|  |  |
| --- | --- |
| Подпись и дата |  |
| Инв. № дубл. |  |
| Взам. инв. № |  |
| Подпись и дата |  |
| Инв. № подл. | RU. 17701729. 503200-01 12 1-1ЛУ |

ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

|  |  |  |
| --- | --- | --- |
| СОГЛАСОВАНО  Доцент факультета компьютерных наук базовой кафедры «Системное программирование» НИУ ВШЭ, канд. физ.-мат. наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д.Ю. Турдаков  «\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2017 г. | УТВЕРЖДАЮ   |  | | --- | | Академический руководитель  Образовательной программы  «Программная инженерия» |   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.В.Шилов  «\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2017 г. |

**Построение иерархии аспектов по пользовательским отзывам об электронных устройствах**

**Текст программы**

**ЛИСТ УТВЕРЖДЕНИЯ**

RU. 17701729. 503200-01 12 1-1ЛУ

Исполнитель: студентка группы БПИ143

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/А.А.Репина/

«\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2017 г.

2017

|  |  |
| --- | --- |
| Подпись и дата |  |
| Инв. № дубл. |  |
| Взам. инв. № |  |
| Подпись и дата |  |
| Инв. № подл. | RU. 17701729. 503200-0112 |

**УТВЕРЖДЕНО**

**RU. 17701729. 503200-01 12 1-1-ЛУ**

|  |  |
| --- | --- |
|  |  |

**Построение иерархии аспектов по пользовательским отзывам об электронных устройствах**

**Текст программы**

RU. 17701729. 503200-0112 1-1

Листов 54

2017

Содержание

[Содержание 2](#_Toc478330689)

[1. Aspects.py 3](#_Toc478330690)

[2. Context.py 6](#_Toc478330691)

[3. DB.py 12](#_Toc478330692)

[4. FrequentAspects.py 21](#_Toc478330693)

[5. HierarchyBuilder.py 22](#_Toc478330694)

[6. IdealAspectsDB.py 24](#_Toc478330695)

[7. Lexical.py 26](#_Toc478330696)

[8. Main.py 26](#_Toc478330697)

[9. OneClassSVM.py 29](#_Toc478330698)

[10. PMI.py 30](#_Toc478330699)

[11. RandomForest.py 34](#_Toc478330700)

[12. SemanticDistanceLearning.py 36](#_Toc478330701)

[13. Sentence.py 41](#_Toc478330702)

[14. Splitter.py 43](#_Toc478330703)

[15. Syntactic.py 44](#_Toc478330704)

[16. Unnecessary.py 48](#_Toc478330705)

[17. CategoryNames.py 48](#_Toc478330706)

[18. DataBase\_Ulmart.py 49](#_Toc478330707)

[19. Parser.py 50](#_Toc478330708)

[ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ 53](#_Toc478330709)

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ 54](#_Toc478330711)

# Aspects.py

**import** json  
**import** requests  
  
**class Aspects**:  
 api\_key = **"43de6ee952010c5e0870b999f2a1949183456c73"** url\_syntatic\_parsing = \  
 **"http://api.ispras.ru/texterra/v3.1/nlp/syntax?filtering=KEEPING&class=syntax-relation&apikey="** url\_pos = **"http://api.ispras.ru/texterra/v3.1/nlp/pos?filtering=KEEPING&class=pos-token&apikey="** url\_sentence = **"http://api.ispras.ru/texterra/v3.1/nlp/sentence?filtering=KEEPING&class=sentence&apikey="  
  
 def** \_\_init\_\_(self):  
 self.url\_syntatic\_parsing += self.api\_key  
 self.url\_pos += self.api\_key  
 self.url\_sentence += self.api\_key  
  
 **def** process(self, aspect, db):  
 row\_aspect = db.cursor\_reviews.execute(**'SELECT** *\** **FROM Review'**).fetchone()  
 count = 0  
 **while** row\_aspect **is not None**: *# iterate through all reviews* print(count)  
 count += 1  
 article = str(row\_aspect[2])  
 adv = str(row\_aspect[3])  
 dis = str(row\_aspect[4])  
 com = str(row\_aspect[5])  
 **if** len(adv) != 0:  
 adv\_parsed = self.syntatic\_parsing(adv, aspect)  
 list\_adv\_aspects = self.aspects\_find(adv\_parsed, aspect) *# load aspects for advantage* **else**:  
 list\_adv\_aspects = []  
  
 **if** len(dis) != 0:  
 dis\_parsed = self.syntatic\_parsing(dis, aspect)  
 list\_dis\_aspects = self.aspects\_find(dis\_parsed, aspect) *# load aspects for disadvantage* **else**:  
 list\_dis\_aspects = []  
  
 **if** len(com) != 0:  
 com\_parsed = self.syntatic\_parsing(com, aspect)  
 list\_com\_aspects = self.aspects\_find(com\_parsed, aspect) *# load aspects for comment* **else**:  
 list\_com\_aspects = []  
 *# join the results* str\_adv\_aspects = **';'**.join(list\_adv\_aspects)  
 str\_dis\_aspects = **';'**.join(list\_dis\_aspects)  
 str\_com\_aspects = **';'**.join(list\_com\_aspects)  
 *# add found information to DB* db.add\_review(article, str\_adv\_aspects, str\_dis\_aspects, str\_com\_aspects)  
 row\_aspect = db.cursor\_reviews.fetchone()  
 db.conn\_aspects.commit()  
  
 @staticmethod  
 **def** syntatic\_parsing(review, aspect): *# detects syntactic structure for each sentence of a given text* **try**:  
 payload = {**'text'**: str(review)}  
 headers = {**'Accept'**: **'application/json'**}  
 r = requests.post(aspect.url\_syntatic\_parsing, data=payload, headers=headers)  
 **while** r.status\_code != 200:  
 r = requests.post(aspect.url\_syntatic\_parsing, data=payload, headers=headers)  
 **return** r.content.decode(**'utf8'**)  
 **except** Exception:  
 **import** sys  
 type, value, traceback = sys.exc\_info()  
 **return None  
  
 def** aspects\_find(self, part, aspect): *# find aspects in each review part* list\_aspects = [] *# list with aspects* print(part)  
 data = json.loads(part)  
 items = data[**'annotations'**][**'syntax-relation'**]  
 pos\_arr = self.parse\_pos(self.tag\_part\_of\_speech(data[**'text'**], aspect))  
 **for** item **in** items: *# iterate through words/word pairs in concrete review* **if 'parent' in** item[**'value'**]: *# word pair  
 # look for aspect noun(parent) + word* list\_aspects = self.word\_pair(data, item, list\_aspects, pos\_arr)  
 **else**: *# one word  
 # look for aspect noun(word)* list\_aspects = self.one\_word(data, item, list\_aspects, pos\_arr)  
 **return** list\_aspects  
  
