Trabalho de Conclusão de Curso

Replicando Abordagem para Geração Automática de Tratadores de Exceções

Eric dos Santos Coelho
esc2@ic.ufal.br

Orientadores:

Prof. Dr. Baldoino Fonseca dos Santos Neto
Jairo Raphael Moreira Correia De Souza

Maceió, Janeiro de 2023

Eric dos Santos Coelho

Replicando Abordagem para Geração Automática de Tratadores de Exceções

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Prof. Dr. Baldoino Fonseca dos Santos Neto

Jairo Raphael Moreira Correia De Souza

Maceió, Janeiro de 2023

Catalogação na Fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico

Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

C672r Coelho, Eric dos Santos.

Replicando abordagem para geração automática de tratadores de exceção / Eric dos Santos Coelho. – 2023.

26 f. : il.

Orientador: Baldoino Fonseca dos Santos Neto.

Co-orientador: Jairo Raphael Moreira Correia de Souza.

Monografia (Trabalho de conclusão de curso em Ciência da Computação) – Universidade Federal de Alagoas, Instituto de Computação. Maceió, 2023.

Bibliografia: f. 26.

1. Tratamento de exceções (Computação). 2. *Deep learning* (Aprendizado do computador). 3. Engenharia de *software*. I. Título.

CDU: 004.41

Agradecimentos

Agradeço aos meus pais e meus irmãos que me apoiaram ao longo de toda esta jornada.

Agradeço aos meus orientadores, professor Baldoino e Jairo Raphael, que com seus conselhos me auxiliaram diretamente na conclusão deste trabalho.

Por fim, agradeço aos meus colegas, com os quais convivi durante minha graduação, que me ajudaram a continuar e finalmente chegar aqui.

Eric Coelho

Resumo

Neste trabalho, replicamos uma abordagem baseada em aprendizado profundo para tratamento automatizado de exceções, mas não conseguimos obter o mesmo desempenho relatado originalmente. O modelo foi implementado utilizando o código-fonte e o conjunto de dados disponibilizados pelo autor do trabalho original. Para realizar a abordagem foi utilizado o ambiente Google Colab, que fornece um *runtime* para executar *scripts* na linguagem Python. Utilizando a configuração da versão gratuita obtivemos uma redução de 0,56% na acurácia e de 0,12% no *F1 - score*, derivados de um aumento de 1,98% na sensibilidade e uma perda de 2,59% na precisão do modelo localizador de blocos *try*. Entretanto, o modelo gerador de blocos *catch* apresentou um ganho de 0,1% na acurácia e de 0,2% no *BLEU score*. Por fim, concluímos que as causas prováveis de tais divergências foram a configuração dos hiperparâmetros do modelo e as diferenças entre os ambientes utilizados.

Palavras-chave: Reprodutibilidade, Tratamento de exceções, *Deep Learning*, *Machine Learning*, Engenharia de *software*.

Abstract

In this work, we replicated an approach based on deep learning for automated exception handling, but we were unable to obtain the same performance originally reported. The model was implemented using the source code and data set provided by the author of the original work. To carry out the approach, the Google Colab environment was used, which provides a runtime to execute scripts in the Python language. Using the configuration of the free version, we obtained a reduction of 0.56% in accuracy and 0.12% in the F1 - score, resulting from an increase of 1.98% in sensitivity and a loss of 2.59% in model precision try block locator. However, the catch block generator model showed a gain of 0.1% in Accuracy and 0.2% in the BLEU score. Finally, we conclude that the probable causes of such divergences were the configuration of the model's hyperparameters and the differences between the environments used.

Key-words: Reproducibility, Exception handling, Deep Learning, Machine Learning, Software engineering.

Lista de Figuras

Lista de Tabelas

Lista de Quadros

Conteúdo

1

Introdução

1.1 Motivação

Com o avanço da tecnologia e o desenvolvimento de sistemas cada vez mais complexos, os desenvolvedores tem enfrentado problemas em lidar com a grande quantidade de situações atípicas que podem resultar em falhas durante a execução de tais sistemas. Segundo Garcia et al. 2001, empregar técnicas de tratamento de exceções enquanto os desenvolvedores procuram satisfazer requisitos relacionados à confiabilidade, manutenibilidade e reutilização, continua sendo uma grande preocupação para os desenvolvedores de sistemas orientados a objetos.

Os mecanismos de tratamento de exceção, presentes em várias linguagens de programação modernas, fornecem um meio em que os desenvolvedores possam declarar código para lidar com as situações excepcionais. Segundo Garcia et al. 2001, tal mecanismo deve ser projetado adequadamente, pois caso contrário, pode tornar uma aplicação não confiável, de difícil compreensão, manutenção e reuso.

Para isto, algumas técnicas de geração automática de código foram propostas para auxiliar os desenvolvedores no processo de escrita de código de tratamento de exceção. O trabalho de J. Zhang et al. 2020 propõe uma nova abordagem utilizando redes neurais para gerar tais códigos, que consiste em prever localizações de blocos *try* e gerar automaticamente os blocos *catch* completos em linguagem Java.

A técnica de aprendizado profundo (*deep learning*) é bastante utilizada para aprender automaticamente padrões de grandes quantidades de dados, mas requerem uma grande quantidade de processamento. Com base nesta técnica, J. Zhang et al. 2020 utiliza uma grande quantidade de código minerado de repositórios *open source* do GitHub, e projeta dois modelos baseados em redes neurais profundas para lidar automaticamente com as exceções dos trechos de código:

1. Prever as localizações dos blocos *try* através da identificação do código-fonte que precisa lidar com possíveis exceções;

2. Gerar blocos *catch* completos para lidar com as exceções.

Apesar da importância do tratamento de exceções, os desenvolvedores tendem a evitar o uso do mecanismo, tornando o código mais propenso a erros. Em seu trabalho, J. Zhang et al. 2020 coletou milhões de métodos escritos na linguagem Java em 2.000 projetos relevantes no GitHub e encontrou uma proporção de 31,2% de código que não utiliza blocos *catch* ou utilizam, mas não fazem nada quando exceções ocorrem. Portanto, é essencial propor técnicas para auxiliar os desenvolvedores a escrever código para tratar exceções.

Neste trabalho será apresentada a tentativa de replicação da implementação do artigo Learning to Handle Exceptions de J. Zhang et al. 2020. Segundo Voets, Møllersen e Bongo 2018 a replicação de um trabalho significa a tentativa de repetir o estudo como descrito. Além disso, se os dados que produziram os resultados relatados estiverem disponíveis, o método replicado deve reproduzir os resultados originais. Apesar de terem sido disponibilizados o código-fonte e o conjunto de dados original, não obtivemos os mesmos resultados. As diferenças encontradas podem ter sido provocadas pela configuração dos hiperparâmetros e do ambiente utilizados na replicação.

1.2 Objetivo

Este trabalho tem como objetivo a reproduzir a implementação do artigo intitulado *Learning to Handle Exceptions* de J. Zhang et al. 2020 utilizando o código-fonte e o conjunto de dados disponibilizados pelo autor.

1.3 Estrutura do documento

O documento está organizado da seguinte forma. O Capítulo 2 apresenta os fundamentos da abordagem. O Capítulo 3 descreve a formulação do problema do tratamento automatizado de exceções e os detalhes da implementação de J. Zhang et al. 2020. No Capítulo 4 são apresentados o procedimento de avaliação e os resultados da execução da abordagem. Por fim, o trabalho é concluído no Capítulo 5.

2

Fundamentos

Neste capítulo, serão apresentados alguns fundamentos para a abordagem desenvolvida por J. Zhang et al. 2020 em seu trabalho.

2.1 Tratamento de exceções

A grande maioria das linguagens de programação modernas possuem algum mecanismo para lidar com as condições excepcionais que podem ocorrem em tempo de execução. De acordo com Shah, Gorg e Harrold 2010, os mecanismos de tratamento de exceção fornecem uma maneira de lançar explicitamente condições excepcionais e uma maneira de declarar blocos de código para lidar com uma ou mais dessas condições excepcionais, .

