Fully Automatic Text Data Augmentation

CS7322 – Introduction to Natural Language Processing

Léon Lee Yuan

2024-05-07

Disclaimer

Some of the code and text in this project are automatically completed by the GitHub Copilt and Open AI ChatGPT4 given by my writing prompt and idea. I have reviewed and modified the code and text to fit the context of the project.

Introduction

Some machine learning and deep learning models require a large amount of data to train effectively. However, in many cases, the amount of data available is limited. Some natural language neural networks, such as BERT, require a large amount of data to overcome overfitting issue. If overfitting occurs, the model will not generalize well to new data. To address this issue, data augmentation is a technique that can be used to artificially increase the size of the training dataset. It is a common technique used in computer vision, such as flipping, rotating, and cropping images. However, data augmentation for text data is more challenging because the meaning of the text must be preserved. Hiring human annotators to create and edit text data is expensive and time-consuming, although it is the most effective way to augment text data and ensure the quality of the augmented data. In this project, I will explore fully automatic text data augmentation techniques that can be used to augment text data without human intervention.

Challenge

The biggest challenge in text data augmentation is to preserve the meaning and grammatical structure of the text while generating new text data. In computer vision, data augmentation techniques such as flipping, rotating, and cropping images are relatively straightforward and do not require much effort to preserve the meaning of

the image. So automatic image augmentation techniques can be easily applied to image data and it shows a very good performance boost in the model. However, text data augmentation is complete different from image data augmentation. For example, if we replace a word in a sentence with a synonym, the meaning of the sentence may change or may not change at all, or it may even become wired or meaningless. Therefore, it is important to carefully design text data augmentation techniques that can preserve the meaning and grammatical structure of the text. Unfortunately, there is no one-size-fits-all solution for text data augmentation, and some techniques may even introduce a lot of noise into the data, which can degrade the performance of the model. In this area, the research is still ongoing, and much less well-defined research literature is available compared to computer vision. This project is exploring a unknown area of text data augmentation and it is a challenging task.

Research Question

The research question of this project is: Is it possible by myself to develop fully automatic text data augmentation techniques that can be used to augment text data without human intervention? The goal is to explore and develop different text data augmentation techniques that can be used to generalize well to new data and improve the performance of natural language processing models. This is an open-ended research question, and the answer is not known in advance.

Exsiting Papers

I found a few papers that are related to text data augmentation. The first paper is the most inspiration one, "LEARNING THE DIFFERENCE THAT MAKES A DIFFERENCE WITH COUNTERFACTUALLY-AUGMENTED DATA" published by Divyansh Kaushik, Eduard Hovy, Zachary C. Lipton in 2020. They hired a lot of human annotators to create counterfactually-augmented data by editting the original data and let human review the augmented data. Their augmented text data is high quality because they are consistent and coherent in the meaning. They indeed found that the performance of the model trained on the augmented data is better than the model trained on the original data. However, the cost of hiring human annotators is very high and it is not scalable to large datasets. This is the reason why I want to explore fully automatic text data augmentation techniques that can be used to augment text data without human intervention. The second paper is "EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks" published by Jason Wei and Kai Zou in 2019. They proposed a simple data augmentation technique called EDA (Easy Data Augmentation) that can be used to augment text data. EDA consists of four simple operations: synonym replacement, random insertion, random swap, and random deletion. They found that EDA can boost the performance of convolutional and recurrent neural networks. So I will apply EDA on the AMIC model and BERT model to see if it can improve the performance of the models. The third paper is "Unsupervised Data Augmentation for Consistency Training" published by Qizhe Xie, Zihang Dai, Eduard Hovy, Minh-Thang Luong, Quoc V. Le in 2020. They applied the back-translation technique to augment text data and found that it can improve the performance of the mode. The fourth paper is "BAE: BERT-based Adversarial Examples

for Text Classification" published by Siddhant Garg and Goutham Ramakrishnan in 2020. They proposed a BERT-based adversarial example generation technique that can be used to augment text data.