 **def** one\_word(self, data, item, list\_aspects, pos\_arr):  
 start = item[**'start'**]  
 end = item[**'end'**]  
 word = data[**'text'**][start:end]  
 **if** pos\_arr[str(word)] == **'S'**: *# our word is noun* list\_aspects.append(self.replacer(word)) *# add an aspect noun(our word)* **return** list\_aspects  
  
 **def** word\_pair(self, data, item, list\_aspects, pos\_arr):  
 word = data[**'text'**][item[**'start'**]:item[**'end'**]]  
 parent = data[**'text'**][item[**'value'**][**'parent'**][**'start'**]:item[**'value'**][**'parent'**][**'end'**]]  
 pos\_parent = pos\_arr[str(parent)]  
 pos\_word = pos\_arr[str(word)]  
 **if** pos\_parent == **'S' and** pos\_word != **'PUNCT' and** pos\_word != **'CONJ' and** pos\_word != **'PR'**:  
 *# don't need punctuations, conjunctions, pretexts as words  
 # find pairs: S(parent) + smf* start\_par = item[**'value'**][**'parent'**][**'start'**]  
 start\_word = item[**'start'**]  
 **if** start\_par < start\_word: *# the word order is important in if-idf calculation  
 # add an aspect noun(parent) + our word* list\_aspects.append(self.replacer(parent) + **" "** + self.replacer(word))  
 **else**:  
 *# add an aspect noun(parent) + our word* list\_aspects.append(self.replacer(word) + **" "** + self.replacer(parent))  
 **elif** pos\_word == **'S'**:  
 list\_aspects.append(self.replacer(word)) *# add an aspect noun(our word)* **return** list\_aspects  
  
 @staticmethod  
 **def** replacer(item):  
 item = item.replace(**"\r"**, **" "**)  
 item = item.replace(**"\t"**, **" "**)  
 item = item.replace(**","**, **""**)  
 item = item.replace(**"."**, **""**)  
 item = item.replace(**"•"**, **""**)  
 item = item.replace(**";"**, **""**)  
 item = item.replace(**":"**, **""**)  
 item = item.replace(**"!"**, **""**)  
 item = item.replace(**"?"**, **""**)  
 item = item.replace(**")"**, **""**)  
 item = item.replace(**"("**, **""**)  
 item = item.replace(**"™"**, **""**)  
 item = item.replace(**"®"**, **""**)  
 item = item.replace(**"\*"**, **""**)  
 item = item.replace(**"\""**, **""**)  
 item = item.replace(**"—"**, **""**)  
 item = item.replace(**"-"**, **""**)  
 item = item.replace(**"~"**, **""**)  
 item = item.replace(**"'"**, **""**)  
 **return** item.lower()  
  
 @staticmethod  
 **def** parse\_pos(pos\_sentence):  
 sentence = json.loads(pos\_sentence)  
 pos\_items = sentence[**'annotations'**][**'pos-token'**]  
 result = {}  
 **for** item **in** pos\_items:  
 text = sentence[**'text'**][item[**'start'**]:item[**'end'**]]  
 result[text] = item[**'value'**][**'tag'**]  
 **return** result  
  
 @staticmethod  
 **def** tag\_part\_of\_speech(item, aspect): *# detects part of speech tag for each word of a given text* payload = {**'text'**: str(item)}  
 headers = {**'Accept'**: **'application/json'**}  
 r = requests.post(aspect.url\_pos, data=payload, headers=headers)  
 **while** r.status\_code != 200:  
 r = requests.post(aspect.url\_pos, data=payload, headers=headers)  
 **return** r.content.decode(**'utf8'**)  
  
 **def** move\_ideal\_aspects(self, ideal, ideal\_aspects, db):  
 row\_aspect = db.cursor\_aspects.execute(**'SELECT** *\** **FROM Aspects'**).fetchone()  
 count = 0  
 **while** row\_aspect **is not None**: *# iterate through all reviews* print(count)  
 count += 1  
 article = str(row\_aspect[0])  
 adv = str(row\_aspect[1])  
 dis = str(row\_aspect[2])  
 com = str(row\_aspect[3])  
 ideal\_adv = self.get\_ideal(adv, ideal\_aspects)  
 ideal\_dis = self.get\_ideal(dis, ideal\_aspects)  
 ideal\_com = self.get\_ideal(com, ideal\_aspects)  
 *# join the results* str\_adv\_aspects = **';'**.join(ideal\_adv)  
 str\_dis\_aspects = **';'**.join(ideal\_dis)  
 str\_com\_aspects = **';'**.join(ideal\_com)  
 ideal.add\_review(article, str\_adv\_aspects, str\_dis\_aspects, str\_com\_aspects)  
 row\_aspect = db.cursor\_aspects.fetchone()  
 ideal.conn\_aspects.commit()  
  
 @staticmethod  
 **def** get\_ideal(part, ideal\_aspects):  
 aspect\_arr = []  
 **if** len(part) != 0:  
 splitted\_part = part.split(**";"**)  
 **for** item **in** splitted\_part:  
 **if** item **in** ideal\_aspects:  
 aspect\_arr.append(item)  
 **return** aspect\_arr

# Context.py

**import** codecs  
**from** datetime **import** datetime  
**from** scipy **import** stats  
**import** numpy **as** np  
**from** sklearn.feature\_extraction.text **import** CountVectorizer  
  
**class Context**:  
 **def** process(self, db, aspects):  
 all\_aspects\_words = {}  
 reviews = self.get\_reviews\_and\_vocabulary(db, all\_aspects\_words) *# get user reviews* db.create\_context\_local\_prepare\_db()  
 db.create\_context\_global\_prepare\_db()  
 *# fill the db where the aspects with 4-words substrs as context their context are were calculated* self.form\_local\_context\_db(db, aspects, reviews, 0)  
 self.form\_global\_context\_db(db, aspects, reviews, 0)  
 self.form\_global\_context\_extra\_db(db, aspects)  
 print(**"started"**)  
 self.local\_context(db, all\_aspects\_words) *# calculate the local context* print(**"local finished"**)  
 self.global\_context(db, all\_aspects\_words) *# calculate the global context* print(**"global finished"**)  
 *# In both calculations we build language model for each aspect, then we calculate the KL - divergence  
 # for every language model combination. The difference between global and local contexts is that in global  
 # we take words from all the reviews and in local we consider only the words of concrete review* **def** process\_ideal(self, db):  
 aspects = self.get\_ideal\_dict()  
 all\_aspects\_words = {}  
 reviews = self.get\_reviews\_and\_vocabulary(db, all\_aspects\_words) *# get user reviews* db.create\_context\_local\_prepare\_ideal\_db()  
 db.create\_context\_global\_prepare\_ideal\_db()  
 *# fill the db where the aspects with 4-words substrs as context their context are were calculated* self.form\_local\_context\_db(db, aspects, reviews, 1)  
 self.form\_global\_context\_db(db, aspects, reviews, 1)  
 self.form\_global\_context\_extra\_ideal\_db(db, aspects)  
 print(**"started"**)  
 self.local\_context\_ideal(db, all\_aspects\_words) *# calculate the local context* print(**"local finished"**)  
 self.global\_context\_ideal(db, all\_aspects\_words) *# calculate the global context* print(**"global finished"**)  
 *# In both calculations we build language model for each aspect, then we calculate the KL - divergence  
 # for every language model combination. The difference between global and local contexts is that in global  
 # we take words from all the reviews and in local we consider only the words of concrete review* **def** get\_ideal\_dict(self):  
 **import** os  
 dict = {}  
 path = os.getcwd()  
 filenames = os.listdir(path + **"/../productTrees/Subcategories"**)  
 os.chdir(path + **"/../productTrees/Subcategories"**)  
 filenames.remove(**".DS\_Store"**)  
 filenames.remove(**"Subcategories.txt"**)  
 count = 0  
 **for** filename **in** filenames:  
 line = codecs.open(filename, **'r'**, **'cp1251'**).readlines()[0]  
 words = line.split(**";"**)  
 **for** word **in** words:  
 low\_word = word.lower().replace(**","**, **" "**).replace(**" "**, **" "**).replace(**" "**, **"\_"**)  
 *# клавиатуры!!!! мыши и клавиатуры* **if** low\_word **not in** dict:  
 dict[low\_word] = count  
 count += 1  
 **return** dict  
  