Por exemplo, na linguagem Java as instruções que podem lançar algum tipo de erro durante a execução do programa são encapsuladas em um bloco chamado *try*. Dentro de um ou mais blocos *catch*, posicionados diretamente após o bloco *try*, é escrito o código para tratar o erro lançado. Por exemplo, no trecho 2.1

Quadro 2.1: Exemplo de par *try-catch*

```
public E next() {
    try {
      return arr[index++];
    } catch (IndexOutOfBoundsException e) {
      throw new NoSuchElementException(e.getMessage());
    }
  }
```

Este mecanismo é ser utilizado para recuperação de erros internos do programa, para solicitar que o usuário tome uma decisão ou para propagar o erro até chegar no nível em que a exceção será tratada, usando exceções encadeadas. Na Figura 2.1, o método *next* pode ten-

tar acessar uma posição que não existe no array *arr*. No bloco *catch*, o erro é capturado e o tratamento adotado foi o de relançar uma exceção diferente.

Em alguns casos, a causa da exceção pode se originar a partir de alguma declaração feita anteriormente no código, como o exemplo da Figura 2.1, onde o método *decode* da linha 8 pode lançar uma exceção, já que o processamento depende da variável *encrypted* definida anteriormente. Durante o processamento do método *decode* pode ser encontrado um erro em tempo de execução, e tal erro é propagado para o método chamador.

Figura 2.1: Exemplo de código sem tratamento de exceção

```
1  public char[] decrypt(char[] chars) {
2      if (!isEncrypted(chars)) {
3          return chars;
4      }
5      String encrypted = new String(chars, ENCRYPTED_TEXT_PREFIX.length(),
6                                    chars.length - ENCRYPTED_TEXT_PREFIX.length());
7      byte[] bytes;
8      bytes = Base64.getDecoder().decode(encrypted);
9      byte[] decrypted = decryptInternal(bytes, encryptionKey);
10     return CharArrays.utf8BytesToChars(decrypted);
11 }
```

Fonte: Adaptado de J. Zhang et al. 2020, página 2.

A Figura 2.2 ilustra o código após a implementação do mecanismo. Basicamente, é selecionada a instrução que lança a exceção para ser incluída no bloco *try* e em seguida o bloco *catch* é especificado para lidar com a exceção. Na Figura 2.2, o bloco *catch* (linhas 11-13) é adicionado para tratar a exceção.

Figura 2.2: Exemplo de código com tratamento de exceção

```
1  public char[] decrypt(char[] chars) {
2      if (!isEncrypted(chars)) {
3          return chars;
4      }
5      String encrypted = new String(chars, ENCRYPTED_TEXT_PREFIX.length(),
6                                    chars.length - ENCRYPTED_TEXT_PREFIX.length());
7      byte[] bytes;
8  (1) try {
9          bytes = Base64.getDecoder().decode(encrypted);
10     }
11 (2) catch (IllegalArgumentException e) {
12         throw new ElasticsearchException("unable to decode encrypted data", e);
13     }
14     byte[] decrypted = decryptInternal(bytes, encryptionKey);
15     return CharArrays.utf8BytesToChars(decrypted);
16 }
```

Fonte: Adaptado de J. Zhang et al. 2020, página 2.

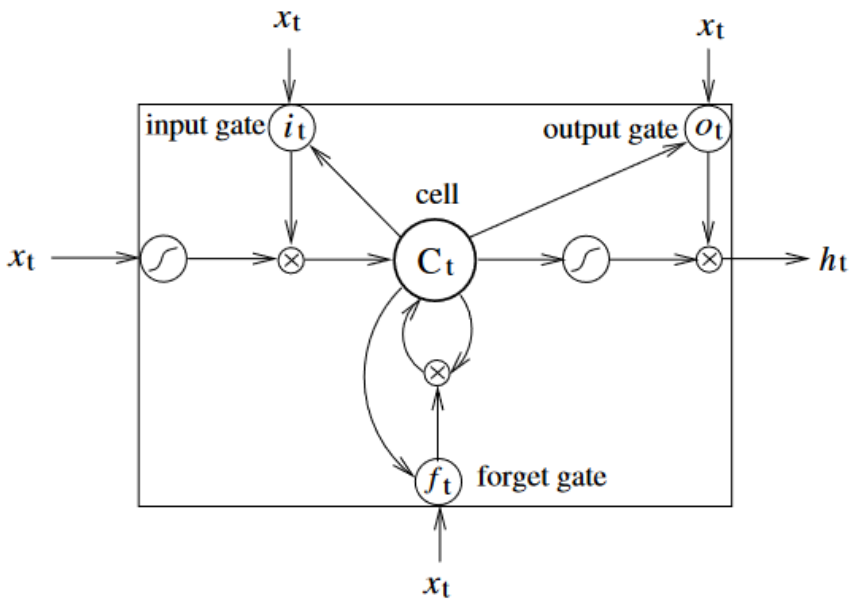
Logo, o problema pode ser subdividido em duas abordagens: classificar quais instruções lançam exceção e qual seria o código adequado para tratar as exceções. Para isso, J. Zhang et al. 2020 implementa duas arquiteturas de redes neurais cujos detalhes serão apresentados nas seções seguintes.

2.2 LSTM

Long Short-Term Memory (LSTM) é um tipo de Rede Neural Recorrente (RNN) bastante aplicada em problemas de Processamento de Linguagem Natural (NLP), como o da tradução de máquina (NMT) e classificação automática de textos.

Conforme Huang, Xu e Yu 2015, a arquitetura da LSTM é composta por células que possuem uma memória baseada em informações históricas, o que permite ao modelo prever qual a saída para o instante atual de leitura com base na informação anterior recuperada da memória. Na Figura 2.3 é ilustrada uma célula LSTM que recebe os valores da entrada de dados em sequência. O *forget gate* é o responsável por remover as informações que não são úteis, o *input gate* é o responsável adicionar informações para a célula e o *output gate* é o responsável por extrair as informações que servirão de entrada para a próxima célula LSTM.

Figura 2.3: Uma célula LSTM



Fonte: Huang, Xu e Yu 2015, página 2.

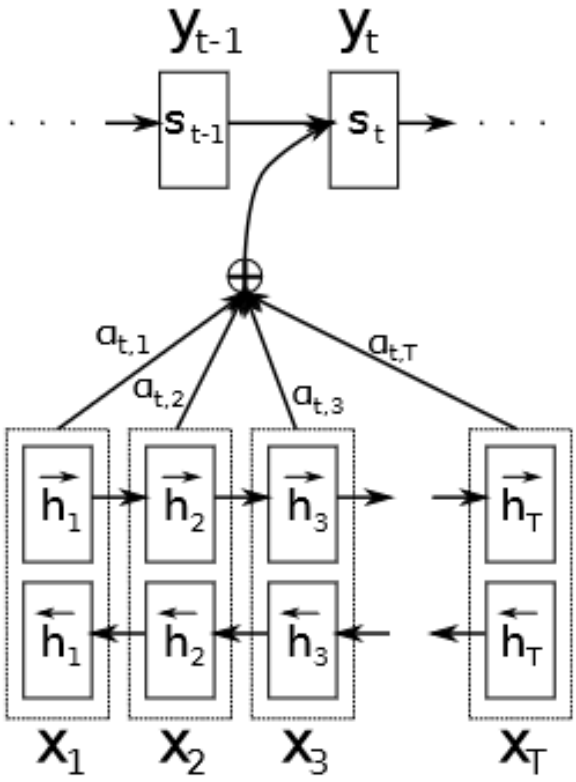
Algumas arquiteturas de redes neurais tem características que permitem um melhor desempenho em certos tipos ou formatos de dados. Um estudo feito por Yin et al. 2017 compara três arquiteturas de redes neurais mais utilizadas em tarefas relacionadas a Processamento de Linguagem Natural (NLP), as Redes Neurais Convolucionais (CNNs), as *Gated Recurrent Unit* GRU (um tipo de RNN) e a LSTM. No estudo, é concluído que as RNNs têm bom desempenho em várias dessas tarefas. Além disso, o autor observou que o tamanho das camadas ocultas e o tamanho do lote (*batch size*) podem fazer com que o desempenho varie drasticamente, sugerindo que a otimização desses dois parâmetros é crucial para um bom desempenho.

