ДОДАТОК А

Текст програми

Нижче наведено код файлу application.py, який включає в себе опис моделі данних та функції алгоритму.

```
# -*- coding: utf-8 -*-
from os import listdir
from os.path import isfile, join
import sys
import codecs
from lxml import etree
from lxml.html.clean import Cleaner
from io import StringIO, BytesIO

def enum(**enums):
    return type('Enum', (), enums)

# TODO:
'''
- many links - menu
- one link - title

- big GOOD_CANDIDATE-s -> GOOD
'''

# ====================== MODEL ====================
Classifier = enum(GOOD=0, BAD=1, GOOD_CANDIDATE=2, SHORT=3, NEYTRAL=4)

Type = enum(MAIN_CONTENT_BLOCK=0, CONTENT_BLOCK=1, TITLE=2, MENU=3, OTHER=4)
```

```
ignore_tags = ['script', 'meta', 'noscript', 'iframe', 'head', 'form', 'input', 'fieldset', 'select', 'style']
text_tags = ['p', 'b', 'i', 'span', 'strong', 'pre', 'blockquote', 'h1', 'h2', 'h3', 'h4', \
             'h5', 'h6', 'img', 'a', 'center', 'textarea']
content_text_tags = ['p', 'b', 'i', 'span', 'strong', 'pre', 'blockquote', 'img', 'center', 'textarea']
link_tags = ['img', 'a']
title_tags = ['h1', 'h2']

class TreeNode:
    def __init__(self):
        self.classifier = None          # Classifier
        self.final_classifier = None    # Classifier
        self.keywords = {}              # 'word':weight
        self.type = None                # Type
        self.lxmlNode = None            # lxml node instance
        self.tag = ""                   # html tag
        self.text = None                # html text
        self.childs = []                # TreeNode[]
        self.link_density = 0
        self.length = 0
        self.stopword_density = 0
        self.elementClass = None

class Chunk:
    def __init__(self):
        self.list = []
        self.size = 0
```

```
            self.count = 0
            self.classifier = None

# ====================== ALGO ====================

# <div style="border:2px double black">

def clear(parent, lxmlTree):

    for node in lxmlTree:
        node = find_valid_tag(node)
        if node == None:
            continue

        treeNode = TreeNode()
        treeNode.tag = node.tag
        treeNode.text = get_all_text(node)
        treeNode.lxmlNode = node
        treeNode.elementClass = node.get('class')

        clear(treeNode, node)

        parent.childs.append(treeNode)


def get_all_text(lxmlNode):
    text = ''
    if lxmlNode.text:
        text += lxmlNode.text.strip()
```

```
        for node in lxmlNode:
                if node.tail:
                        text += ' ' + node.tail.strip()
        return text
treeNode = TreeNode()
                treeNode.tag = node.tag
                treeNode.text = get_all_text(node)
                treeNode.lxmlNode = node
                treeNode.elementClass = node.get('class')

def classify_step_0(parent):

        parent.link_density = get_link_density(parent)
        parent.length = get_length(parent)

        # classify blocks
        title_count = 0
        for node in parent.childs:
                classify_step_0(node)
                if node.tag == 'a':
                        node.link_density = 1
                if node.link_density > 0.3:
                        node.classifier = Classifier.BAD
                else:
                        if node.length < 70 and node.tag not in ['h1', 'h2']:
                                #if node.link_density > 0:
                                #       node.classifier = Classifier.BAD
                                #else:
                                node.classifier = Classifier.SHORT
```

```
                else:
                    node.classifier = Classifier.GOOD_CANDIDATE


    def classify_step_1_merge_basic_blocks(parent):

        for node in parent.childs:
            classify_step_1_merge_basic_blocks(node)

        # merge blocks
        index = 0
        while index < len(parent.childs) - 1:
            curr_cl = parent.childs[index].classifier
            next_cl = parent.childs[index + 1].classifier
            curr_size = parent.childs[index].length
            next_size = parent.childs[index + 1].length
            final_classifier = merge_two_classifiers(curr_cl, next_cl,
curr_size, next_size)
            if final_classifier != None:
                parent.childs[index].classifier = final_classifier
                parent.childs[index + 1].classifier = final_classifier
            index += 1


    def classify_step_1_merge_set_of_basic_blocks(parent):

        for node in parent.childs:
            classify_step_1_merge_set_of_basic_blocks(node)
```

```
                    if len(node.childs) == 1:
                        continue
                    chunks = get_chunks(node)
                    index = 0

                    while index < len(chunks) - 1:
                        curr_cl = chunks[index].classifier
                        next_cl = chunks[index + 1].classifier
                        curr_size = chunks[index].size
                        next_size = chunks[index + 1].size
                        final_classifier = merge_two_classifiers_strong(curr_cl,
next_cl, curr_size, next_size)
                        if final_classifier != None:
                            chunks[index].classifier = final_classifier
                            chunks[index + 1].classifier = final_classifier
                        index += 1

                    index = 0
                    for chunk in chunks:
                        limit = index + chunk.count
                        while index < limit:
                            node.childs[index].classifier = chunk.classifier
                            index += 1


        def classify_step_2_title(parent):

            for node in parent.childs:
