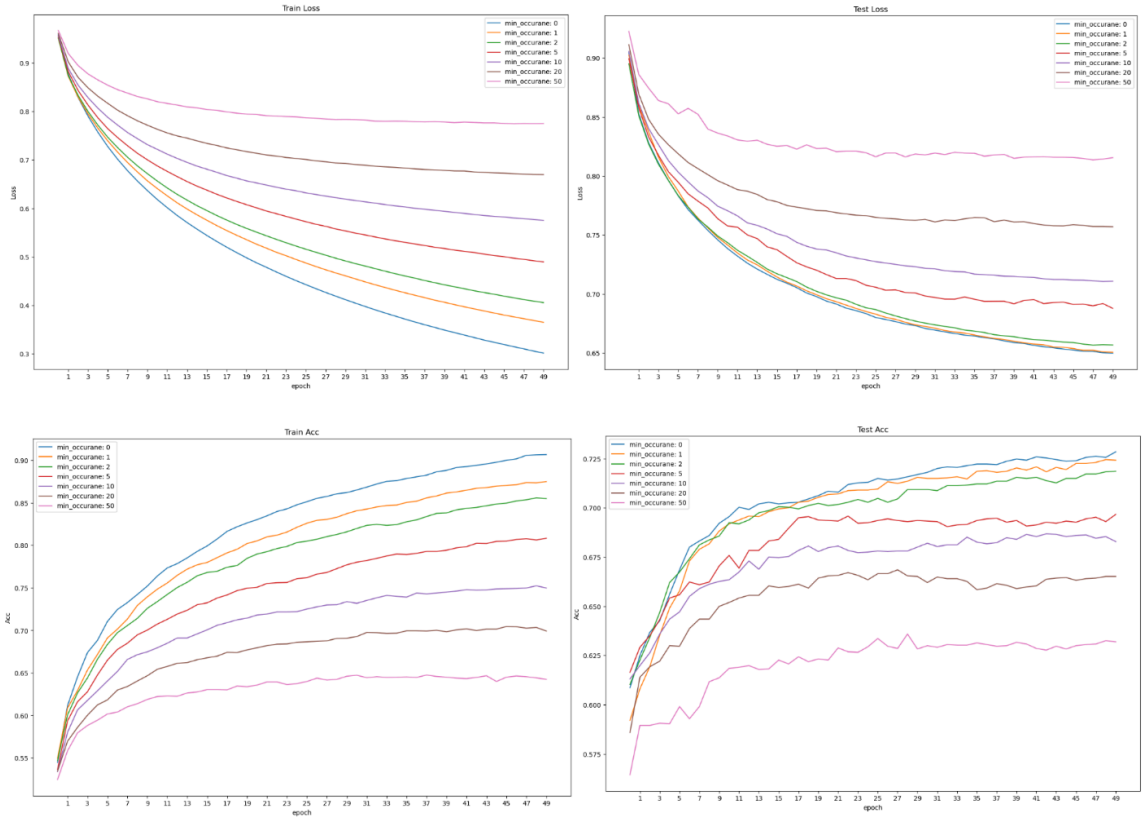
5.1 Bag-of-Words(BOW)

5.1.1 BOW: min_occurane

We eliminated tokens with a count that is smaller than [0, 1, 2, 5, 10, 20, 50].

The BOW here relies simply by the word count, a better weight representation is TF-IDF. We will test this vectorization if time applies. Vector dimensions corresponding to each min_occurane are: [11627, 4928, 3074, 1401, 693, 347, 146].

Starting point of tuning min_occurane value: SGD without momentum, learning rate: 0.5, Epoch amount: 50.



Observations:

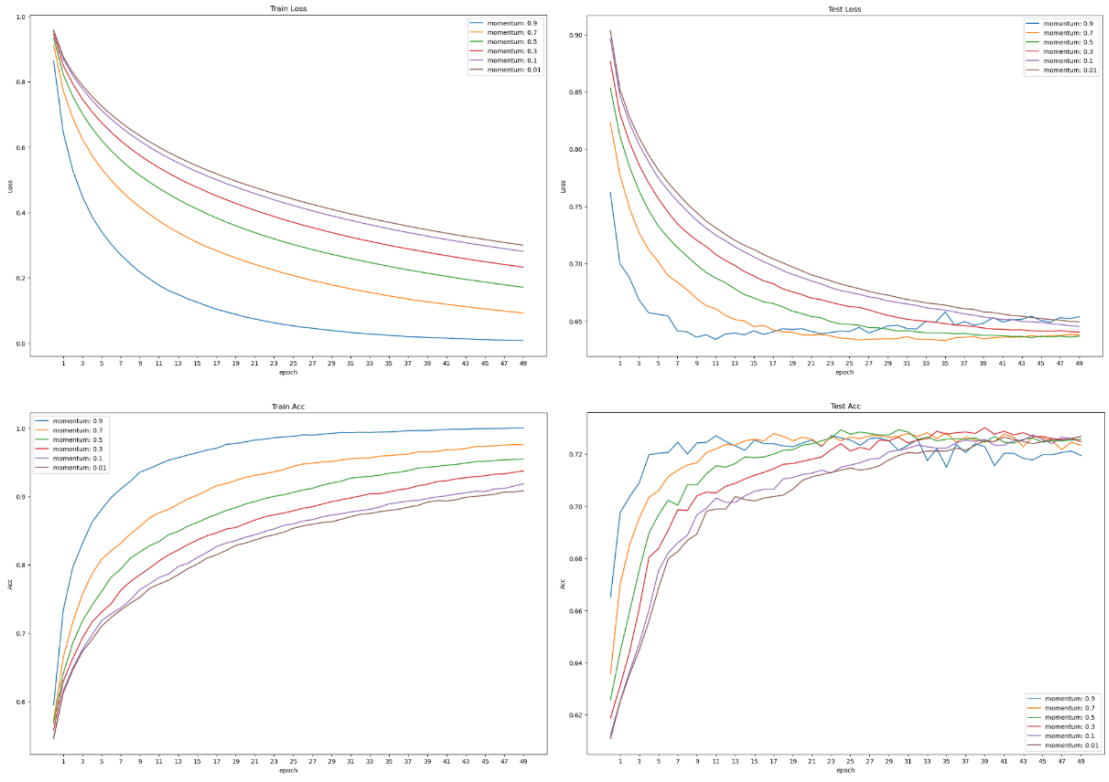
1. Learning rate 0.5 is too small for the model to learn, the loss decreases slowly, and is no near to convergence after 50 epochs;
2. Although removing rare words is likely to improve model performance and to fight against overfitting, in our experiments, the more words removed, the worse the model gets, with slower