Algumas melhorias nesse tipo de rede neural vem sido adotadas para aumentar o desempenho, como o mecanismo de RNN bidirecional, proposto por Schuster e Paliwal 1997. Ba-

sicamente, uma RNN bidirecional pode ser treinada com a sequência de entrada em sentido de tempo positivo e negativo. Esta arquitetura permitiu avanços nos trabalhos de tradução de máquina.

O problema da tradução de máquina (Neural Machine Translation - NMT), consiste em ler uma sequência de palavras para gerar a tradução correspondente. Algumas trabalhos aplicaram o conceito de RNN *encoder-decoder* para solucionar esse problema, como o trabalho de Bahdanau, Cho e Bengio 2014 que propõe uma arquitetura de RNNs onde é adicionado um mecanismo para calcular a influência de cada palavra do idioma de origem ao predizer cada palavra do idioma de destino, o mecanismo de atenção ilustrado na Figura 2.4.

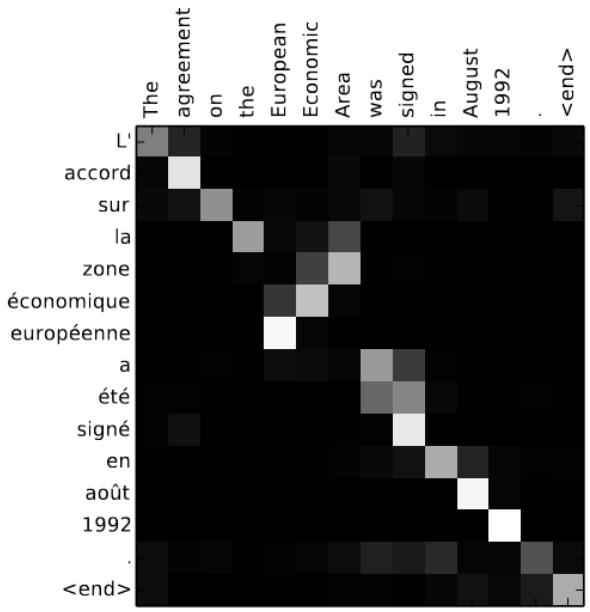
Figura 2.4: RNN com mecanismo de atenção



Fonte: Y. Zhang et al. 2018, página 1.

Por exemplo, na Figura 2.5 é realizada uma tradução do francês para o inglês. Como as classes gramaticais nem sempre são escritas na mesma ordem entre os idiomas, a tradução de uma palavra pode depender das informações contidas nas palavras em ambas as direções. Analogamente, o código-fonte de um programa também possui tal característica de dependência sequencial. No exemplo de tratamento de exceção citado anteriormente (2.2), uma instrução de código pode influenciar na execução do programa ocasionar um erro em um ponto mais à frente, por exemplo.

Figura 2.5: Exemplo de tradução utilizando RNN com mecanismo de atenção



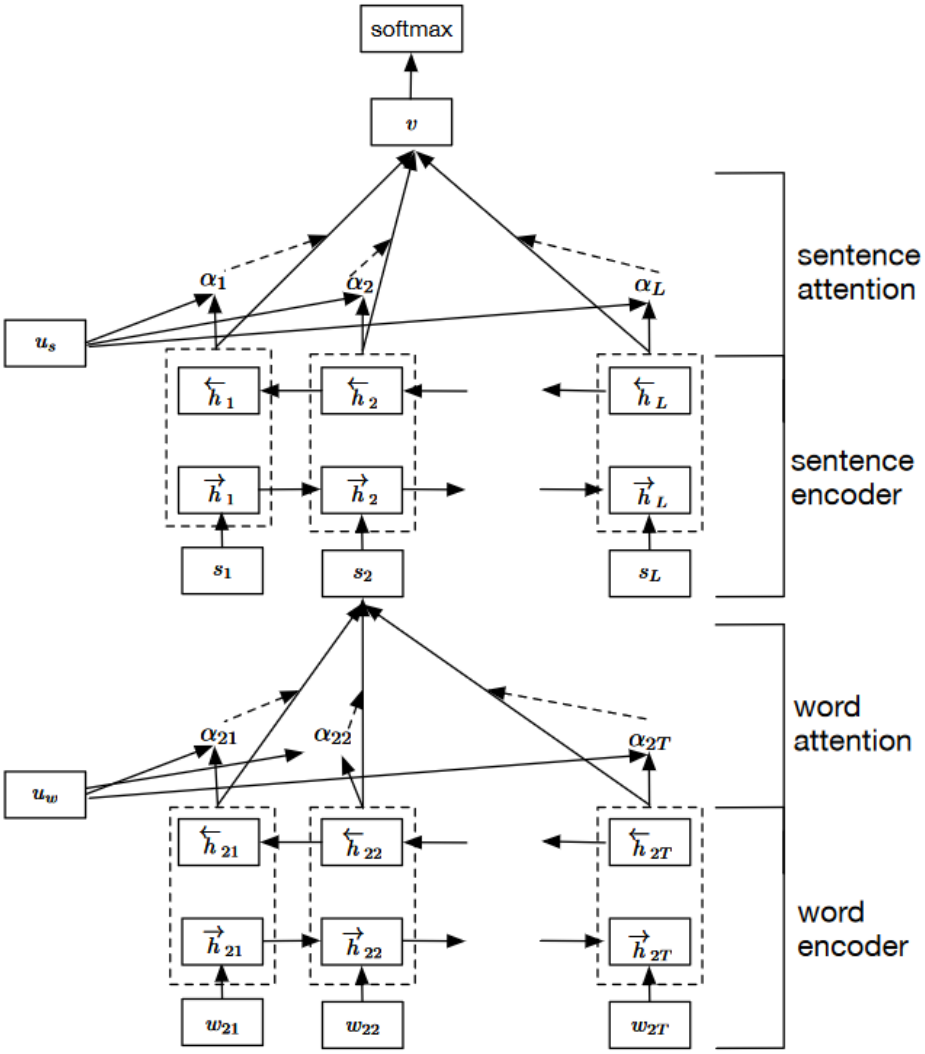
Fonte: Bahdanau, Cho e Bengio 2014, página 6.

2.3 Modelo hierárquico de atenção

O modelo hierárquico proposto por Yang et al. 2016 consiste em dois níveis de RNNs, com o objetivo de aumentar a capacidade de obter informações a partir de textos. Este modelo possui uma estrutura hierárquica que espelha a estrutura da informação presente em documentos. Os dois níveis de RNNs com mecanismo de atenção são aplicados ao nível da palavra e da sentença para calcular a influência que cada nível terá na classificação do documento completo.

Durante o treinamento da rede, são extraídos alguns vetores ao longo da hierarquia. Na Figura 2.6, os vetores u_w e u_s são utilizados para agregar a importância das informações de cada nível. u_w representa as palavras importantes para serem agregadas em vetores de sentenças u_s e, por fim, vetores de sentenças u_s são agregados em vetores de documentos.