Methodology

I will introduce the following text data augmentation techniques in this report. The first technique is Synonym Replacement. The 2nd one is Random Insertion. The 3rd one is Random Swap. The 4th one is Random Deletion. The 5th one is Negation Addition. The 6th one is Back Translation. The 7th one is BERT Mask Prediction. These techniques are introduced in the above mentioned papers. However, my implementation is different from the original papers. As a result, the performance of the models should have some differences.

Synonym Replacement

I applied the pos_tag function from nltk package to tag each word in the sentence with its part of speech. Then I randomly some selected words that are tagged as nouns, verbs, adjectives, and adverbs to replace with their synonyms searched through the synsets in the wordnet. Here is a quick example of the synonym replacement technique. The original sentence is "I love listening to good music." The augmented sentence is "I love listening to full euphony." The word "good" is replaced with its synonym "full" and the word "music" is replaced with its synonym "euphony". In this case, the meaning of the sentence is preserved and the augmented sentence is grammatically correct. The following is the code implementation of the synonym replacement technique.

```
def synonym_replacement(sentence, num_replacement=2):
    words = word_tokenize(sentence)
    stop_words = set(stopwords.words('english'))
    tagged = pos_tag(words)

    # Only replace nouns, adjectives, verbs, and adverbs
    allowed_pos = ['NN', 'JJ', 'VB', 'RB']

    synonyms = {}
    for word, pos in tagged:
        if word.lower() not in stop_words and pos in allowed_pos:
            synsets = wordnet.synsets(word, pos=wordnet_tag(pos))
            for synset in synsets:
                for lemma in synset.lemmas():
                    synonym = lemma.name().replace('_', ' ')
                    if synonym != word and synonym not in synonyms:
                        synonyms[word] = synonym
```

```
                        break
                if word in synonyms:
                    break

    # Randomly replace words with synonyms
    to_replace = random.sample(list(synonyms.keys()), min(num_replacement, len(synonyms)))
    new_words = [synonyms[word] if word in to_replace else word for word in words]
    return ' '.join(new_words)
```

Negation Addition

I compiled a list of negation words such as “not”, “never”, “no”, “hardly” and so on. Then I applied the pos_tag function to tag each word to locate the adjectives, adverbs, verbs. I randomly selected one negation from the list to add right before the selected word that is NOT followed by another negation word. After this, I flipped the sentiment label. Here is a quick example of the negation addition technique. The original sentence is “I love cute pandas!”. The augmented sentence is “I not love cute pandas!”. The word “love” is negated by adding the negation word “not” before it. As you can tell, this addition changes the sentiment of the sentence but doesn’t hold the grammatical rule. The following is the code implementation of the negation addition technique.

```
def negate_sentence(sentence):
    words = word_tokenize(sentence)
    tagged = pos_tag(words)

    # List of negation words
    negation_words = ["not", "never", "no", "hardly", "scarcely", "barely"]

    # Parts of speech to negate
    target_pos = {'JJ', 'RB', 'VB'}  # Adjectives, Adverbs, Verbs

    negated_sentence = []
    just_negated = False
    for word, pos in tagged:
        if pos.startswith(tuple(target_pos)) and not just_negated:
            # Randomly select a negation word
            negation_word = random.choice(negation_words)
            negated_sentence.append(negation_word)
            just_negated = True
        else:
            just_negated = False
```

```
        negated_sentence.append(word)

    return ' '.join(negated_sentence)
```

Random Insertion

I compiled a list of neutral words such as “indeed”, “perphas” and “generally”. Then I randomly selected one position in the sentence to insert one random word from the neutral word list. Repeat this process for a few times depending how long the original sentence is. Here is a quick example. The original sentence is “The movie is fun!”. The augmented sentence is “The sometimes movie perphas indeed is fun!”. The words “sometimes”, “perphas”, and “indeed” are randomly inserted into the sentence. Although the sentiment keeps the same, the meaning of the sentence is changed a bit. The following is the code implementation.

```
def random_insertion(sentence, neutral_words, n=2):
    words = word_tokenize(sentence)
    for _ in range(n):
        # Select a random neutral word to insert
        new_word = random.choice(neutral_words)
        # Select a random position to insert the word
        position = random.randint(0, len(words))
        words.insert(position, new_word)
    # Detokenize the modified word list back into a sentence
    return TreebankWordDetokenizer().detokenize(words)
```