convergence, higher loss and lower accuracy. It is possible that since our data is originally not of a large scope, that some of the rare words actually speak out decisive features for the classification, rather than introducing in outliers or making the feature vector too sparse. Also, the removal might also make the weights for the “UNK” padding in the test samples too large, which also negatively affects the model prediction with low generalization ability.

3. Same reason as above, although machine learning results can be weakened by the “curse of high dimensionality[10]”, since our original vocabulary is not of high dimension, the reduction of dimension(by removing rare words) does not improve the model behavior.
4. We will use the min_occurane=0 for later tuning, since it has the best prediction results on the test set.

5.1.2 BOW: SGD Momentum

Momentum is faster than stochastic gradient descent; the training will be faster than SGD. Local minima can be an escape and reach global minima due to the momentum involved. Larger momentum means more aggressive update in the gradient descent.

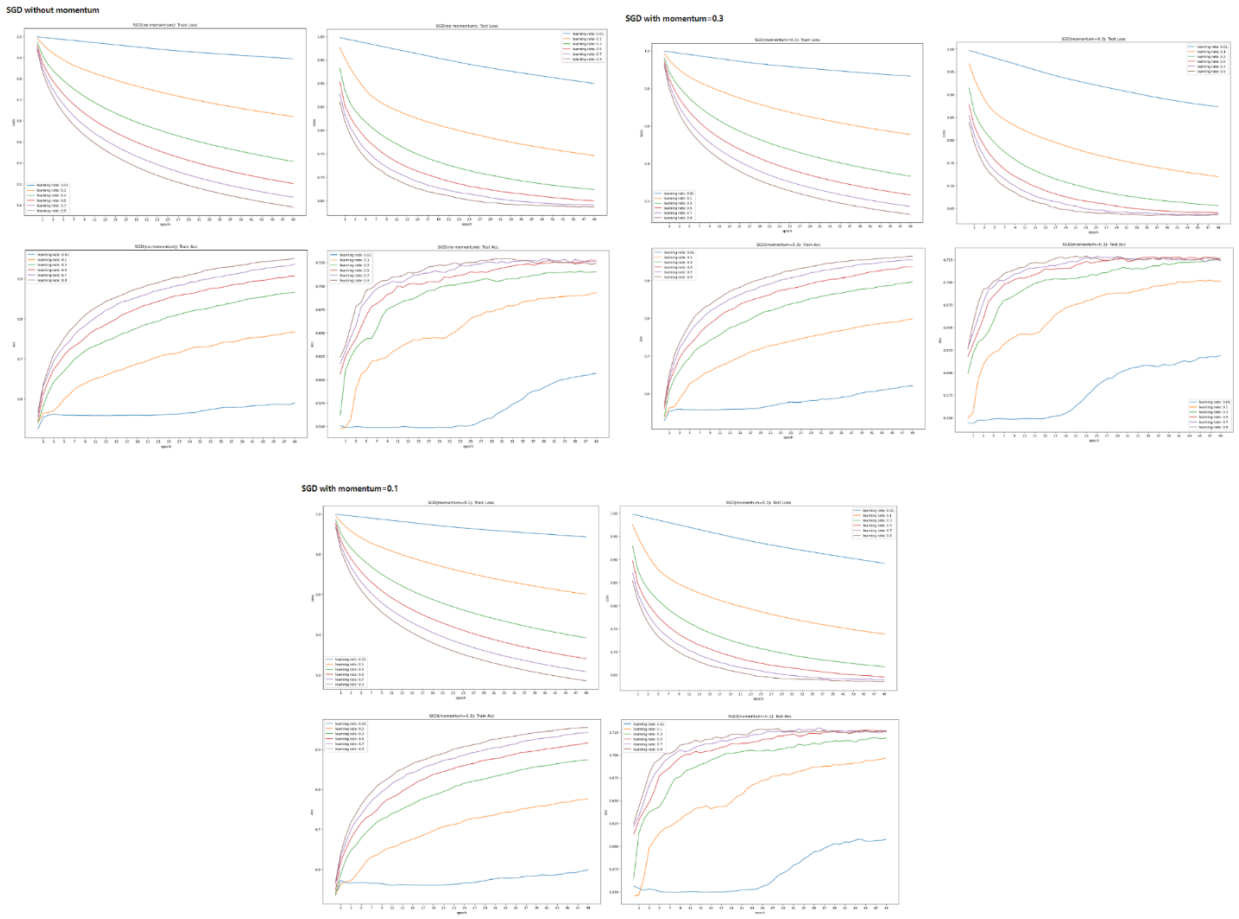


Observations:

1. When the momentum is set to 0.9, the model updates its weight parameters too aggressively, which leads to overfitting quickly. Its loss curve fluctuates a lot on the test set, this is because the model updates too aggressively and always jumps around the optimal point.
2. The highest accuracy on the test set is with momentum 0.3. The accuracy curve and loss curve of momentum 0.1 approach to momentum 0.3 at the end of the epochs, and are still converging. Therefore, we will use momentum 0.1 and 0.3 for comparing learning rate.