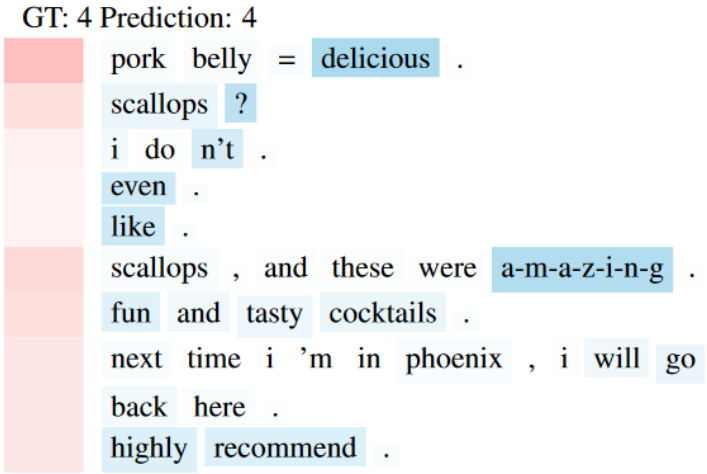
Figura 2.6: Arquitetura da rede de atenção hierárquica



Fonte: Yang et al. 2016, página 2.

Para ilustrar o modelo, na Figura 2.5 a contribuição de cada palavra para a classificação está destacada em azul e a importância de cada sentença é destacada em vermelho no canto esquerdo.

Figura 2.7: Exemplo da classificação hierárquica de documento



Fonte: Yang et al. 2016, página 8. No topo da figura, "Prediction: 4"significa 5 estrelas de classificação.

Na Figura 2.8, é apresentada uma versão simplificada que dá destaque para as duas sentenças classificadas como mais relevantes. Analogamente, o mesmo efeito pode ser considerado em um trecho de código, onde cada *token* de cada instrução pode impactar em intensidades diferentes para a classificação da instrução como um todo.

Figura 2.8: Exemplo de classificação de documento

pork belly = delicious . || scallops? || I don't even like scallops, and these were a-m-a-z-i-n-g . || fun and tasty cocktails. || next time I in Phoenix, I will go back here. || Highly recommend.

Fonte: Yang et al. 2016, página 1.

3

Nexgen

Este capítulo será destinado a apresentar a abordagem de redes neurais para geração automatizada de código de tratamento de exceções proposta por J. Zhang et al. 2020, nomeadamente Nexgen. Assim, todas as informações aqui dispostas são de autoria e/ou embasadas em seu trabalho. Os trechos de código foram extraídos do repositório[1] disponibilizado.

3.1 Visão geral

Como apresentado anteriormente, a estrutura dos mecanismos de tratamento de exceção é composta pelo bloco de código que pode lançar exceção e o bloco que captura a exceção para tratá-la. Por isso, na abordagem de automatização, J. Zhang et al. 2020 decompõe o problema em duas tarefas sucessivas, a localização do bloco *try* e a geração do bloco *catch*:

1. Localização do bloco *try*: A partir de um trecho de código, determinar quais as instruções que podem lançar exceções e que devem ser delimitadas por um bloco *try*.

2. Geração do bloco *catch*: Gerar o bloco *catch* correspondente para o tratamento das exceções.

As definições formais de cada tarefa serão apresentadas em seguida.

3.2 Definição formal do problema

3.2.1 Localização do bloco *try*

Esta tarefa consiste em localizar as instruções em um trecho de código que deve ser delimitado por blocos *try*. De acordo com J. Zhang et al. 2020 a definição formal é dada por:

[1] https://github.com/zhangj111/nexgen

> Dado o trecho de código $C = \{s_1, s_2, \ldots, s_k\}$ onde K é o número de declarações, o objetivo é encontrar uma sequência $Y = \{y_1, y_2, \ldots, y_k\}$, onde $y_i = Y$ ou N significa se a instrução s_i está em um bloco *try* ou não.

Por questões de simplificação, o autor adotou a estratégia de incluir todas as instruções adjacentes que lançam exceção em um único bloco *try*. Além disso, os pares *try-catch* aninhados são desconsiderados, pois se as instruções fossem incluídas em um único bloco *try*, o fluxo normal do código original seria afetado, já que a informação do escopo de cada exceção encontrada não estaria presente.

3.2.2 Geração do bloco *catch*

Esta tarefa tem como objetivo gerar os *tokens* do bloco *catch* para um dado bloco *try*. Formalmente, de acordo com J. Zhang et al. 2020:

> Suponha uma sequência de *tokens* $C = \{c_1, c_2, \ldots, c_l\}$ onde c_l $(i \in [1, l])$ significa um *token* no bloco *try* e o código-fonte antes dele, o objetivo é a geração da sequência de *tokens* dos blocos *catch* $Y = \{y_1, y_2, \ldots, y_l\}$ de tal modo que Y seja capaz de lidar com as exceções do bloco *try*. Y pode incluir vários blocos *catch* para diferentes tipos de exceções lançadas pelo bloco *try*.

Ou seja, a geração dos blocos *catch* é baseada na informação das instruções que lançam exceção no bloco *try* e de como suas dependências foram definidas no bloco principal do código.

O bloco *try* pode conter instruções que lançam diferentes tipos de exceção. Para lidar com esses tipos, a linguagem Java permite que vários blocos de tratamento de exceção específicos sejam criados.

Os detalhes da implementação dos dois modelos serão apresentados nas seções a seguir.

3.3 Arquitetura do modelo localizador de blocos *try*

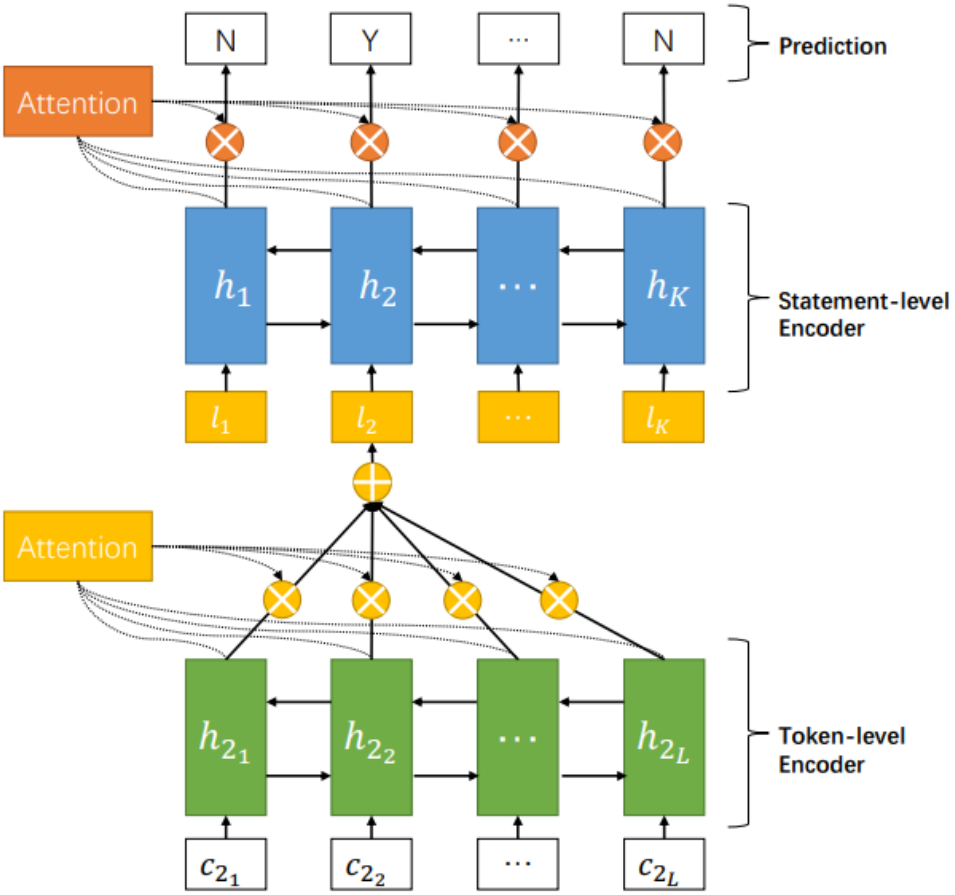
O localizador de blocos *try* implementa uma arquitetura semelhante ao modelo de atenção hierárquica apresentado no capítulo anterior 2.3, porém o modelo é focado na classificação das instruções, ou seja, a saída do modelo é centrada no módulo de atenção para a sentença (neste caso as instruções de código), pois é lá onde a classificação das instruções se concentra.