 **def** get\_reviews\_and\_vocabulary(self, db, all\_aspects\_words):  
 reviews = []  
 row\_review = db.cursor\_reviews.execute(**'SELECT** *\** **FROM Review'**).fetchone()  
 count = 0  
 **while** row\_review **is not None**:  
 adv = str(row\_review[3])  
 dis = str(row\_review[4])  
 com = str(row\_review[5])  
 review = adv + **" "** + dis + **" "** + com  
 review = self.replacer(review)  
 review = **" "**.join(review.split())  
 reviews.append(review)  
 words = review.split(**" "**)  
 **for** word **in** words:  
 **if** word **not in** all\_aspects\_words:  
 all\_aspects\_words[word] = count  
 count += 1  
 row\_review = db.cursor\_reviews.fetchone()  
 **return** reviews  
  
 **def** form\_local\_context\_db(self, db, aspects, reviews, which\_part):  
 count = 0  
 **for** aspect **in** aspects:  
 print(count)  
 clear\_aspect = aspect.lower().replace(**"\_"**, **" "**)  
 str\_context = **""  
 for** review **in** reviews:  
 clear\_aspect\_words = clear\_aspect.split(**" "**)  
 **if** len(clear\_aspect\_words) == 1: *# the aspect is only 1 word* str\_context = self.is\_one\_word\_aspect\_in\_review(clear\_aspect, review, str\_context, **False**)  
 **else**: *# the aspect consists of several words* str\_context = self.is\_several\_word\_aspect\_in\_review(clear\_aspect\_words, review, str\_context, **False**)  
 **if** which\_part == 0:  
 db.add\_context\_local\_prepare(aspect, str\_context) *# collect the 4 words context for each aspect occurrence* db.conn\_local\_context\_prepare.commit()  
 **else**: *# ideal aspects* db.add\_context\_local\_prepare\_ideal(aspect, str\_context) *# collect the 4 words context for each aspect occurrence* db.conn\_local\_context\_prepare\_ideal.commit()  
 count += 1  
  
 **def** form\_global\_context\_db(self, db, aspects, reviews, which\_part):  
 count = 0  
 **for** aspect **in** aspects:  
 print(count)  
 clear\_aspect = aspect.lower().replace(**"\_"**, **" "**)  
 str\_context = **""  
 for** review **in** reviews:  
 clear\_aspect\_words = clear\_aspect.split(**" "**)  
 **if** len(clear\_aspect\_words) == 1: *# the aspect is only 1 word* is\_aspect\_in\_review = self.is\_one\_word\_aspect\_in\_review(clear\_aspect, review, str\_context, **True**)  
 **else**: *# the aspect consists of several words* is\_aspect\_in\_review = self.is\_several\_word\_aspect\_in\_review(clear\_aspect\_words, review, str\_context,  
 **True**)  
 **if** is\_aspect\_in\_review: *# collect all the reviews with the aspect* **if** which\_part == 0:  
 db.add\_context\_global\_prepare(aspect, review)  
 **else**:  
 db.add\_context\_global\_prepare\_ideal(aspect, review)  
 **if** which\_part == 0:  
 db.conn\_global\_context\_prepare.commit()  
 **else**:  
 db.conn\_global\_context\_prepare\_ideal.commit()  
 count += 1  
  
 @staticmethod  
 **def** form\_global\_context\_extra\_db(db, aspects):  
 db.create\_context\_global\_prepare\_extra\_db()  
 count = 0  
 **for** aspect **in** aspects:  
 print(count)  
 aspect\_row = db.cursor\_global\_context\_prepare.execute(**'SELECT** *\** **FROM Context WHERE aspect = ?'**,  
 (aspect,)).fetchone()  
 context = **""  
 while** aspect\_row **is not None**:  
 context += str(aspect\_row[1]) + **" "** aspect\_row = db.cursor\_global\_context\_prepare.fetchone()  
 db.add\_context\_global\_prepare\_extra(aspect, context)  
 db.conn\_global\_context\_prepare\_extra.commit()  
 count += 1  
  
 @staticmethod  
 **def** form\_global\_context\_extra\_ideal\_db(db, aspects):  
 db.create\_context\_global\_prepare\_extra\_ideal\_db()  
 count = 0  
 **for** aspect **in** aspects:  
 print(count)  
 aspect\_row = db.cursor\_global\_context\_prepare\_ideal.execute(**'SELECT** *\** **FROM Context WHERE aspect = ?'**,  
 (aspect,)).fetchone()  
 context = **""  
 while** aspect\_row **is not None**:  
 context += str(aspect\_row[1]) + **" "** aspect\_row = db.cursor\_global\_context\_prepare\_ideal.fetchone()  
 db.add\_context\_global\_prepare\_extra\_ideal(aspect, context)  
 db.conn\_global\_context\_prepare\_extra\_ideal.commit()  
 count += 1  
  