5.1.3 BOW: Learning Rate

Based on the learning diagrams, with momentum set to 0.1 and 0.3, learning rate as 0.5 is still at a low convergence speed. We will also test the learning rate with momentum disabled.



Observations:

1. For all models we tested, the best performances are all with learning rate 0.9.
2. Models finished convergence at around 30 epochs.

Model	Best Train Loss	Best Test Loss	Best Train Acc	Best Test Acc	Train Time (in sec)
SGD(no momentum)	0.18997	0.63665	0.9513224	0.729319	6.980279
SGD(momentum=0.3)	0.12929	0.63461	0.96370	0.72847	8.04234
SGD(momentum=0.1)	0.171239	0.63581	0.955261	0.72903	8.241274

With BOW model and word count to vectorize the textual data, learning rate 0.9 with momentum 0.1. The best test accuracy reaches 0.729(that is error 0.271).

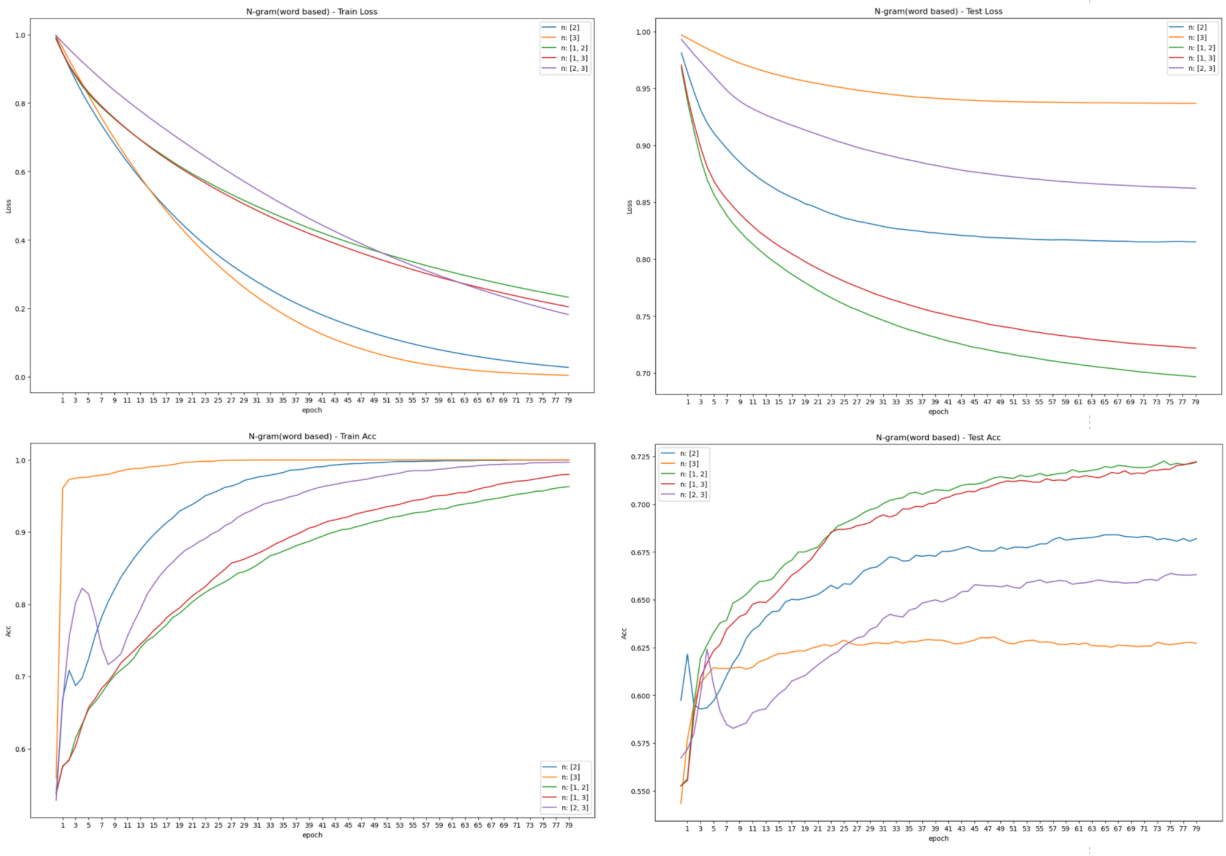
Due to the scope of our data set is not too large, keeping all the words extracted from the train samples in the vocabulary will not lead to weak generalization, or make the model less robust. But all in all, the generalization using BOW is not too ideal. The accuracy on the train set is above 0.95, whilst the best accuracy on the test set only hits around 0.73. The loss value also reflects the low generalization, with train loss below 0.2 and test loss above 0.6. This might be because: 1. Model tries to memorize too many rules from the train set, which does not apply to unknown data(test data); 2. The vocabulary is limited due to the scope of the train set, which makes the vector representations of test samples have large weights.

5.2 N-gram – Word Based

5.3.1 N-gram – Word Based: n

We will use learning rate 0.5 and max epoch 80 for a start.

The n values tested are: [2], [3], [1,2], [1,3], [2,3]. And the associated feature dimensions are [46613, 66460, 58239, 78086, 114148].