                # ...
```

```
                classify_step_2_title(node)
                if node.classifier == Classifier.GOOD_CANDIDATE and node.tag in title_tags:
                        node.classifier = Classifier.GOOD


    def classify_step_fill_good_childs(parent):

        for node in parent.childs:
            # ...
            if parent.classifier == Classifier.GOOD:
                node.classifier = Classifier.GOOD
            classify_step_fill_good_childs(node)


    def classify_step_temp(parent):

        good_count = 0
        for node in parent.childs:
            # ...
            classify_step_temp(node)
            if node.classifier in [Classifier.GOOD, Classifier.GOOD_CANDIDATE]:
                good_count += 1
        if good_count == 0 and parent.classifier == Classifier.GOOD_CANDIDATE:
            parent.classifier = Classifier.SHORT
```

```
def find_valid_tag(node):
        if node.tag in ignore_tags:
                return None
        if len(node):
                while len(node) == 1 and node.tag not in text_tags:
                        node = node[0]
                        if node.tag in ignore_tags:
                                return None
        return node


def get_link_density(tree):
        has_link = False
        links_length = 0
        for node in tree.childs:
                if node.tag == 'a':
                        has_link = True
                        links_length += get_link_size(node)
        if not has_link:
                return 0.0
        return float(links_length) / len(etree.tostring(tree.lxmlNode))


def get_link_size(link):
        length = 1
        if link.lxmlNode.text:
                length += len(link.lxmlNode.text.strip())
        for node in link.childs:
```

```
            if node.lxmlNode.tail:
                length += len(node.lxmlNode.tail.strip())
            length += len(etree.tostring(node.lxmlNode))
        return length


def get_length(tree):
    length = 1
    if tree.text:
        length += len(tree.text.strip())
    for node in tree.childs:
        if node.tag in content_text_tags and (node.text or node.length):
            if node.length:
                length += node.length
            else:
                length += len(node.text.strip())
    return length


def get_chunks(tree):
    chunks = []
    last_classifier = None
    chunk = None
    for node in tree.childs:
        if node.classifier != last_classifier:
            if chunk != None:
                chunks.append(chunk)
            chunk = Chunk()
            last_classifier = node.classifier
```

```
                    chunk.classifier = node.classifier
                chunk.size += node.length
                chunk.count += 1
            if chunk != None:
                chunks.append(chunk)
            return chunks


    def merge_two_classifiers(curr_cl, next_cl, curr_size, next_size):
        final_classifier = None
        if curr_cl == Classifier.BAD and next_cl == Classifier.BAD:
            final_classifier = Classifier.BAD
        if curr_cl == Classifier.SHORT and next_cl == Classifier.BAD and float(curr_size) / next_size < 0.7:
            final_classifier = Classifier.BAD
        if curr_cl == Classifier.BAD and next_cl == Classifier.SHORT and float(next_size) / curr_size < 0.7:
            final_classifier = Classifier.BAD
        if curr_cl == Classifier.SHORT and next_cl == Classifier.SHORT:
            final_classifier = Classifier.SHORT
        if curr_cl == Classifier.GOOD_CANDIDATE and next_cl == Classifier.SHORT and float(next_size) / curr_size < 0.7:
            final_classifier = Classifier.GOOD_CANDIDATE