 **def** is\_several\_word\_aspect\_in\_review(self, clear\_aspect\_words, review, str\_context, is\_global):  
 is\_all\_aspect\_words\_in\_review = **True  
 for** word **in** clear\_aspect\_words:  
 **if** word **not in** review.split():  
 is\_all\_aspect\_words\_in\_review = **False  
 break  
 if** is\_global:  
 **return** is\_all\_aspect\_words\_in\_review  
 **if** is\_all\_aspect\_words\_in\_review:  
 *# if aspect parts are held in different places of review take the left and the right word* left\_aspect\_part = clear\_aspect\_words[0]  
 right\_aspect\_part = clear\_aspect\_words[len(clear\_aspect\_words) - 1]  
 words = review.split(**' '**)  
 left\_aspect\_part\_index = np.where(np.array(words) == left\_aspect\_part)[0][0]  
 right\_aspect\_part\_index = np.where(np.array(words) == right\_aspect\_part)[0][0]  
 left = self.check\_left\_index(left\_aspect\_part\_index, words)  
 right = self.check\_right\_index(right\_aspect\_part\_index, words)  
 **if** len(str\_context) > 0:  
 str\_context += **" "** str\_context += left + **" "** + right  
 **return** str\_context  
  
 **def** is\_one\_word\_aspect\_in\_review(self, aspect, review, str\_context, is\_global):  
 **if** aspect **in** review.split():  
 **if** is\_global:  
 **return True** *# try to find every 2 left and 2 right words for aspect* words = review.split(**' '**)  
 aspect\_indexes = np.where(np.array(words) == aspect)[0]  
 *# find 2 left and 2 right word for each aspect occurrence* **for** index **in** aspect\_indexes:  
 str\_context = self.form\_str\_context(index, words, str\_context)  
 **if** is\_global:  
 **return False  
 return** str\_context  
  
 **def** form\_str\_context(self, index, words, str\_context):  
 *# if there is no 2 left or no 2 right words make their str as \_BEGIN\_SENTENCE\_ and \_END\_SENTENCE\_* left = self.check\_left\_index(index, words)  
 right = self.check\_right\_index(index, words)  
 **if** len(str\_context) > 0:  
 str\_context += **" "** str\_context += left + **" "** + right  
 **return** str\_context  
  
 **def** check\_left\_index(self, index, words):  
 **if** index - 1 < 0:  
 left\_1 = **"\_BEGIN\_SENTENCE\_"  
 else**:  
 left\_1 = self.replacer(words[index - 1])  
 **if** index - 2 < 0:  
 left\_2 = **"\_BEGIN\_SENTENCE\_"  
 else**:  
 left\_2 = self.replacer(words[index - 2])  
 **return** left\_2 + **" "** + left\_1  
  
 **def** check\_right\_index(self, index, words):  
 **if** index + 1 > len(words) - 1:  
 right\_1 = **"\_END\_SENTENCE\_"  
 else**:  
 right\_1 = self.replacer(words[index + 1])  
 **if** index + 2 > len(words) - 1:  
 right\_2 = **"\_END\_SENTENCE\_"  
 else**:  
 right\_2 = self.replacer(words[index + 2])  
 **return** right\_1 + **" "** + right\_2  
  
 @staticmethod  
 **def** replacer(item):  
 item = item.replace(**"\r"**, **" "**)  
 item = item.replace(**"\t"**, **" "**)  
 item = item.replace(**","**, **""**)  
 item = item.replace(**"."**, **""**)  
 item = item.replace(**"•"**, **""**)  
 item = item.replace(**";"**, **""**)  
 item = item.replace(**":"**, **""**)  
 item = item.replace(**"!"**, **""**)  
 item = item.replace(**"?"**, **""**)  
 *# item = item.replace(")", "")  
 # item = item.replace("(", "")* item = item.replace(**"™"**, **""**)  
 item = item.replace(**"®"**, **""**)  
 item = item.replace(**"\*"**, **""**)  
 item = item.replace(**"\""**, **""**)  
 *# item = item.replace("—", "")  
 # item = item.replace("-", "")* item = item.replace(**"~"**, **""**)  
 item = item.replace(**"'"**, **""**)  
 **return** item.lower()  
  
 **def** local\_context(self, db, all\_aspects\_words):  
 context\_for\_aspects\_dict = {}  
 db.create\_context\_local\_db()  
 vectorizer = CountVectorizer(ngram\_range=(1, 1), vocabulary=all\_aspects\_words)  
 aspect\_row = db.cursor\_local\_context\_prepare.execute(**'SELECT** *\** **FROM Context'**).fetchone()  
 *# load all the data from context db to context\_for\_aspects\_dict* count = 0  
 ngram\_prepared\_dict = {}  
 **while** aspect\_row **is not None**:  
 aspect = str(aspect\_row[0])  
 context = str(aspect\_row[1])  
 context\_for\_aspects\_dict[count] = np.array([aspect, context])  
 ngram = self.add\_one\_smoothing(vectorizer.fit\_transform([context]).toarray()[0]) *# get ngram* divider = len(context.split())  
 **if** divider != 0:  
 ngram = [x / divider **for** x **in** ngram]  
 ngram\_prepared\_dict[aspect] = ngram  
 aspect\_row = db.cursor\_local\_context\_prepare.fetchone()  
 count += 1  
 print(**"local rows loaded"**)  
 *# look through all aspect pairs to calculate their kl\_divergence  
 # the strs with many 4-words substrs which were calculated in form\_context\_db method for each aspect* **for** i **in** range(len(context\_for\_aspects\_dict)):  
 print(i)  
 start = datetime.now()  
 aspect1 = context\_for\_aspects\_dict[i][0]  
 ngram1 = ngram\_prepared\_dict[aspect1]  
 **for** j **in** range(i + 1, len(context\_for\_aspects\_dict)):  
 aspect2 = context\_for\_aspects\_dict[j][0]  
 ngram2 = ngram\_prepared\_dict[aspect2]  
 *# calculate the kl-divergence for local context* kl\_diver = stats.entropy(np.array(ngram1), np.array(ngram2), 2) *# send 2 unigram language models in vector form* db.add\_context\_local(aspect1, aspect2, kl\_diver)  
 db.conn\_local\_context.commit()  
 print(datetime.now() - start)  
  
 **def** local\_context\_ideal(self, db, all\_aspects\_words):  
 context\_for\_aspects\_dict = {}  
 db.create\_context\_local\_ideal\_db()  
 vectorizer = CountVectorizer(ngram\_range=(1, 1), vocabulary=all\_aspects\_words)  
 aspect\_row = db.cursor\_local\_context\_prepare\_ideal.execute(**'SELECT** *\** **FROM Context'**).fetchone()  
 *# load all the data from context db to context\_for\_aspects\_dict* count = 0  
 ngram\_prepared\_dict = {}  
 **while** aspect\_row **is not None**:  
 aspect = str(aspect\_row[0])  
 context = str(aspect\_row[1])  
 context\_for\_aspects\_dict[count] = np.array([aspect, context])  
 ngram = self.add\_one\_smoothing(vectorizer.fit\_transform([context]).toarray()[0]) *# get ngram* divider = len(context.split())  
 **if** divider != 0:  
 ngram = [x / divider **for** x **in** ngram]  
 ngram\_prepared\_dict[aspect] = ngram  
 aspect\_row = db.cursor\_local\_context\_prepare\_ideal.fetchone()  
 count += 1  
 print(**"local rows loaded"**)  
 *# look through all aspect pairs to calculate their kl\_divergence  
 # the strs with many 4-words substrs which were calculated in form\_context\_db method for each aspect* **for** i **in** range(len(context\_for\_aspects\_dict)):  
 print(i)  
 start = datetime.now()  
 aspect1 = context\_for\_aspects\_dict[i][0]  
 ngram1 = ngram\_prepared\_dict[aspect1]  
 **for** j **in** range(i + 1, len(context\_for\_aspects\_dict)):  
 aspect2 = context\_for\_aspects\_dict[j][0]  
 ngram2 = ngram\_prepared\_dict[aspect2]  
 *# calculate the kl-divergence for local context* kl\_diver = stats.entropy(np.array(ngram1), np.array(ngram2),  
 2) *# send 2 unigram language models in vector form* db.add\_context\_local\_ideal(aspect1, aspect2, kl\_diver)  
 db.conn\_local\_context\_ideal.commit()  
 print(datetime.now() - start)  
  