Random Deletion

I applied the pos_tag function to tag each word. The I randomly selected some words tagged as nouns, verbs, adjectives, adverbs and so on to delete. Here is a quick example. The original sentence is “The weather is sunny and clear!”. The augmented sentence would be “The weather is sunny and !”. The word “clear” is deleted from the sentence. The following is the code implementation.

```
def random_deletion(sentence, deletion_rate=0.3):
    words = word_tokenize(sentence)
    pos_tags = pos_tag(words)  # Get POS tags to help identify less sentiment-critical words

    # Keep important words based on their POS tags, likely adjectives, nouns, and verbs
    important_tags = {'JJ', 'NN', 'NNS', 'NNP', 'NNPS', 'VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ'}
    filtered_words = [word for word, tag in zip(words, pos_tags) if tag[1] not in important_tags
```

```
    # Make sure the sentence is not empty after deletion
    if len(filtered_words) == 0:
        return sentence  # Return original if all words were to be deleted
    return ' '.join(filtered_words)
```

Random Swap

I first found all non-stops words in the sentence. Then I randomly selected two words from all non-stop words to swap their positions. Then I repeated this swap operation for a few times depending on the length of the sentence. Here is a quick example. The original sentence is "I plan to become a pilot.". The augmented sentence is "I pilot to become a plan." The words "plan" and "pilot" are swapped. The following is the code implentation.

```
def random_swap(sentence, n=2):
    words = word_tokenize(sentence)
    stop_words = set(stopwords.words('english'))
    non_stop_words_indices = [i for i, word in enumerate(words) if word.lower() not in stop_words]

    # Limit the number of swaps to the minimum of 'n' or half the number of nonstop words
    num_swaps = min(n, len(non_stop_words_indices) // 2)

    for _ in range(num_swaps):
        idx1, idx2 = random.sample(non_stop_words_indices, 2)
        # Perform the swap
        words[idx1], words[idx2] = words[idx2], words[idx1]

    # Reconstruct the sentence
    return ' '.join(words)
```

BERT Mask Prediction

I first applied the pos_tag function to tag each word in the sentence. Then I randomly selected some words tagged as nouns, verbs, adjectives, and adverbs to mask. I used the BERT model to predict the masked words without training. The original sentence is "I am learning and practicing my Dota 2 skills at home." The word "home" is masked by the function. The BERT model predicts the masked word as "home". The augmented sentence is "I am learning and practicing my Dota 2 skills at home." As you can see, this augmentation produces the same sentence as the original one. So it doesn't actually augment the text data. The following is the code implementation.

```python
def mask_tokens(sentence, mask_rate=0.7):
    # Tokenize and get part of speech tags
    words = word_tokenize(sentence)
    tagged = nltk.pos_tag(words)
    non_sentiment_tags = {'JJ', 'RB', 'VB', 'NN', 'IN', 'DT', 'CC', 'RP'}  # Example tags
    selected_for_masking = [word for word, tag in tagged if tag in non_sentiment_tags]
    #print(selected_for_masking)

    # Randomly mask some of the selected words
    num_to_mask = int(len(selected_for_masking) * mask_rate)
    #print(num_to_mask)
    masked_words = random.sample(selected_for_masking, num_to_mask)
    #print(masked_words)
    masked_sentence = [word if word not in masked_words else '[MASK]' for word in words]
    return ' '.join(masked_sentence), masked_words

def predict_masked_tokens(masked_sentence):
    # Tokenize input sentence and convert to tensor and use try to capture error
    try:
        input_ids = tokenizer.encode(masked_sentence, return_tensors='pt')
    except:
        print("Error in tokenization")
        return masked_sentence

    # Predict all tokens
    with torch.no_grad():
        outputs = model(input_ids)
        predictions = outputs[0]

    # Get predicted tokens
    predicted_index = torch.argmax(predictions[0], dim=1)
    predicted_tokens = [tokenizer.convert_ids_to_tokens(idx.item()) for idx in predicted_index]

    # Replace '[MASK]' with predicted tokens
    masked_indices = (input_ids == tokenizer.mask_token_id).nonzero(as_tuple=True)[1]
    for idx in masked_indices:
        masked_sentence = masked_sentence.replace('[MASK]', predicted_tokens[idx], 1)
    return masked_sentence
```