Observations:

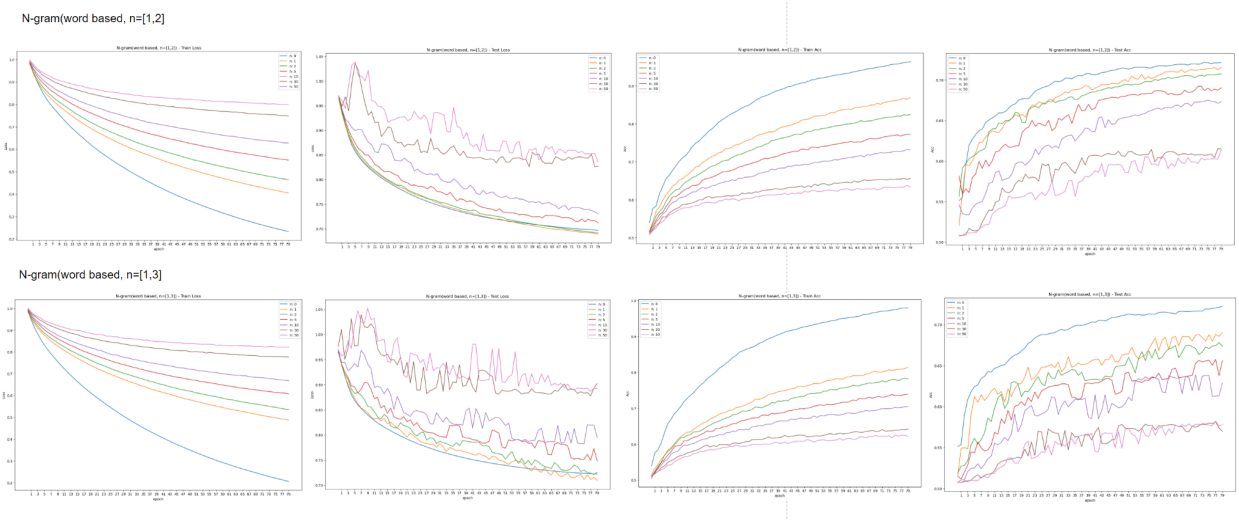
1. When the n for ngram is set to a single value, e.g. 2 or 3, the model has weak generalization ability. An assumption is that the single ngram(n not equal to 1) have comparatively small vocabulary, and that there exists many unseen adjacent pairs in the test samples. A language model can achieve low perplexity by choosing a small vocabulary and assigning the unknown word a high probability[11]. Once mixing 1-gram with other ngrams, the model can be improved significantly.
2. The above assumption can also be proved by the model behavior with 2-gram and 3-gram, it is much worse than the combination of 1-gram with any other n-grams.
3. [1,2]-gram and [1,3] gram have the best performances among tested configurations. [1,2]-gram is slightly better with higher test accuracy and lower test loss.
4. Another phenomenon is that all models' loss curves approach 0.0 and accuracies approach 1.0 at the end of the training. This might weaken the model generalization on unseen data. One possible reason is that the extracted feature vector is of too high of dimensionality.

5.3.2 N-gram – Word Based: min_occurane

We will choose n=[1,2] and [1,3] for testing the value for min_occurane.

The min_occurane counts for removing pairs from the vocabulary are: [0, 1, 2, 5, 10, 30, 50]. Associated dimension of vectors are:

1. n=[1,2]: [58239, 12552, 6567, 2501, 1130, 357, 193];
2. n=[1,3]: [78086, 8697, 4368, 1667, 768, 262, 150].



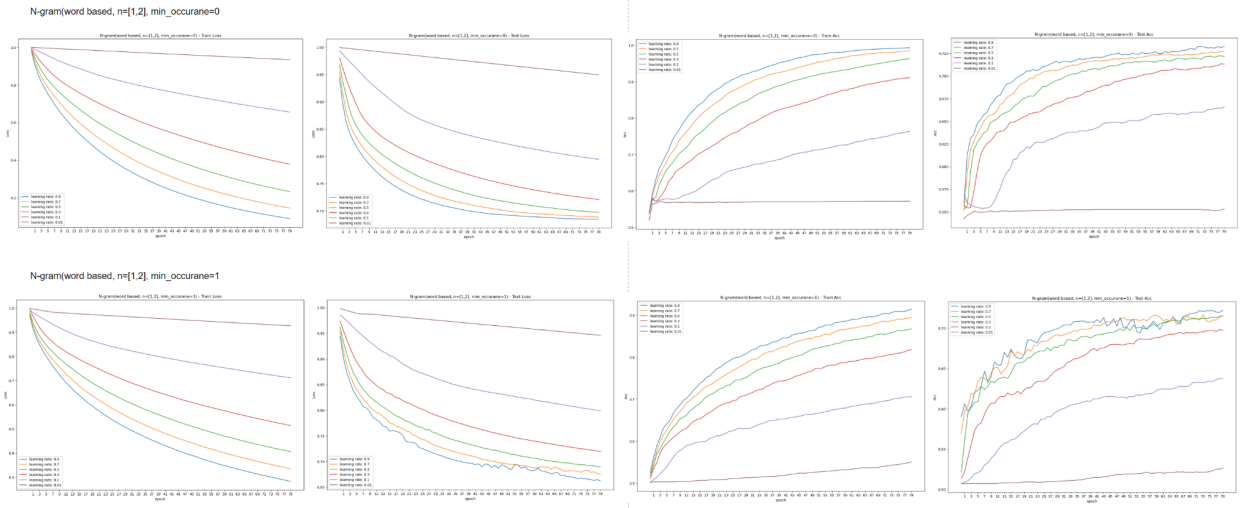
Observations:

1. Both models have the worst predictions when the minimum count to keep a token in vocabulary is set to 30 or 50. When too many tokens are removed from the vocabulary, the vocabulary becomes too small and has not enough perplexity to represent the textual contents and the classification rules.
2. On the contrary to the assumption we made in the above section, the training loss approaches 0 and training accuracy approaches 1 is not caused by the impact of high dimensionality. Since in

this section, the model has the best generalization ability when no tokens get removed from the vocabulary.