        if curr_cl == Classifier.SHORT and next_cl == Classifier.GOOD_CANDIDATE and float(curr_size) / next_size < 0.7:
            final_classifier = Classifier.GOOD_CANDIDATE
        if curr_cl == Classifier.GOOD_CANDIDATE and next_cl == Classifier.GOOD_CANDIDATE:
            final_classifier = Classifier.GOOD
```

```
        if curr_cl == Classifier.GOOD_CANDIDATE and next_cl == Classifier.GOOD and float(curr_size) / next_size < 0.7:
            final_classifier = Classifier.GOOD
        if curr_cl == Classifier.GOOD and next_cl == Classifier.GOOD_CANDIDATE and float(next_size) / curr_size < 0.7:
            final_classifier = Classifier.GOOD
        if curr_cl == Classifier.SHORT and next_cl == Classifier.GOOD and float(curr_size) / next_size < 0.7:
            final_classifier = Classifier.GOOD
        if curr_cl == Classifier.GOOD and next_cl == Classifier.SHORT and float(next_size) / curr_size < 0.7:
            final_classifier = Classifier.GOOD
        return final_classifier


    def merge_two_classifiers_strong(curr_cl, next_cl, curr_size, next_size):
        final_classifier = None
        if curr_cl == Classifier.BAD and next_cl == Classifier.BAD:
            final_classifier = Classifier.BAD
        if curr_cl == Classifier.SHORT and next_cl == Classifier.BAD and float(curr_size) / next_size < 0.7:
            final_classifier = Classifier.BAD
        if curr_cl == Classifier.BAD and next_cl == Classifier.SHORT and float(next_size) / curr_size < 0.7:
            final_classifier = Classifier.BAD
        if curr_cl == Classifier.SHORT and next_cl == Classifier.SHORT:
            final_classifier = Classifier.SHORT
        if curr_cl == Classifier.GOOD_CANDIDATE and next_cl == Classifier.SHORT and float(next_size) / curr_size < 0.7:
```

```
                final_classifier = Classifier.GOOD_CANDIDATE
            if curr_cl == Classifier.SHORT and next_cl == Classifier.GOOD_CANDIDATE and float(curr_size) / next_size < 0.7:
                final_classifier = Classifier.GOOD_CANDIDATE
            if curr_cl == Classifier.GOOD_CANDIDATE and next_cl == Classifier.GOOD and float(curr_size) / next_size < 0.7:
                final_classifier = Classifier.GOOD
            if curr_cl == Classifier.GOOD and next_cl == Classifier.GOOD_CANDIDATE and float(next_size) / curr_size < 0.7:
                final_classifier = Classifier.GOOD
            if curr_cl == Classifier.SHORT and next_cl == Classifier.GOOD and float(curr_size) / next_size < 0.7:
                final_classifier = Classifier.GOOD
            if curr_cl == Classifier.GOOD and next_cl == Classifier.SHORT and float(next_size) / curr_size < 0.7:
                final_classifier = Classifier.GOOD

            if curr_cl == Classifier.GOOD_CANDIDATE and next_cl == Classifier.BAD and float(next_size) / curr_size < 0.3:
                final_classifier = Classifier.GOOD_CANDIDATE
            if curr_cl == Classifier.BAD and next_cl == Classifier.GOOD_CANDIDATE and float(curr_size) / next_size < 0.3:
                final_classifier = Classifier.GOOD_CANDIDATE

            return final_classifier
```

```
    def count_pricision_good(tree):
        '''
        x-nc-sel1, x-nc-sel2
        x-nc-sel0, x-nc-sel3, x-nc-sel4, x-nc-sel5
        '''
        good_count = 0
        for node in tree.childs:
            good_count += count_pricision_good(node)

        if tree.elementClass and tree.classifier in [Classifier.GOOD,
Classifier.GOOD_CANDIDATE]:
            if tree.elementClass in ['x-nc-sel1', 'x-nc-sel2', 'x-nc-sel5']:
                good_count += 1
        return good_count



    def count_found_good(tree):
        '''
        x-nc-sel1, x-nc-sel2
        x-nc-sel0, x-nc-sel3, x-nc-sel4, x-nc-sel5