Back Translation

I compiled a list of intermiate target language such as Turkish, German, Spanish, Chinese and so on. For each text, I randomly selected one target language to translate text into the target language and then translate back to the original language. Here is a quick example. The original sentence is "This movie is fun to watch and many people like it." The target language is Chinese. The translated sentence is " " Then I back translated it into English. The back translated sentence is "This movie is funny and a lot of people enjoyed it." The meaning of the sentence is preserved and there is no grammatical error. The following is the code implementation.

```python
#from google.cloud import translate_v2 as translate
#translate_client = translate.Client()
def translate_text(target: str, text: str) -> str:
    """Translates text into the target language and returns only the translated text."""
    if isinstance(text, bytes):
        text = text.decode("utf-8")

    result = translate_client.translate(text, target_language=target)
    return result['translatedText']
```

Experimental Dataset

The dataset I used in this project is the Wine Review. This dataset is labeled with the sentiment of the review, either positive or negative. The dataset originally contains 130k reviews. I randomly selected 5,000 reviews for the experiment to speed up the training process. I also randomly selected 7,095 rows from the dataset as the validation set, 7,096 rows as the test set. I used this validation set to evaluate the performance of the model during training. I used the test set to evaluate the performance of the model after training. For each augmentation technique, I generated 5,000 augmented samples from the original 5,000 samples by one-on-one relationship. So I have 10,000 samples in total for each augmentation technique. To keep the binary class balance, I selected the orginal 5,000 samples with roughly 50% positive and 50% negative sentiment.

Experimental Model

I used two models in this project to evaluate the performance of the text data augmentation techniques. The first model is the AMIC model. The full name is Attention-Based Multiple Instance Classification. This new proposed model is designed to classify the sentiment of the text data. The model was designed by Chenyu Yang and Jing Cao at SMU in 2023. They also published a paper about their model in the Annals of Applied Statistics. The AMIC consists of a self-attention class, tiedLinear class, PositionalEncoding class, Mask_block

class, Snetiment block class, Synthesizer class, and Embeds class. The advantage of the AMIC model is that it can capture the sentiment score of each word in the sentence and then aggregate the sentiment scores to make the final prediction. It is also very lightweight compared to the traditional large language model, BERT. The AMIC is a shallow neural network. It can achieve the equivalent performance of the BERT model with much less computational resources and training time. The second model is the BERT model. The BERT model I used is google-bert/bert-base-uncased which ignores the case of the text data and is most widely used. The BERT model is a large language model that can capture the context of the text data and achieve state-of-the-art performance on various natural language processing tasks. The BERT model is pre-trained on a large corpus of text data and fine-tuned on the sentiment classification task. The BERT model is computationally expensive and requires a large amount of data to train effectively. The AMIC model is trained from scratch on the Wine Review dataset. The BERT model is fine-tuned on the Wine Review dataset.

Experimental Set Up

I used the PyTorch library to implement the AMIC model and the BERT model. The whole code is written in Python. I used the Hugging Face Transformers library to load the pre-trained BERT model and fine-tune it on the Wine Review dataset. The training platform is google colab. The GPU is NIVIDIA T4. The RAM is high mode. The code is written in Jupyter Notebook. The loss function is Binary Cross Entropy. The optimizer is Adam. All the learning rates for the AMIC model are 0.0002. The initial step size for the AMIC model is 2 and the gamma is 0.65 to reduce the step size. The batch size for AMIC model is 64. The number of epochs for AMIC model is 15. The learning rate for the BERT model is 2e-5. The batch size for the BERT model is 1. The optimizer for the BERT model is AdamW. The epoch for the BERT model is 1. I found more epochs for the BERT model is not necessary because the model is already pre-trained on a large corpus of text data and more epochs didn't improve the test accuracy. The training process is the same for all the text data augmentation techniques. I first trained the model on the original dataset. Then I trained the model on the augmented dataset including the original datset. I evaluated the performance of the model on the test set using the accuracy metric. The accurary metric is the percentage of the correctly predicted sentiment labbels.