3. When using the combination of 1-gram and 2-gram, the result of minimum count 1 starts to approach minimum count 0 in the converging process. Therefore, we will test both configurations with other hyperparameters.
4. Curves on the test set fluctuate a lot with the combination of 1-gram and 3-gram. Reason for this is if the individual accuracy of a given n value is lower for a particular selected corpus size it has a slightly negative effect on combined feature selection accuracy. (larger the n larger can be the corpus size needed for improving a unit of accuracy)[12].

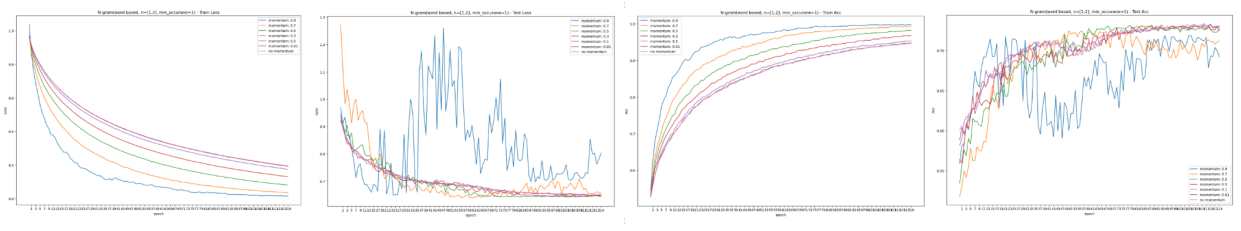
5.3.3 N-gram – Word Based: Learning Rate



Observations:

1. Models have their best performances when the learning rate is set to 0.9. In addition, none of the models with 0.9 learning rate has converged within 80 epochs. Therefore, we will introduce momentum to update the model more aggressively later for faster convergence.
2. As observed above, although the model behavior when removing tokens that only appear once is close to not removing any tokens, the minimum count 0 still has better prediction accuracy.

5.3.4 N-gram – Word Based: Momentum



Observations:

1. The Lowest quality of learning is when the momentum is set to 0.9. The parameter update is too aggressive, which leads to overfitting quickly. The loss and accuracy curves fluctuate a lot on the

test set, this is because the model updates too aggressively and always jumps around the optimal point.

2. Overfitting also happens for momentum 0.7 and 0.5. Always having a large momentum can make the model jump around the global minimum since it updates too aggressively and can never actually reach the optimal solution point. It is recommended to have low momentum at the beginning of learning and increase it later to help the network converge more effectively. Momentum is a hyperparameter that helps accelerate gradient descent in the relevant direction and dampens oscillations. At the beginning of training, having low momentum allows the network to explore the parameter space more thoroughly, which can help it escape shallow local minima and prevent overshooting the global minimum[13]. However, due to the time being not enough, we did not try Adam or other optimizers that decay the updates as the model converges.

	Model	Best Train Loss	Best Test Loss	Best Train Acc	Best Test Acc	Train Time (in sec)	Converge at epoch
N-gram n=(1,2) min_occurane = 1 lr = 0.9 max epoch = 120	momentum=0.3	0.13013994	0.6434025	0.968204	0.732695	44.939846	114
	momentum=0.5	0.080661	0.64254	0.98199	0.73128	42.27238	81

The above is the final model configurations we choose for word-based ngram. We could see that the model performance is better than the BOW. In n-grams, word order is important, whereas in BOW it is not important to maintain word order. For example, “he likes chocolate but hates vanilla” has a different meaning compared to “he likes vanilla but hates chocolate”. This type of difference cannot be interpolated by BOW models.

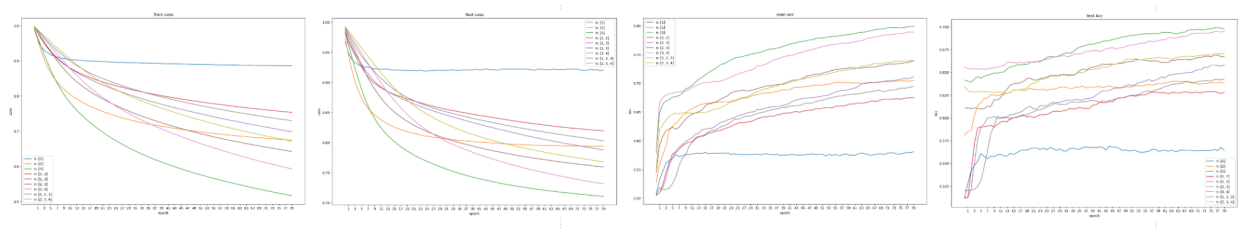
5.3 N-gram – Byte Based

Analysis for this section is similar to the Section 5.2 for Word-based ngrams. So we will mainly put the learning curve diagrams and omit the observations.