Para realizar a classificação de cada instrução como lançadora de exceção, o modelo inicialmente obtem a informação contextual de quais *tokens* são relevantes para a predição. Para isso, é aplicada a RNN com mecanismo de atenção na sequência de *tokens* do código, permitindo que o modelo faça uma busca por *tokens* relevantes para a predição.

Especificamente, na região inferior da Figura 3.1, destacada por *Token-level Encoder*, é descrita a RNN que codifica a sequência de *tokens* de uma matriz de incorporação (gerada através

do mapeamento dos *tokens* para números inteiros), ou seja, nessa região é obtida a informação contextual de cada *token* para as instruções de código. Como nem todos os *tokens* influenciam na predição, é descrita a região denotada por *Attention*. Essa região é responsável por obter a relevância que cada *token* possui para o lançamento da exceção. O mesmo princípio é aplicado para a região destacada por *Statement-level Encoder*, que tem como entrada a sequência de importâncias obtidas através da camada anterior (*Token-level Encoder* e *Attention*), mas neste caso o resultado desta camada é utilizado para classificar a instrução como lançadora de exceção ou não, denotado por *Prediction*.

Figura 3.1: A arquitetura do localizador de blocos *try*



Fonte: J. Zhang et al. 2020, página 2.

Cada região será descrita a seguir, começando a partir da região *Token-level encoder* até a região *Prediction*.

Atenção a nível de *token*

O módulo *WordAttention* é responsável por resumir a importância de cada *token* para o lançamento da exceção. Inicialmente, o módulo recebe a matriz de incorporação que é construída a partir da combinação do vocabulário com código.

```
1 line_limit=50
2 word_limit=20
3
4 # Geração do vocabulário
5 vocab = torchtext.vocab.Vocab(word_counter, max_size=vocab_size, min_freq=
      min_word_count)
6 PAD, UNK = vocab.stoi['<pad>'], vocab.stoi['<unk>']
7
8 # Geração da matriz de incorporação
9 encoded_train_docs = list(
10     map(lambda doc: list(
11         map(lambda s: list(map(
12             lambda w: vocab.stoi.get(w, UNK), s)
13         ) + [PAD] * (word_limit - len(s)), doc)) + [[PAD] * word_limit] * (
      line_limit - len(doc)),
14     train_docs)
15 )
16 lines_per_train_document = list(map(lambda doc: len(doc), train_docs))
17 words_per_train_line = list(
18     map(lambda doc: list(map(lambda s: len(s), doc)) + [0] * (line_limit -
      len(doc)), train_docs))
19 train_labels = list(
20     map(lambda y: y + [tmaps['<pad>']] * (line_limit - len(y)),
      train_labels))
```

Quadro 3.1: Função de *word embedding*

No trecho 3.1, o vocabulário (linha 5) é armazenado em um objeto que mapeia os *tokens* para números inteiros. A matriz de incorporação é construída iterando sobre cada *token* do trecho de código (linhas 9-15), resultando no vetor de incorporações c_i. Caso o *token* não esteja presente no vocabulário é convertido no *token* '<unk>'. O *token* '<pad>' representa os espaços vazios das dimensões fixadas para a quantidade máxima de linhas e *tokens*. A listagem a seguir exemplifica o resultado do processo.

```
Trecho da matriz de incorporação (20 x 50) e sua respectiva decodificação:
[[16, 19, 0, 2, 3, 31, 78, 9, ..., 1], ...]
-------------------------------------------------------------
[['public', 'void', '<unk>', '(', ')', 'throws', 'IOException',
    '{', '<pad>', ..., '<pad>'], ...]
```

Cada *token* incorporado (c_i) é codificado utilizando uma rede LSTM bidirecional, que obtêm seus estados ocultos com base nas entradas x_{i1} a x_{iL}.

$$\begin{aligned} \overrightarrow{h_{it}} &= \overrightarrow{LSTM}(h_{i_{t-1}}, x_{i_t}), t \in [1, L] \\ \overleftarrow{h_{it}} &= \overleftarrow{LSTM}(h_{i_{t+1}}, x_{i_t}), t \in [L, 1], \\ h_{it} &= [\overrightarrow{h_{it}}, \overleftarrow{h_{it}}], t \in [1, L] \end{aligned} \tag{3.1}$$

Na linha 7 do trecho 3.2, a matriz de incorporação é aplicada ao módulo Bi-LSTM para que seja obtido o resumo dos *tokens* que serão aplicados à camada de atenção.

```
1 sentences = self.dropout(self.embeddings(sentences))  # (n_sentences,
      word_pad_len, emb_size)
2 packed_words = pack_padded_sequence(sentences,
3                 lengths=words_per_sentence.tolist(),
4                 batch_first=True,
5                 enforce_sorted=False)
6
7 packed_words, _ = self.word_rnn(packed_words)
```

Quadro 3.2: Trecho do código de forward do módulo *WordAttention*

O trecho 3.3 define os módulos que serão utilizados nesta camada. Aqui são inicializados o módulo da matriz de incorporação, uma camada Bi-LSTM que serve para resumir a informação dos *tokens*, uma camada de atenção para os *tokens* e um módulo MLP para calcular o vetor de contexto:

```
1 # Inicialização
2 # Embeddings (look-up) layer
3 self.embeddings = nn.Embedding(vocab_size, emb_size)
4
5 # Bidirectional word-level RNN
6 self.word_rnn = nn.LSTM(emb_size, word_rnn_size, num_layers=word_rnn_layers
      , bidirectional=True, dropout=dropout, batch_first=True)
7
8 # Word-level attention network
9 self.word_attention = nn.Linear(2 * word_rnn_size, word_att_size)
10
11 # Word context vector to take dot-product with
12 self.word_context_vector = nn.Linear(word_att_size, 1, bias=False)
13
14 self.dropout = nn.Dropout(dropout)
```

Quadro 3.3: Inicialização do módulo *WordAttention*

O mecanismo de atenção é descrito na Equação 3.2, onde os estados da LSTM são aplicados a um *Multilayer perceptron* (MLP) de camada única para obter a codificação dos *tokens*. A função *softmax* é aplicada para gerar a_{it}, com o objetivo de calcular a influência de cada *token* na classificação da instrução. As pontuações para esses estados ocultos são obtidas medindo quão bem eles correspondem a um vetor fixo u_{ω}.

$$
\begin{aligned}
u_{i_t} &= tanh(W_\omega h_{i_t} + b_\omega), \\
\alpha_{i_t} &= \frac{exp(u_{i_t}^T u_\omega)}{\sum_{t=1}^{L} exp(u_{i_t}^T u_\omega)}, \\
s_i &= \sum_{t=1}^{L} \alpha_{i_t} h_{i_t}.
\end{aligned}
\tag{3.2}
$$

As linhas 1 e 2 do trecho 3.4, aplicam a camada MLP para obter o estado u_{it}. O mecanismo de atenção (linhas 4-15) é realizado através da aplicação da função *softmax* no estado u_{it}^T, com *max_value* sendo a instrução mais representativa.

```
1  att_w = self.word_attention(packed_words.data)  # (n_words, att_size)
2  att_w = torch.tanh(att_w)
3
4  att_w = self.word_context_vector(att_w).squeeze(1)  # (n_words)
5
6  max_value = att_w.max()  # scalar, for numerical stability during exponent
       calculation
7  att_w = torch.exp(att_w - max_value)  # (n_words)
8
9  att_w, _ = pad_packed_sequence(PackedSequence(data=att_w,
10     batch_sizes=packed_words.batch_sizes,
11     sorted_indices=packed_words.sorted_indices,
12     unsorted_indices=packed_words.unsorted_indices),
13     batch_first=True)
14
15 word_alphas = att_w / torch.sum(att_w, dim=1, keepdim=True)  # (n_sentences
       , max(words_per_sentence))
16
17 sentences, _ = pad_packed_sequence(packed_words,
18                                    batch_first=True)  # (n_sentences, max(
       words_per_sentence), 2 * word_rnn_size)
19
20 sentences = sentences * word_alphas.unsqueeze(2)  # (n_sentences, max(
       words_per_sentence), 2 * word_rnn_size)
21 sentences = sentences.sum(dim=1)  # (n_sentences, 2 * word_rnn_size)
```

Quadro 3.4: Trecho do código de forward da classe *WordAttention*

A saída desta camada é a soma ponderada do vetor de contexto com a instrução (linhas 20-21) representando a influência de cada *token* na classificação da instrução.