Experimental Results

This section presents the experimental results of the text data augmentation techniques on the AMIC model and the BERT model. The performance of the models is evaluated on the test set using the accuracy metric. Because I make the binary sentiment labels in balance, the accuracy metric itself is a good metric to evaluate the performance of the models. The following tables show the test accuracy of the AMIC model and the BERT model trained on the original dataset and the augmented dataset using different text data augmentation techniques. The sample size of the training set is 5,000 for the original dataset and 10,000 for the augmented dataset for the AMIC model. The sample size of the training set is 30,000 and 60,000 for the augmented dataset for the BERT model. As we can see, the test accuracy trained on the original dataset with the AMIC model

is 82.47%. Except for the negation addition, all other techniques have 82% accuracy that is close enough to the original dataset. Negation Addition only has a 59.32% test accuracy. This means my negation method is not effective. My current negation method is easy to break the grammatical rule and change the sentiment of the sentence. For example, the original sentence is "This new movie is good and fun to watch.". After adding negation words, the augmented sentence can be "This new movie is NOT good and fun to watch.". In such way, the sentiment of sentence becomes unclear and we can't tell if it is positive or negative. There are many issues occurring in the negation addition. This negation addition introduces a lot of garbage in, as a result, the garbage out is also high. Only Random Swap slightly improves the performance of the model.

Training Set	Sample Size	Test Accuracy
Original Dataset	5K	82.47 %
Synonym Replacement	10K	82.28 %
Negation Addition	10K	59.32 %
Random Insertion	10K	81.34 %
Random Deletion	10K	82.01 %
Random Swap	10K	82.84 %
BERT Mask Predict	10K	81.62 %
Back Translation	10K	82.06 %

The next table shows the test accuracy of the BERT model trained on the original dataset and the augmented dataset using different text data augmentation techniques. The sample size of the training set is 30,000 for the original dataset and 60,000 for the augmented dataset. As we can see, the test accuracy trained on the original dataset with the BERT model is 87.89%. Only Synonym Replacement and Random Deletion slightly improve the performance of the model on the test set. Other models have similar performance to the original dataset.

Training Set	Sample Size	Test Accuracy
Original	30K	87.89 %
Synonym Replacement	60K	88.16 %
Negation Addition	60K	87.89 %
Random Insertion	60K	87.22 %
Random Deletion	60K	88.80 %
Random Swap	60K	87.68 %
BERT Mask Predict	60K	87.57 %
Back Translation	60K	87.92 %

Conclusion

In this project, I explored fully automatic text data augmentation techniques that can be used to augment text data without human intervention. I implemented 7 text augmentation techniques such as Synonym Replacement,

Negation Addition, Random Insertion, Random Deletion, Random Swap, BERT Mask Prediction, and Back Translation. The Negation Addition actually degraded the performance of the AMIC model when the sample size is small. Other text augmentation techniques have similar performance to the original dataset. The random swap, synonym replacement, and random deletion slightly improved the performance. However, most techniques didn't show a significant improvement in the performance of the models. One possible reason is that the test accuracy on the original dataset is already high enough. So there is not enough space for the augmented dataset to improve the performance. Another possible reason is that the text data augmentation techniques introduced a lot of noise into the data, which can degrade the performance of the model. Actually, designing a good text data augmentation technique is a challenging task. It requires a deep understanding of the text data and the ability to preserve the meaning and grammatical structure of the text. In the future, I will try augmentation techniques on a original dataset with low accuracy to see if the augmentation techniques can improve the performance of the model. I will also explore more advanced text data augmentation techniques that can be used to augment text data without human intervention. Another possible direction is to combine multiple text data augmentation techniques to improve the performance of the model. A mixed automatic and human intervention augmentation technique can be a good solution to improve the performance of the model.