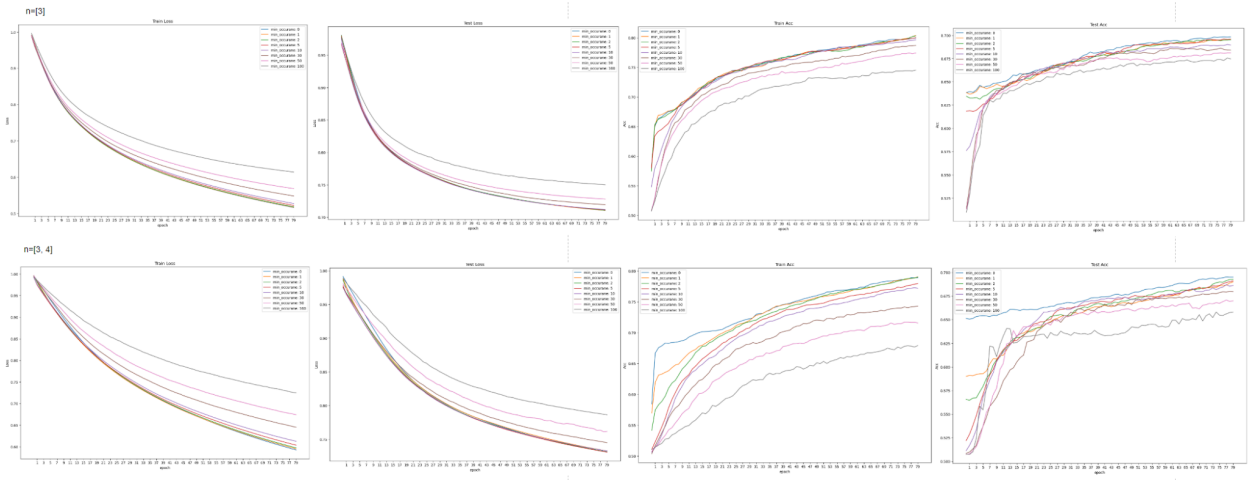
5.3.1 N-gram – Byte Based: n

All compared techniques for input vector preprocessing required identical conditions. Therefore, the highest value of n in n-grams was determined as the first step. Most NLP tasks work usually with n = {1,2,3}. The higher value of n (4-grams, 5-grams, etc.) has significant demands on hardware and software, calculation time, and overall performance. On the other hand, the potential contribution of the higher n-grams in increasing the accuracy of created models is limited.

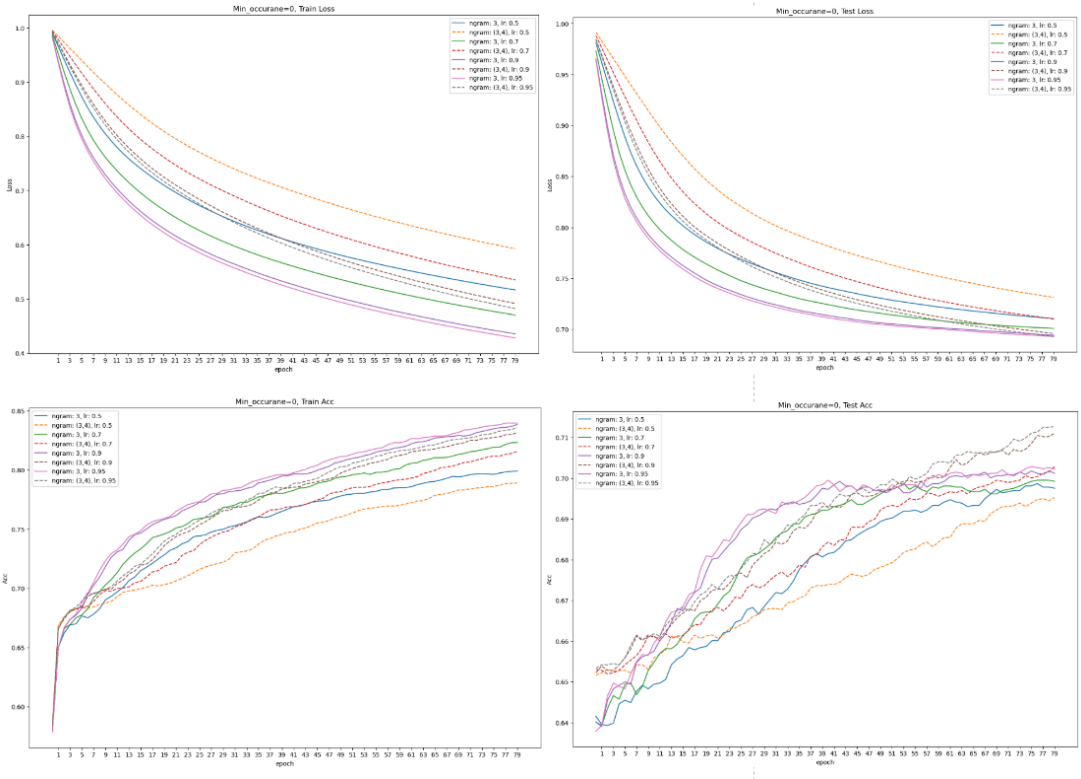
Values for n-gram n tested are: [[1], [2], [3], [1,2], [1,3], [2,3], [3, 4], [1,2,3], [2,3,4]].
Associated feature dimensions: [137, 1058, 7670, 1136, 7748, 8669, 37167, 8747, 38166].