Atenção a nível de instrução

A partir do resultado anterior, o modelo é aplicado em outra Bi-LSTM para obter as dependências sequenciais das instruções, resultando nos estados ocultos $h_i = BiLSTM(s_i)$. No trecho 3.5,

é obtido o resultado do módulo *WordAttention* com as medidas de importância de cada *token* no lançamento da exceção. Na linha 8 é aplicada a camada Bi-LSTM para que seja obtido o resumo das instruções.

```
1 # Find sentence embeddings by applying the word-level attention module
2 sentences, word_alphas = self.word_attention(packed_sentences.data,
3   packed_words_per_sentence.data)  # (n_sentences, 2 * word_rnn_size), (
      n_sentences, max(words_per_sentence))
4
5 sentences = self.dropout(sentences)
6
7 # Apply the sentence-level RNN over the sentence embeddings (PyTorch
      automatically applies it on the PackedSequence)
8 packed_sentences, _ = self.sentence_rnn(PackedSequence(data=sentences,
9                          batch_sizes=packed_sentences.batch_sizes,
10                         sorted_indices=packed_sentences.sorted_indices,
11                         unsorted_indices=packed_sentences.unsorted_indices)
      )
12 documents, _ = pad_packed_sequence(packed_sentences, batch_first=True)
```

Quadro 3.5: Início do código de forward da classe *SentenceAttention*

No código, a classe *SentenceAttention* é responsável por implementar a camada *Statement-level Encoder*. A inicialização dessa classe, presente no trecho 3.6, define os módulos que serão utilizados nesta camada.

```
1 # Word-level attention module
2 self.word_attention = WordAttention(vocab_size, emb_size, word_rnn_size,
3   word_rnn_layers, word_att_size, dropout)
4
5 # Bidirectional sentence-level RNN
6 self.sentence_rnn = nn.LSTM(2 * word_rnn_size, sentence_rnn_size,
7   num_layers=sentence_rnn_layers, bidirectional=True,
8   dropout=dropout, batch_first=True)
9
10 # Sentence-level attention network
11 self.sentence_attention = nn.Linear(2 * sentence_rnn_size,
      sentence_rnn_size)
12
13 # Sentence context vector to take dot-product with
14 self.sentence_context_vector = nn.Linear(sentence_rnn_size, 1, bias=False)
15
16 # Dropout
17 self.dropout = nn.Dropout(dropout)
```

Quadro 3.6: Inicialização da classe *SentenceAttention*

Aqui são inicializadas a camada de atenção para os *tokens*, uma camada LSTM bidirecional

para resumir a informação das instruções, uma camada de atenção para as instruções e um módulo MLP para o vetor de contexto. O mecanismo de atenção para as instruções é dado pela 3.3, onde:

$$\begin{aligned} u_{i_t} &= tanh(W_s h_i + b_s), \\ \alpha_i &= \frac{exp(u_i^T u_s)}{\sum_{t=1}^{L} exp(u_i^T u_s)}, \\ h_i &= \alpha_i h_i. \end{aligned} \tag{3.3}$$

```
1 # Find attention vectors by applying the attention linear layer on the
      output of the RNN
2 att_s = self.sentence_attention(packed_sentences.data)  # (n_sentences,
      att_size)
3 att_s = torch.tanh(att_s)  # (n_sentences, att_size)
4 # Take the dot-product of the attention vectors with the context vector (i.
      e. parameter of linear layer)
5
6 att_s = self.sentence_context_vector(att_s).squeeze(1)  # (n_sentences)
7 max_value = att_s.max
8 att_s = torch.exp(att_s - max_value)
9 sentence_alphas = att_s / torch.sum(att_s, dim=1, keepdim=True)
10
11 # Similarly re-arrange sentence-level RNN outputs as documents by re-
      padding with 0s (SENTENCES -> DOCUMENTS)
12 documents, _ = pad_packed_sequence(packed_sentences,
13                                     batch_first=True)
14
15 # Find document embeddings (n_documents, max(sentences_per_document), 2 *
      sentence_rnn_size)
16 documents = documents * sentence_alphas.unsqueeze(2)
```

Quadro 3.7: Fim do código de forward da classe *SentenceAttention*

No trecho 3.7, o mecanismo de atenção é implementado aplicando o estado oculto obtido anteriormente a um *Multilayer perceptron* (MLP) para obter a codificação dos *tokens* (linhas 2 e 3). Em seguida, é aplicada a função *softmax* para gerar *sentence_alphas*, com o objetivo de resumir os estados ocultos e calcular a influência de cada *token*. As pontuações para esses estados ocultos são obtidas medindo quão bem eles correspondem a um vetor fixo *max_value* (linha 7) que é a instrução mais representativa. Por fim, vetor que representa o resumo das instruções é obtido na linha 16.

Módulo principal do modelo hierárquico

A classe *HierarchicalAttentionNetwork* representa a região mais abrangente do modelo hierárquico. No trecho 3.8, é possível observar que ela inicializa a camada de atenção para as instruções e uma camada MLP que serve para representar em um nível mais alto as *features* do código que serão submetidas à classificação, nas linhas 2 e 6 respectivamente.

```
1 # Sentence-level attention module (which will, in-turn, contain the word-
      level attention module)
2 self.sentence_attention = SentenceAttention(vocab_size, emb_size,
      word_rnn_size, sentence_rnn_size,
3   word_rnn_layers, sentence_rnn_layers, word_att_size, dropout)
4
5 # Classifier
6 self.fc = nn.Linear(2 * sentence_rnn_size, n_classes)
7
8 self.dropout = nn.Dropout(dropout)
```

Quadro 3.8: Inicialização do modelo Hierárquico

No trecho 3.9, a matriz de incorporação é aplicada ao módulo *SentenceAttention* (que posteriormente aplica ao módulo *WordAttention*), para obter as medidas de importância das instruções no lançamento da exceção.

```
1 document_embeddings = self.sentence_attention(documents,
      sentences_per_document, words_per_sentence)
2
3 # Classify
4 scores = self.fc(document_embeddings)  # (n_documents, n_classes)
5 scores = torch.sigmoid(scores)
```

Quadro 3.9: Forward do modelo Hierárquico

As instruções consecutivas que tiverem seus rótulos preditos com Y serão delimitadas por um mesmo bloco *try*. Desta forma, serão obtidas as localizações dos blocos *try*.

A probabilidade da instrução ser classificada como lançadora de exceção é calculada aplicando uma MLP sobre os estados ocultos h_i conforme a equação 3.4:

$$\hat{y_i} = sigmoid(W_p h_i + b_p) \in [0,1], \tag{3.4}$$

onde W_p é a matriz de pesos e b_p é o termo de viés da MLP.

A função de perda de entropia cruzada binária é utilizada para treinar o modelo:

$$\mathcal{L}(\Theta, \hat{y}, y) = \sum_{n=1}^{N} \sum_{i=1}^{L} (-(y_{ni} * log(\hat{y_{ni}}) + (1 - y_{ni}) * log(1 - \hat{y_{ni}}))) \tag{3.5}$$

onde y é a classificação de referência para cada instrução, Θ representa os parâmetros a serem aprendidos e N é o número total de instâncias no *dataset*. Se $\hat{y_i}$ for maior que um dado

limite a instrução lança exceção, caso contrário não lança.