 **def** global\_context(self, db, all\_aspects\_words):  
 count = 0  
 context\_for\_aspects\_dict = {}  
 db.create\_context\_global\_db()  
 vectorizer = CountVectorizer(ngram\_range=(1, 1), vocabulary=all\_aspects\_words)  
 *# load all the data from context db* aspect\_row = db.cursor\_global\_context\_prepare\_extra.execute(**'SELECT** *\** **FROM Context'**).fetchone()  
 ngram\_prepared\_dict = {}  
 **while** aspect\_row **is not None**:  
 aspect = str(aspect\_row[0])  
 context = str(aspect\_row[1])  
 context\_for\_aspects\_dict[count] = [aspect, context]  
 ngram = self.add\_one\_smoothing(vectorizer.fit\_transform([context]).toarray()[0]) *# get ngram* divider = len(context.split())  
 **if** divider != 0:  
 ngram = [x / divider **for** x **in** ngram]  
 ngram\_prepared\_dict[aspect] = ngram  
 aspect\_row = db.cursor\_global\_context\_prepare\_extra.fetchone()  
 count += 1  
 print(**"global rows loaded"**)  
 *# look through all aspect pairs to calculate their kl\_divergence* **for** i **in** range(len(context\_for\_aspects\_dict)):  
 print(i)  
 start = datetime.now()  
 aspect1 = context\_for\_aspects\_dict[i][0]  
 ngram1 = ngram\_prepared\_dict[aspect1]  
 **for** j **in** range(i + 1, len(context\_for\_aspects\_dict)):  
 aspect2 = context\_for\_aspects\_dict[j][0]  
 ngram2 = ngram\_prepared\_dict[aspect2]  
 *# calculate the kl-divergence for global context* kl\_diver = stats.entropy(np.array(ngram1), np.array(ngram2), 2) *# send 2 unigram language models in vector form* db.add\_context\_global(aspect1, aspect2, kl\_diver)  
 db.conn\_global\_context.commit()  
 print(datetime.now() - start)  
  
 **def** global\_context\_ideal(self, db, all\_aspects\_words):  
 count = 0  
 context\_for\_aspects\_dict = {}  
 db.create\_context\_global\_ideal\_db()  
 vectorizer = CountVectorizer(ngram\_range=(1, 1), vocabulary=all\_aspects\_words)  
 *# load all the data from context db* aspect\_row = db.cursor\_global\_context\_prepare\_extra\_ideal.execute(**'SELECT** *\** **FROM Context'**).fetchone()  
 ngram\_prepared\_dict = {}  
 **while** aspect\_row **is not None**:  
 aspect = str(aspect\_row[0])  
 context = str(aspect\_row[1])  
 context\_for\_aspects\_dict[count] = [aspect, context]  
 ngram = self.add\_one\_smoothing(vectorizer.fit\_transform([context]).toarray()[0]) *# get ngram* divider = len(context.split())  
 **if** divider != 0:  
 ngram = [x / divider **for** x **in** ngram]  
 ngram\_prepared\_dict[aspect] = ngram  
 aspect\_row = db.cursor\_global\_context\_prepare\_extra\_ideal.fetchone()  
 count += 1  
 print(**"global rows loaded"**)  
 *# look through all aspect pairs to calculate their kl\_divergence* **for** i **in** range(len(context\_for\_aspects\_dict)):  
 print(i)  
 start = datetime.now()  
 aspect1 = context\_for\_aspects\_dict[i][0]  
 ngram1 = ngram\_prepared\_dict[aspect1]  
 **for** j **in** range(i + 1, len(context\_for\_aspects\_dict)):  
 aspect2 = context\_for\_aspects\_dict[j][0]  
 ngram2 = ngram\_prepared\_dict[aspect2]  
 *# calculate the kl-divergence for global context* kl\_diver = stats.entropy(np.array(ngram1), np.array(ngram2), 2) *# send 2 unigram language models in vector form* db.add\_context\_global\_ideal(aspect1, aspect2, kl\_diver)  
 db.conn\_global\_context\_ideal.commit()  
 print(datetime.now() - start)  
  
 @staticmethod  
 **def** add\_one\_smoothing(list):  
 **for** i **in** range(len(list)):  
 **if** list[i] == 0:  
 list[i] = 1  
 **else**:  
 list[i] += 1  
 **return** list