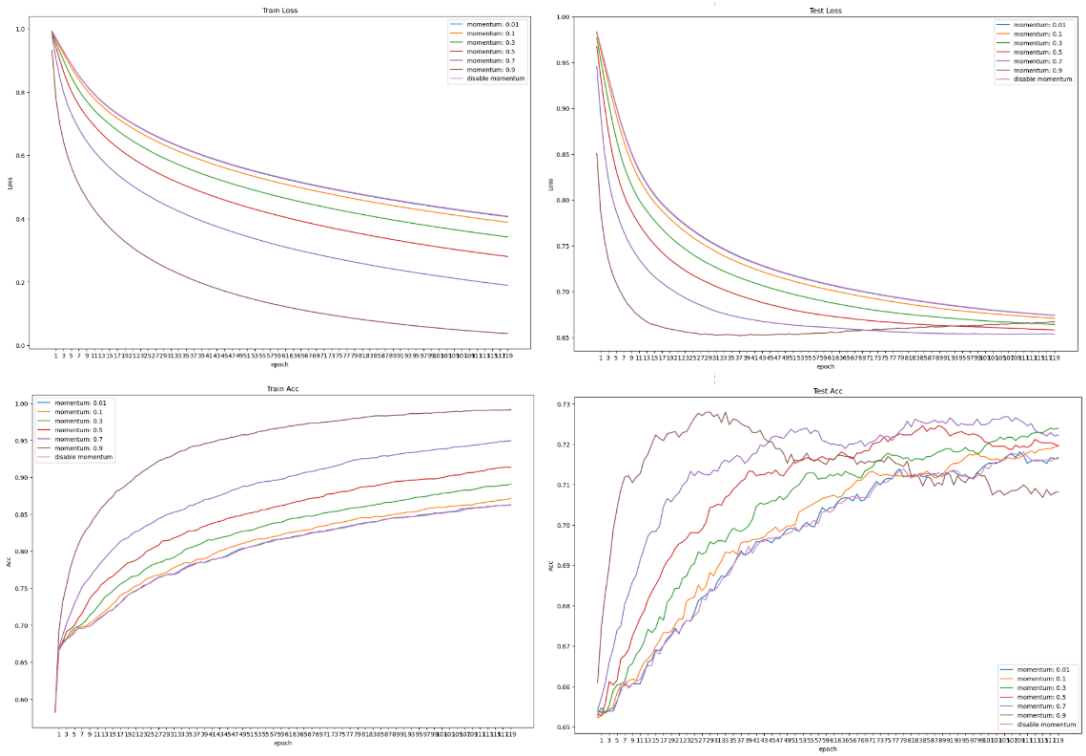
5.3.2 N-gram – Byte Based: min_occurane



5.3.3 N-gram – Byte Based: Learning Rate



5.3.4 N-gram – Byte Based: Momentum



	Model	Best Train Loss	Best Test Loss	Best Train Acc	Best Test Acc	Train Time (in sec)	Converge Epoch
Character-Based N-gram(n=(3,4)) min_occurane=0 lr=0.95 epoch=120	momentum=0.9	0.03699	0.6520	0.9909	0.72763	76.5160	26
	momentum=0.7	0.18973	0.65333	0.949071	0.72734	82.6407	105

The above is the final model configurations we choose for letter-based ngram.

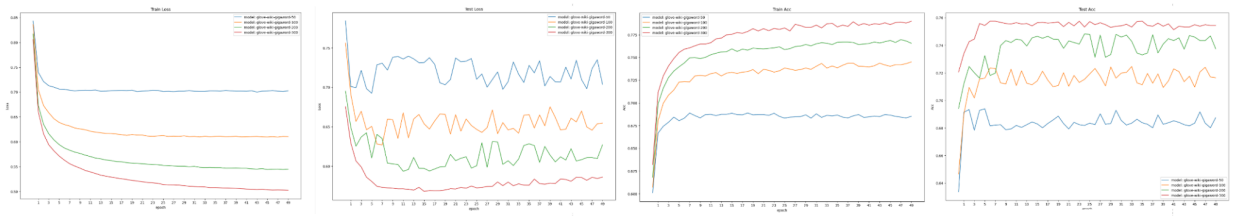
Compared to the word based n gram in Section 5.2, the best accuracy scores on the test set from byte-based n gram is lower. The byte based n gram has a smaller vocabulary, since the combination of letters are more limited. This also suggests that byte-based n grams hold much less information. They are usually utilized in such tasks as language identification, writer identification (i.e. fingerprinting), anomaly detection[14]. In addition, byte-based n-grams converge faster than word-based ones. The limited information and dimension leads to easy-overfitting and low generalization.

5.4 Gensim Pretrained Models

5.4.1 Pretrained Models

We mainly tested different pre-trained vectorization models from the Gensim module. In this section, we will focus on the pretrained "glove-wiki-gigaword" with different embed dimensions.

We start with a learning rate of 0.5 and disabled momentum.

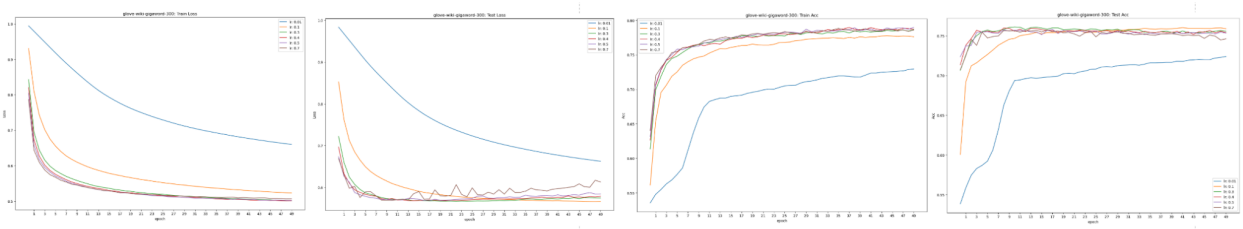


Observations:

1. When we use a pretrained model that embed the vectorization to 300 dimensions, the classification has the best accuracy scores on the test set. We will use "glove-wiki-gigaword-300" for later experiments.
2. Using pretrained models makes the classifier converge faster. Classifiers all overfit with different pretrained models. In later sections, we could either decrease the epoch amount, or to lower the learning rate.

5.4.2 Learning Rate

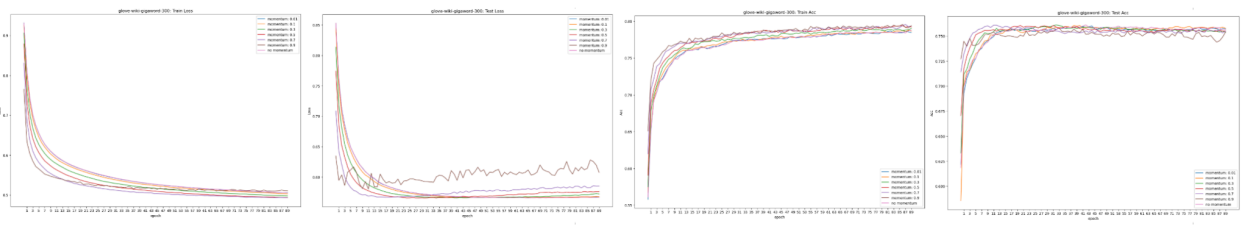
From Section 5.4.1, we could notice that the model overfits before the 20th epoch. This suggests that learning rate 0.5 updates the model too aggressively. Therefore, we will mainly compare the model performances with lower learning rate.