3.4 Arquitetura do modelo gerador de blocos *catch*

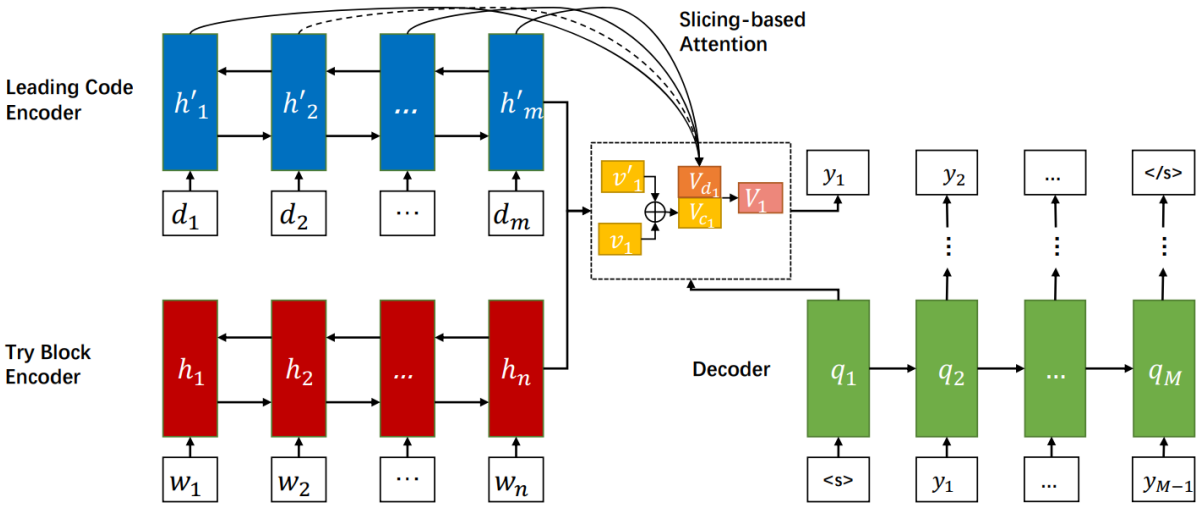
O gerador de blocos *catch* implementa a arquitetura de RNN *encoder-decoder*, proposta por Bahdanau, Cho e Bengio 2014, que consiste na codificação dos *tokens* utilizando uma RNN bidirecional e na decodificação utilizando uma RNN. O *encoder* utiliza um mecanismo de atenção que faz com que a decodificação funcione como uma busca no código com foco na informação relevante para a geração do próximo *token*.

Neste contexto, a entrada para o *encoder* é a sequência de tokens do bloco *try*, porém algumas instruções no código principal podem influenciar no lançamento da exceção. Por isso, é adicionado mais um *encoder* RNN bidirecional para capturar a informação contextual do código principal. O resultado são dois vetores de contexto que serão concatenados para formarem um único vetor que representa a influência de cada *token* na geração dos *tokens* do bloco *catch*.

Especificamente, na Figura 3.2, as regiões *Leading Code Encoder* (destacado em azul) e *Try Block Encoder* (destacado em vermelho) descrevem as RNNs bidirecionais do tipo LSTM utilizadas para codificar o código principal e o bloco *try*, respectivamente.

Em seguida, na primeira parte da região *Slicing-based Attention* são gerados os vetores de contexto v_i e v'_i do *encoder* do bloco *try* e do bloco *catch*, respectivamente. O resultado é o vetor de contexto V_{ci} que representa a relevância dos *tokens* na geração de um *token* do bloco *catch*.

Figura 3.2: A arquitetura do gerador de bloco *catch*



Fonte: J. Zhang et al. 2020, página 5.

Adicionalmente, é gerado um vetor de contexto V_{di} para filtrar instruções do código principal que são irrelevantes para a geração do bloco *catch*, como processamento de dados ou expressões condicionais. Por isso, a sequência de entrada do módulo denotado por *Leading Code Encoder*

é rotulada a partir da busca recursiva das instruções que podem influenciar no lançamento da exceção utilizando a técnica de *program slicing*.

A técnica de *program slicing*, descrita por Weiser 1984, consiste em obter uma redução aproximada das instruções do programa através da análise automática do fluxo de controle e de dados recursivamente.

Neste caso, a sequência de instruções do código principal são rotuladas conforme a equação 3.4, onde D' é o conjunto de instruções relevantes do código principal que serão combinados no mecanismo de atenção.

$$l_i = \begin{cases} 1 & \text{se } d_i \in D' \\ 0 & \text{caso contrário} \end{cases}$$

Conforme a equação, a partir dos rótulos gerados denotados por $L = l_1, l_2, ..., l_m$ as instruções que não são relevantes tem peso 0 e portanto serão filtradas na camada de atenção gerando o vetor de contexto V_{di}:

$$V_{di} = Attention(q_i, L \cdot H') \tag{3.6}$$

O vetor de contexto V_i, obtido através da concatenação dos vetores V_{di} e V_{ci} (3.2 *Slicing-based Attention*) é aplicado a uma camada única MLP emula uma consulta ao código focado na geração dos *tokens* do bloco *catch*.

Por fim, a região denotada por *Decoder* representa a RNN responsável por gerar os *tokens* do bloco *catch*. O objetivo é encontrar a probabilidade de gerar o *i*-ésimo *token* que irá compor o bloco *catch*, calculada por:

$$P(y_i \mid y_1, ..., y_{i-1}, C) = softmax(W_g V_i + b_g), \tag{3.7}$$

onde a equação $softmax(W_g V_i + b_g)$ é MLP aplicada sobre o vetor V_i, C representa as duas sequências de entrada do código principal e o bloco *try*.

O objetivo é minimizar a função *negative log-likelihood loss* dada por:

$$\mathcal{L}(\Theta) = -\sum_{i=1}^{N}\sum_{t=1}^{M} log(P(y_t^i \mid y_{<t} i, C)), \tag{3.8}$$

onde Θ são os parâmetros treináveis, N é o número total de instâncias de treinamento e M é o comprimento da instrução gerada.

4

Replicação da Abordagem

Este capítulo descreve a replicação da abordagem de tratamento de exceção automatizado utilizando o código-fonte e os conjuntos de dados originais disponibilizados por J. Zhang et al. 2020.

4.1 Configuração do ambiente

O ambiente utilizado para conduzir os experimentos foi a plataforma Google Colab utilizando a Unidade de Processamento Gráfico (GPU) disponível na versão gratuita. As configurações de hardware do ambiente são uma GPU NVIDIA Tesla T4 de 16GB de RAM e 68 GB de disco. Além disso, o Google Colab fornece um *runtime* configurado com Python 3.8, porém foi necessário instalar a versão Python 3.6 para utilizar as bibliotecas nas versões especificadas.

Para a execução do experimento, foi necessário instalar manualmente as bibliotecas *javalang, pandas, scikit-learn, torch, torchtext, tqdm* e *scipy* utilizando o gerenciador de dependências PIP. O código-fonte, disponibilizado pelo autor na plataforma Github[1], foi importado para a plataforma, bem como o conjunto de dados, disponibilizado pelo autor em uma pasta no Google Drive[2].

A composição original do conjunto de dados foi mantida, cujas estatísticas estão descritas na Tabela 4.1.

[1] https://github.com/zhangj111/nexgen

[2] https://drive.google.com/drive/folders/1tdg2DtvdqY8338g2pTzLQxnUd95nIU2S?usp=sharing

Tabela 4.1: Estatísticas do conjunto de dados
Fonte: Adaptado de Y. Zhang et al. 2018, página 6.