# DB.py

**import** sqlite3  
**import** os  
  
**class** DB:  
 conn\_aspects = **None** conn\_reviews = **None** conn\_merged = **None** conn\_sentence = **None** conn\_aspects\_one\_word = **None** conn\_reviews\_one\_word = **None** conn\_sentences\_one\_word = **None** conn\_pmi\_review = **None** conn\_pmi\_sentence = **None** conn\_pmi\_ideal\_review = **None** conn\_pmi\_ideal\_sentence = **None** conn\_path\_weight = **None** conn\_semantic\_distance = **None** conn\_semantic\_distance\_ideal = **None** conn\_local\_context\_prepare = **None** conn\_local\_context\_prepare\_ideal = **None** conn\_global\_context\_prepare = **None** conn\_global\_context\_prepare\_extra = **None** conn\_global\_context\_prepare\_ideal = **None** conn\_global\_context\_prepare\_extra\_ideal = **None** conn\_global\_context = **None** conn\_local\_context = **None** conn\_lexical = **None** conn\_lexical\_ideal = **None** conn\_syntactic = **None** conn\_syntactic\_ideal = **None** conn\_tree = **None** conn\_local\_context\_ideal = **None** conn\_global\_context\_ideal = **None** conn\_hierarchy = **None** conn\_frequent = **None** conn\_ideal\_full = **None** conn\_hierarchy\_real = **None** conn\_semantic\_distance\_real = **None** cursor\_aspects = **None** cursor\_aspects2 = **None** cursor\_reviews = **None** cursor\_article = **None** cursor\_merged = **None** cursor\_sentence = **None** cursor\_aspects\_one\_word = **None** cursor\_reviews\_one\_word = **None** cursor\_reviews\_one\_word\_update = **None** cursor\_sentences\_one\_word = **None** cursor\_sentences\_one\_word\_update = **None** cursor\_pmi\_review = **None** cursor\_pmi\_sentence = **None** cursor\_pmi\_ideal\_review = **None** cursor\_pmi\_ideal\_sentence = **None** cursor\_path\_weight = **None** cursor\_semantic\_distance = **None** cursor\_semantic\_distance\_ideal = **None** cursor\_local\_context\_prepare = **None** cursor\_local\_context\_prepare\_ideal = **None** cursor\_global\_context\_prepare = **None** cursor\_global\_context\_prepare\_extra = **None** cursor\_global\_context\_prepare\_ideal = **None** cursor\_global\_context\_prepare\_extra\_ideal = **None** cursor\_global\_context = **None** cursor\_local\_context\_ideal = **None** cursor\_global\_context\_ideal = **None** cursor\_local\_context = **None** cursor\_lexical = **None** cursor\_lexical\_ideal = **None** cursor\_syntactic = **None** cursor\_syntactic\_ideal = **None** cursor\_tree = **None** cursor\_hierarchy = **None** cursor\_frequent = **None** cursor\_ideal\_full = **None** cursor\_hierarchy\_real = **None** cursor\_semantic\_distance\_real = **None** db\_merged\_name = **'Merged.db'** db\_aspects\_name = **'Aspects.db'** db\_reviews\_name = **'Reviews.db'** db\_sentence\_name = **'Sentence.db'** db\_aspects\_one\_word\_name = **'Aspects\_One\_Word.db'** db\_reviews\_one\_word\_name = **'Reviews\_One\_Word.db'** db\_sentences\_one\_word\_name = **'Sentences\_One\_Word.db'** db\_pmi\_review\_name = **'PMI\_Review.db'** db\_pmi\_sentence\_name = **'PMI\_Sentence.db'** db\_pmi\_ideal\_review\_name = **'PMI\_Ideal\_Review.db'** db\_pmi\_ideal\_sentence\_name = **'PMI\_Ideal\_Sentence.db'** db\_path\_weight = **"Path\_Weight.db"** db\_semantic\_distance = **"Semantic\_Distance.db"** db\_semantic\_distance\_ideal = **"Semantic\_Distance\_Ideal.db"** db\_local\_context\_prepare = **"Local\_Context\_Prepare.db"** db\_local\_context\_prepare\_ideal = **"Local\_Context\_Prepare\_Ideal.db"** db\_global\_context\_prepare = **"Global\_Context\_Prepare.db"** db\_global\_context\_prepare\_extra = **"Global\_Context\_Prepare\_Extra.db"** db\_global\_context\_prepare\_ideal = **"Global\_Context\_Prepare\_Ideal.db"** db\_global\_context\_prepare\_extra\_ideal = **"Global\_Context\_Prepare\_Extra\_Ideal.db"** db\_global\_context = **"Global\_Context.db"** db\_local\_context = **"Local\_Context.db"** db\_lexical = **"Lexical.db"** db\_lexical\_ideal = **"Lexical\_Ideal.db"** db\_syntactic = **"Syntactic.db"** db\_syntactic\_ideal = **"Syntactic\_Ideal.db"** db\_tree = **"Tree.db"** db\_local\_context\_ideal = **"Local\_Context\_Ideal.db"** db\_global\_context\_ideal = **"Global\_Context\_Ideal.db"** db\_hierarchy = **"Hierarchy.db"** db\_frequent = **"Frequent.db"** db\_ideal\_full = **"Ideal\_Full.db"** db\_hierarchy\_real = **"Hierarchy\_Real.db"** db\_semantic\_distance\_real = **"Semantic\_Distance\_Real.db"  
  
 def** \_\_init\_\_(self):  
 path = os.getcwd()  
 self.conn\_aspects = sqlite3.connect(path + **"/../db/"** + self.db\_aspects\_name)  
 self.conn\_reviews = sqlite3.connect(path + **"/../db/"** + self.db\_reviews\_name)  
 self.conn\_merged = sqlite3.connect(path + **"/../db/"** + self.db\_merged\_name)  
 self.conn\_sentence = sqlite3.connect(path + **"/../db/"** + self.db\_sentence\_name)  
 self.conn\_aspects\_one\_word = sqlite3.connect(path + **"/../db/"** + self.db\_aspects\_one\_word\_name)  
 self.conn\_reviews\_one\_word = sqlite3.connect(path + **"/../db/"** + self.db\_reviews\_one\_word\_name)  
 self.conn\_sentences\_one\_word = sqlite3.connect(path + **"/../db/"** + self.db\_sentences\_one\_word\_name)  
 self.cursor\_sentences\_one\_word\_update = sqlite3.connect(path + **"/../db/"** + self.db\_sentences\_one\_word\_name)  
 self.conn\_pmi\_review = sqlite3.connect(path + **"/../db/"** + self.db\_pmi\_review\_name)  
 self.conn\_pmi\_sentence = sqlite3.connect(path + **"/../db/"** + self.db\_pmi\_sentence\_name)  
 self.conn\_pmi\_ideal\_review = sqlite3.connect(path + **"/../db/"** + self.db\_pmi\_ideal\_review\_name)  
 self.conn\_pmi\_ideal\_sentence = sqlite3.connect(path + **"/../db/"** + self.db\_pmi\_ideal\_sentence\_name)  
 self.conn\_path\_weight = sqlite3.connect(path + **"/../db/"** + self.db\_path\_weight)  
 self.conn\_semantic\_distance = sqlite3.connect(path + **"/../db/"** + self.db\_semantic\_distance)  
 self.conn\_semantic\_distance\_ideal = sqlite3.connect(path + **"/../db/"** + self.db\_semantic\_distance\_ideal)  
 self.conn\_local\_context\_prepare = sqlite3.connect(path + **"/../db/"** + self.db\_local\_context\_prepare)  
 self.conn\_local\_context\_prepare\_ideal = sqlite3.connect(path + **"/../db/"** + self.db\_local\_context\_prepare\_ideal)  
 self.conn\_global\_context\_prepare = sqlite3.connect(path + **"/../db/"** + self.db\_global\_context\_prepare)  
 self.conn\_global\_context\_prepare\_extra = sqlite3.connect(path + **"/../db/"** + self.db\_global\_context\_prepare\_extra)  
 self.conn\_global\_context\_prepare\_ideal = sqlite3.connect(path + **"/../db/"** + self.db\_global\_context\_prepare\_ideal)  
 self.conn\_global\_context\_prepare\_extra\_ideal = sqlite3.connect(path + **"/../db/"** + self.db\_global\_context\_prepare\_extra\_ideal)  
 self.conn\_global\_context = sqlite3.connect(path + **"/../db/"** + self.db\_global\_context)  
 self.conn\_local\_context = sqlite3.connect(path + **"/../db/"** + self.db\_local\_context)  
 self.conn\_lexical = sqlite3.connect(path + **"/../db/"** + self.db\_lexical)  
 self.conn\_syntactic = sqlite3.connect(path + **"/../db/"** + self.db\_syntactic)  
 self.conn\_tree = sqlite3.connect(path + **"/../db/"** + self.db\_tree)  
 self.conn\_lexical\_ideal = sqlite3.connect(path + **"/../db/"** + self.db\_lexical\_ideal)  
 self.conn\_syntactic\_ideal = sqlite3.connect(path + **"/../db/"** + self.db\_syntactic\_ideal)  
 self.conn\_local\_context\_ideal = sqlite3.connect(path + **"/../db/"** + self.db\_local\_context\_ideal)  
 self.conn\_global\_context\_ideal = sqlite3.connect(path + **"/../db/"** + self.db\_global\_context\_ideal)  
 self.conn\_hierarchy = sqlite3.connect(path + **"/../db/"** + self.db\_hierarchy)  
 self.conn\_frequent = sqlite3.connect(path + **"/../db/"** + self.db\_frequent)  
 self.conn\_ideal\_full = sqlite3.connect(path + **"/../db/"** + self.db\_ideal\_full)  
 self.conn\_hierarchy\_real = sqlite3.connect(path + **"/../db/"** + self.db\_hierarchy\_real)  
 self.conn\_semantic\_distance\_real = sqlite3.connect(path + **"/../db/"** + self.db\_semantic\_distance\_real)  
  