        '''
        good_count = 0
        for node in tree.childs:
            good_count += count_found_good(node)

        if tree.elementClass and tree.classifier in [Classifier.GOOD,
Classifier.GOOD_CANDIDATE]:
```

```
                good_count += 1
        return good_count



def count_all_good(tree):
        '''
        x-nc-sel1, x-nc-sel2
        x-nc-sel0, x-nc-sel3, x-nc-sel4, x-nc-sel5
        '''
        good_count = 0
        for node in tree.childs:
                good_count += count_all_good(node)

        if tree.elementClass:
                if tree.elementClass in ['x-nc-sel1', 'x-nc-sel2', 'x-nc-sel5']:
                        good_count += 1
        return good_count



def to_str(tree):
        bgcolor = '#fff'
        if tree.classifier == Classifier.BAD:
                bgcolor = 'red'
        elif tree.classifier == Classifier.GOOD:
                bgcolor = 'green'
        elif tree.classifier == Classifier.GOOD_CANDIDATE:
                bgcolor = 'yellow'
```

```
            elif tree.classifier == Classifier.SHORT:
                bgcolor = 'gray'

            if tree.tag == 'a':
                #return etree.tostring(tree.lxmlNode)
                return '<a href="#" style="background: ' + bgcolor + '">LINK</a>'
            text = '<' + tree.tag + ' style="background: ' + bgcolor + '">'
            if tree.text:
                text += tree.text
            if len(tree.childs):
                for node in tree.childs:
                    text += to_str(node)
            return text + '</' + tree.tag + ">"


        def to_str_gr(tree):
            text = ''

            if tree.tag == 'a' and tree.classifier == Classifier.GOOD:
                return etree.tostring(tree.lxmlNode)

            if tree.tag == 'img' and tree.classifier == Classifier.GOOD:
                return etree.tostring(tree.lxmlNode)

            if tree.classifier == Classifier.GOOD:
                text = text + '<' + tree.tag + '>'
                if tree.text:
                    text += tree.text
```

```
        if len(tree.childs):
                for node in tree.childs:
                        text += to_str_gr(node)

        if tree.classifier == Classifier.GOOD:
                text += '</' + tree.tag + ">"

        return text

def exe():
        f = codecs.open(sys.argv[1], 'r', 'utf-8')
        html = f.read()
        f.close()
        parser = etree.HTMLParser()
        result = etree.parse(StringIO(html), parser)
        root = result.getroot()



        tree = TreeNode()
        tree.tag = root.tag
        tree.lxmlNode = root




        clear(tree, root)
        classify_step_0(tree)
        classify_step_1_merge_basic_blocks(tree)
        classify_step_1_merge_set_of_basic_blocks(tree)
        classify_step_2_title(tree)
```

```
        classify_step_fill_good_childs(tree)

        f = codecs.open('result.html', 'w', 'utf-8')
        if sys.argv[2] == '-highlight':
                f.write(to_str(tree))
                print 'The most important information is highlighted in document'
        elif sys.argv[2] == '-extract':
                f.write(to_str_gr(tree))
                print 'Information has been successfully extracted'
        f.close()

def main():
        exe()

if __name__ == '__main__':
        main()
```

ДОДАТОК Б

Відомість атестаційної роботи магістра

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	80с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Лизогуб Р.С.			Методи аналізу коду веб-сторінки з метою виявлення глибинних нейронних мереж.	Шифр групи	Код напр./спец.
Перевірив		Узлов Д.Ю.				СШІм-18-2	122
Н.контр.						ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					