Observations:

1. Learning rate 0.01 is too small for our model to learn. Learning rate 0.7 leads to overfitting quickly.
2. Learning rates 0.1, 0.3, 0.4 and 0.5 are more suitable to our task and data, among which learning rate 0.1 has the best performances.

5.4.3 Momentum



Since the model converges and overfits fast in the previous sections, we know that using this pretrained model should avoid aggressive updates. The learning curves also suggest this. With momentum set to 0.9, the model overfits before finishing the first 20 epochs.

	Best Train Loss	Best Test Loss	Best Train Acc	Best Test Acc	Train Time (in sec)	Converge Epoch
glove-wiki-gigaword-300 lr = 0.1 momentum = 0.1	0.502963	0.56548	0.787281	0.7597	3.07006	80

Using pre-trained Word2Vec models enables the classifier to have better generalization ability, with train and test loss following a similar trend and ending up with similar lowest loss values.

6 Summary

We tested four methods to convert the textual contents to vectors in numbers, and the optimal solution results for each are:

1. Bag-of-words
    - Configuration: min_occurane=0, learning_rate=0.9, no momentum
    - Test Loss: 0.6367, Test Acc: 0.7293
2. Word-based N-gram
    - Configuration: n=[1,2], min_occurane=1, learning_rate=0.9, momentum=0.3
    - Test Loss: 0.6434, Test Acc: 0.7327
3. Character-based N-gram
    - Configuration: n=[3,4], min_occurane=0, learning_rate=0.1, momentum=0.1
    - Test Loss: 0.6520, Test Acc: 0.7276
4. Pre-trained Word2Vec
    - Configuration: model="glove-wiki-gigaword-300", learning_rate=0.95, momentum=0.9
    - Test Loss: 0.5654, Test Acc: 0.7597

The pre-trained model has provided the most accurate classification with the shortest time cost. The advantages of using pre-trained models include the ability to leverage the knowledge and experience of others, the ability to save time and resources, and the ability to improve model performance.

Word-based n-gram has the second best performance, whilst the byte-based n-gram has the worst performance. Byte-based n-grams usually require less memory to hold the vocabularies, and hold much less information. The limited vocabulary size leads to easy-overfitting and low generalization easily. Byte-based n-grams are usually utilized in such tasks as language identification, writer identification (i.e. fingerprinting), anomaly detection. This is why it does not have good performances here, since it is not suitable for the classification task for sentiment mining.

BOW is weaker than word-based n-grams, but better than the character-based ngrams. During the NLP application, n-gram is used to consider words in their real order so we can get an idea about the context of the particular word; BOW is used to build vocabulary for your text dataset. An assumption is that the word order and word-probability in regards to another word is important to our task, especially when the

data itself has objectives such as the nature of reviews or comments. That is, for example, “he likes chocolate but hates vanilla” and “he likes vanilla but hates chocolate” have completely different meanings, but this type of difference can not be interpolated by BOW models.

Our current weight calculation when vectorizing texts is purely based on the occurrence count of a word. If time applies, we could try comparing TF-IDF for feature extraction, since TF-IDF not only focuses on the frequency of words present in the corpus but also provides the importance of the words. Also, we could test other pretrained models.

7 References

[1] Must Known Techniques for text preprocessing in NLP
https://www.analyticsvidhya.com/blog/2021/06/must-known-techniques-for-text-preprocessing-in-nlp/

[2] Preprocessing text for sentiment analysis
https://medium.com/@wila.me/preprocessing-text-for-sentiment-analysis-36a9296aa9e5

[3] A Gentle Introduction to the Bag-of-Words Model
https://machinelearningmastery.com/gentle-introduction-bag-words-model/

[4] Bag-of-words model https://en.wikipedia.org/wiki/Bag-of-words_model

[5] Why do we remove frequent and infrequent words when in NLP?
https://www.quora.com/Why-do-we-remove-frequent-and-infrequent-words-when-in-NLP

[6] n-gram https://en.wikipedia.org/wiki/N-gram

[7] 'Word2Vec' object has no attribute 'infer_vector'
https://stackoverflow.com/questions/75023586/word2vec-object-has-no-attribute-infer-vector

[8] How to download pre-trained models and corpora
https://radimrehurek.com/gensim/auto_examples/howtos/run_downloader_api.html

[9] Gradient of Hinge loss https://stats.stackexchange.com/questions/4608/gradient-of-hinge-loss

[10] Curse of dimensionality https://en.wikipedia.org/wiki/Curse_of_dimensionality

[11] N-gram Language Models https://web.stanford.edu/~jurafsky/slp3/3.pdf

[12] Classifier accuracy decreases as n of n-gram models increases. Is this expected?
https://stats.stackexchange.com/questions/301442/classifier-accuracy-decreases-as-n-of-n-gram-models-increases-is-this-expected

[13] Why is it recommendable to have low momentum at the beginning of learning and increase it later when training a Deep Neural Network?
https://www.quora.com/Why-is-it-recommendable-to-have-low-momentum-at-the-beginning-of-learning-and-increase-it-later-when-training-a-Deep-Neural-Network

[14] Bytes vs Characters vs Words - which granularity for n-grams?
https://stackoverflow.com/questions/21656861/bytes-vs-characters-vs-words-which-granularity-for-n-grams