 self.cursor\_merged = self.conn\_merged.cursor()  
 self.cursor\_aspects = self.conn\_aspects.cursor()  
 self.cursor\_aspects2 = self.conn\_aspects.cursor()  
 self.cursor\_reviews = self.conn\_reviews.cursor()  
 self.cursor\_article = self.conn\_aspects.cursor()  
 self.cursor\_sentence = self.conn\_sentence.cursor()  
 self.cursor\_aspects\_one\_word = self.conn\_aspects\_one\_word.cursor()  
 self.cursor\_reviews\_one\_word = self.conn\_reviews\_one\_word.cursor()  
 self.cursor\_reviews\_one\_word\_update = self.conn\_reviews\_one\_word.cursor()  
 self.cursor\_sentences\_one\_word = self.conn\_sentences\_one\_word.cursor()  
 self.cursor\_pmi\_review = self.conn\_pmi\_review.cursor()  
 self.cursor\_pmi\_sentence = self.conn\_pmi\_sentence.cursor()  
 self.cursor\_pmi\_ideal\_review = self.conn\_pmi\_ideal\_review.cursor()  
 self.cursor\_pmi\_ideal\_sentence = self.conn\_pmi\_ideal\_sentence.cursor()  
 self.cursor\_path\_weight = self.conn\_path\_weight.cursor()  
 self.cursor\_semantic\_distance = self.conn\_semantic\_distance.cursor()  
 self.cursor\_semantic\_distance\_ideal = self.conn\_semantic\_distance\_ideal.cursor()  
 self.cursor\_local\_context\_prepare = self.conn\_local\_context\_prepare.cursor()  
 self.cursor\_local\_context\_prepare\_ideal = self.conn\_local\_context\_prepare\_ideal.cursor()  
 self.cursor\_global\_context\_prepare = self.conn\_global\_context\_prepare.cursor()  
 self.cursor\_global\_context\_prepare\_extra = self.conn\_global\_context\_prepare\_extra.cursor()  
 self.cursor\_global\_context\_prepare\_ideal = self.conn\_global\_context\_prepare\_ideal.cursor()  
 self.cursor\_global\_context\_prepare\_extra\_ideal = self.conn\_global\_context\_prepare\_extra\_ideal.cursor()  
 self.cursor\_global\_context = self.conn\_global\_context.cursor()  
 self.cursor\_local\_context = self.conn\_local\_context.cursor()  
 self.cursor\_lexical = self.conn\_lexical.cursor()  
 self.cursor\_lexical\_ideal = self.conn\_lexical\_ideal.cursor()  
 self.cursor\_syntactic = self.conn\_syntactic.cursor()  
 self.cursor\_tree = self.conn\_tree.cursor()  
 self.cursor\_syntactic\_ideal = self.conn\_syntactic\_ideal.cursor()  
 self.cursor\_local\_context\_ideal = self.conn\_local\_context\_ideal.cursor()  
 self.cursor\_global\_context\_ideal = self.conn\_global\_context\_ideal.cursor()  
 self.cursor\_hierarchy = self.conn\_hierarchy.cursor()  
 self.cursor\_frequent = self.conn\_frequent.cursor()  
 self.cursor\_ideal\_full = self.conn\_ideal\_full.cursor()  
 self.cursor\_hierarchy\_real = self.conn\_hierarchy\_real.cursor()  
 self.cursor\_semantic\_distance\_real = self.conn\_semantic\_distance\_real.cursor()  
  
 **def** create\_semantic\_distance\_real\_db(self):  
 self.cursor\_semantic\_distance\_real.execute(  
 **'''CREATE TABLE IF NOT EXISTS Distance (aspect1 TEXT, aspect2 TEXT, distance FLOAT)'''**)  
 self.conn\_semantic\_distance\_real.commit()  
  
 **def** create\_hierarchy\_real\_db(self):  
 self.cursor\_hierarchy\_real.execute(**'''CREATE TABLE IF NOT EXISTS Hierarchy (parent TEXT, child TEXT)'''**)  
 self.conn\_hierarchy\_real.commit()  
  
 **def** create\_ideal\_full\_db(self):  
 self.cursor\_ideal\_full.execute(**'''CREATE TABLE IF NOT EXISTS Ideal (filename TEXT, aspect1 TEXT, aspect2 TEXT, pmi\_review FLOAT, pmi\_sentence FLOAT, lexical FLOAT, syntactic FLOAT, local\_context FLOAT, global\_context FLOAT, weight FLOAT)'''**)  
 self.conn\_ideal\_full.commit()  
  
 **def** create\_frequent\_db(self):  
 self.cursor\_frequent.execute(**'''CREATE TABLE IF NOT EXISTS Frequent (word TEXT, number INT)'''**)  
 self.conn\_frequent.commit()  
  
 **def** create\_hierarchy\_db(self):  
 self.cursor\_hierarchy.execute(**'''CREATE TABLE IF NOT EXISTS Hierarchy (parent TEXT, child TEXT)'''**)  
 self.conn\_hierarchy.commit()  
  