TBLD		**CBGD**	
#Métodos Java	755.846	**#Pares *try-catch***	351.420
#TryNum=1	341.040	**#CatchNum=1**	324.084
#TryNum≥2	36.883	**#CatchNum≥2**	27.336
MaxT	6403	**MaxT Principal**	3.313
AvgT	115,9	**AvgT Principal**	113,1
MaxS	99	**MaxT *catch***	365
AvgS	14,7	**AvgT *catch***	26,5
UniqT	566.378	**UniqT**	214.799

4.2 Avaliação do localizador de blocos *try*

Para a avaliação do desempenho na tarefa de localização de blocos *try* nos trechos de código do conjunto de dados de testes foram utilizadas as métricas de avaliação Precisão, Sensibilidade, *F1-score* e Acurácia. Seus valores são calculados da seguinte forma:

$$\begin{aligned} Acurácia &= \frac{TP+TN}{TP+TN+FP+FN} \\ Precisão &= \frac{TP}{TP+FP} \\ Recall &= \frac{TP}{TP+FN} \\ F1-score &= \frac{2*Precisão*Recall}{Precisão+Recall} = \frac{2*TP}{2*TP+FP+FN} \end{aligned} \tag{4.1}$$

As variáveis das equações acima são interpretadas da seguinte maneira:

1. O valor TP se refere as instruções que devem estar contidas em blocos *try* e foram classificadas corretamente;

2. O valor FP se refere as instruções que não devem estar contidas em blocos *try* e foram classificadas incorretamente;

3. O valor TN se refere as instruções que não devem estar contidas em blocos *try* e foram classificadas corretamente;

4. O valor FN se refere as instruções que devem estar contidas em blocos *try* e foram classificadas incorretamente.

$TP+FP$ corresponde ao total de classificações positivas do modelo e $TP+FN$ se refere as instruções do conjunto de dados que deveriam ser classificadas para serem incluídas no bloco

try. Por fim, a proporção geral de métodos em que todas as instruções são preditas corretamente é denotada por Acurácia.

Tabela 4.2: Métricas do Localizador de blocos *try*

	Acurácia	**Precisão**	**Sensibilidade**	***F1 - score***
Batch size = 64	74,1%	78,3%	76,3%	77,2%
Batch size = 32	73,2%	77,7%	75,8%	76,7%
Nexgen	74,7%	80,9%	74,3%	77,4%

Devido às diferenças nas especificações entre os ambientes de replicação e o descrito pelo autor, os resultados do treinamento do modelo foram diferentes dos apresentados no artigo. Além disso, o código-fonte foi configurado para utilizar o $batchsize = 64$, porém o artigo menciona $batchsize = 32$. Portanto, executamos o experimento utilizando os dois valores, onde é possível observar que o aumento do *batch size* resultou em um melhor desempenho, que mesmo assim, foi menor que o original.

A tabela 4.2 mostra que houve um ganho de 1,98% na sensibilidade e uma perda de 2,59% na precisão. A acurácia e o *F1 - score* reduziram em 0,56% e 0,12%, respectivamente.

4.3 Avaliação do gerador de blocos *catch*

Para a tarefa de gerador de blocos *catch* foi utilizada a métricas de avaliação *BLEU*, que é utilizada para avaliar tarefas de tradução automática. Basicamente o *BLEU score* avalia a distância entre a tradução feita automaticamente e a tradução realizada por um especialista. No caso do modelo apresentado, o $BLEUscore$ avalia o quão distantes do bloco *catch* original são os *tokens* gerados.

Especificamente, dado o código gerado Y' e o valor de referência Y, o *BLEU* mede a precisão dos n-gramas entre Y' e Y calculando a sobreposição dos n-gramas penalizando com um decaimento exponencial o código gerado que seja muito menor do que o tamanho do código de referência. O valor é calculado por:

$$BLEU = BP \cdot exp(\sum_{n=1}^{N} \omega_n log p_n) \tag{4.2}$$

$$BP = \begin{cases} 1 & \text{se } Y' > Y, \\ e^{(1-Y'/Y)} & \text{se } Y' \leq Y. \end{cases} \tag{4.3}$$

Da mesma forma, também foi calculada a razão dos blocos *catch* gerados que são exatamente iguais aos valores de referência, denotados por Acurácia. As métricas obtidas pelo treinamento do modelo são apresentadas na Tabela 4.3.

Tabela 4.3: Métricas do Gerador de blocos *catch*

	Acurácia	*BLEU*
Replicação	22,7%	46,9%
Nexgen	22,6%	46,7%

Novamente, os resultados do treinamento do modelo foram diferentes dos apresentados no artigo. A tabela 4.3 mostra que houve um ganho de 0,1% na Acurácia e um ganho de 0,2% no *BLEU score*.

5

Conclusão

Neste trabalho, replicamos a abordagem baseada em aprendizado profundo para tratamento automatizado de exceções, mas não conseguimos obter o mesmo desempenho relatado no artigo de J. Zhang et al. 2020. Para ambas as tarefas, foram implementados os modelos de redes neurais utilizando o ambiente Google Colab. Por fim, avaliamos as métricas dos modelos de cada uma das tarefas: localizar o bloco *try* e gerar o bloco *catch*.

Observamos que Acurácia e o *F1 - score* reduziram 0,56% e 0,12%, respectivamente devido a um aumento de 1,98% na sensibilidade e uma perda de 2,59% na precisão do modelo localizador de blocos *try*. Porém, o modelo gerador de blocos *catch* apresentou um ganho de 0,1% na Acurácia e de 0,2% no *BLEU score*. A divergência dos resultados se deram, provavelmente, pela diferença entre os ambientes e a configuração dos hiperparâmetros do modelo.

As implementações dos *notebooks* deste trabalho de replicação estão disponíveis em:

1. Localizador de blocos *try*[1]
2. Gerador de blocos *catch*[2]

[1] https://colab.research.google.com/drive/1Ij1La7j63nMucNXl6cApldkbL7NYZW2H?usp=sharing
[2] https://colab.research.google.com/drive/1Xw9vL09BTEMkQJ43Mj9lbRAAyTcphRGb?usp=sharing

Referências bibliográficas

Weiser, Mark (1984). “Program slicing”. Em: *IEEE Transactions on software engineering* 4, pp. 352–357.

Schuster, Mike e Kuldip K Paliwal (1997). “Bidirectional recurrent neural networks”. Em: *IEEE transactions on Signal Processing* 45.11, pp. 2673–2681.

Garcia, Alessandro F et al. (2001). “A comparative study of exception handling mechanisms for building dependable object-oriented software”. Em: *Journal of systems and software* 59.2, pp. 197–222.

Shah, Hina, Carsten Gorg e Mary Jean Harrold (2010). “Understanding exception handling: Viewpoints of novices and experts”. Em: *IEEE Transactions on Software Engineering* 36.2, pp. 150–161.

Bahdanau, Dzmitry, Kyunghyun Cho e Yoshua Bengio (2014). “Neural machine translation by jointly learning to align and translate”. Em: *arXiv preprint arXiv:1409.0473*.

Huang, Zhiheng, Wei Xu e Kai Yu (2015). “Bidirectional LSTM-CRF models for sequence tagging”. Em: *arXiv preprint arXiv:1508.01991*.

Yang, Zichao et al. (2016). “Hierarchical attention networks for document classification”. Em: *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pp. 1480–1489.

Yin, Wenpeng et al. (2017). “Comparative study of CNN and RNN for natural language processing”. Em: *arXiv preprint arXiv:1702.01923*.

Voets, Mike, Kajsa Møllersen e Lars Ailo Bongo (2018). “Replication study: Development and validation of deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs”. Em: *arXiv preprint arXiv:1803.04337*.

Zhang, Yuan et al. (2018). “Learning Tag Dependencies for Sequence Tagging.” Em: *IJCAI*, pp. 4581–4587.

Zhang, Jian et al. (2020). “Learning to handle exceptions”. Em: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 29–41.