 **def** create\_context\_global\_ideal\_db(self):  
 self.cursor\_global\_context\_ideal.execute(**'''CREATE TABLE IF NOT EXISTS Context (aspect1 TEXT, aspect2 TEXT, kl\_divergence DOUBLE)'''**)  
 self.conn\_global\_context\_ideal.commit()  
  
 **def** create\_context\_local\_ideal\_db(self):  
 self.cursor\_local\_context\_ideal.execute(**'''CREATE TABLE IF NOT EXISTS Context (aspect1 TEXT, aspect2 TEXT, kl\_divergence DOUBLE)'''**)  
 self.conn\_local\_context\_ideal.commit()  
  
 **def** create\_tree\_db(self):  
 self.cursor\_tree.execute(**'''CREATE TABLE IF NOT EXISTS Tree (sentence TEXT, tree TEXT)'''**)  
 self.conn\_tree.commit()  
  
 **def** create\_syntactic\_ideal\_db(self):  
 self.cursor\_syntactic\_ideal.execute(**'''CREATE TABLE IF NOT EXISTS Syntactic (aspect1 TEXT, aspect2 TEXT, syntactic\_path INT)'''**)  
 self.conn\_syntactic\_ideal.commit()  
  
 **def** create\_syntactic\_db(self):  
 self.cursor\_syntactic.execute(**'''CREATE TABLE IF NOT EXISTS Syntactic (aspect1 TEXT, aspect2 TEXT, syntactic\_path INT)'''**)  
 self.conn\_syntactic.commit()  
  
 **def** create\_lexical\_db(self):  
 self.cursor\_lexical.execute(**'''CREATE TABLE IF NOT EXISTS Lexical (aspect1 TEXT, aspect2 TEXT, length\_difference INT)'''**)  
 self.conn\_lexical.commit()  
  
 **def** create\_lexical\_ideal\_db(self):  
 self.cursor\_lexical\_ideal.execute(**'''CREATE TABLE IF NOT EXISTS Lexical (aspect1 TEXT, aspect2 TEXT, length\_difference INT)'''**)  
 self.conn\_lexical\_ideal.commit()  
  
 **def** create\_context\_global\_db(self):  
 self.cursor\_global\_context.execute(**'''CREATE TABLE IF NOT EXISTS Context (aspect1 TEXT, aspect2 TEXT, kl\_divergence DOUBLE)'''**)  
 self.conn\_global\_context.commit()  
  
 **def** create\_context\_local\_db(self):  
 self.cursor\_local\_context.execute(**'''CREATE TABLE IF NOT EXISTS Context (aspect1 TEXT, aspect2 TEXT, kl\_divergence DOUBLE)'''**)  
 self.conn\_local\_context.commit()  
  
 **def** create\_context\_local\_prepare\_db(self):  
 self.cursor\_local\_context\_prepare.execute(**'''CREATE TABLE IF NOT EXISTS Context (aspect TEXT, context TEXT)'''**)  
 self.conn\_local\_context\_prepare.commit()  
  
 **def** create\_context\_local\_prepare\_ideal\_db(self):  
 self.cursor\_local\_context\_prepare\_ideal.execute(**'''CREATE TABLE IF NOT EXISTS Context (aspect TEXT, context TEXT)'''**)  
 self.conn\_local\_context\_prepare\_ideal.commit()  
  
 **def** create\_context\_global\_prepare\_ideal\_db(self):  
 self.cursor\_global\_context\_prepare\_ideal.execute(**'''CREATE TABLE IF NOT EXISTS Context (aspect TEXT, review TEXT)'''**)  
 self.conn\_global\_context\_prepare\_ideal.commit()  
  
 **def** create\_context\_global\_prepare\_extra\_ideal\_db(self):  
 self.cursor\_global\_context\_prepare\_extra\_ideal.execute(  
 **'''CREATE TABLE IF NOT EXISTS Context (aspect TEXT, context TEXT)'''**)  
 self.conn\_global\_context\_prepare\_extra\_ideal.commit()  
  
 **def** create\_context\_global\_prepare\_db(self):  
 self.cursor\_global\_context\_prepare.execute(**'''CREATE TABLE IF NOT EXISTS Context (aspect TEXT, review TEXT)'''**)  
 self.conn\_global\_context\_prepare.commit()  
  
 **def** create\_context\_global\_prepare\_extra\_db(self):  
 self.cursor\_global\_context\_prepare\_extra.execute(**'''CREATE TABLE IF NOT EXISTS Context (aspect TEXT, context TEXT)'''**)  
 self.conn\_global\_context\_prepare\_extra.commit()  
  
 **def** create\_semantic\_distance\_db(self):  
 self.cursor\_semantic\_distance.execute(**'''CREATE TABLE IF NOT EXISTS Distance (aspect1 TEXT, aspect2 TEXT, distance FLOAT)'''**)  
 self.conn\_semantic\_distance.commit()  
  
 **def** create\_semantic\_distance\_ideal\_db(self):  
 self.cursor\_semantic\_distance\_ideal.execute(**'''CREATE TABLE IF NOT EXISTS Distance (aspect1 TEXT, aspect2 TEXT, distance FLOAT)'''**)  
 self.conn\_semantic\_distance\_ideal.commit()  
  
 **def** create\_path\_weight\_db(self):  
 self.cursor\_path\_weight.execute(**'''CREATE TABLE IF NOT EXISTS Weight (filename TEXT, aspect1 TEXT, aspect2 TEXT, weight INT)'''**)  
 self.conn\_path\_weight.commit()  
  
 **def** create\_pmi\_ideal\_review\_db(self):  
 self.cursor\_pmi\_ideal\_review.execute(**'''CREATE TABLE IF NOT EXISTS PMI  
 (aspect1 TEXT, aspect2 TEXT, aspect1Num INT, aspect2Num INT, bothNum INT, pmi DOUBLE)'''**)  
 self.conn\_pmi\_ideal\_review.commit()  
  
 **def** create\_pmi\_ideal\_sentence\_db(self):  
 self.cursor\_pmi\_ideal\_sentence.execute(**'''CREATE TABLE IF NOT EXISTS PMI  
 (aspect1 TEXT, aspect2 TEXT, aspect1Num INT, aspect2Num INT, bothNum INT, pmi DOUBLE)'''**)  
 self.conn\_pmi\_ideal\_sentence.commit()  
  
 **def** create\_pmi\_review\_db(self):  
 self.cursor\_pmi\_review.execute(**'''CREATE TABLE IF NOT EXISTS PMI  
 (aspect1 TEXT, aspect2 TEXT, aspect1Num INT, aspect2Num INT, bothNum INT, pmi DOUBLE)'''**)  
 self.conn\_pmi\_review.commit()  
  
 **def** create\_pmi\_sentence\_db(self):  
 self.cursor\_pmi\_sentence.execute(**'''CREATE TABLE IF NOT EXISTS PMI  
 (aspect1 TEXT, aspect2 TEXT, aspect1Num INT, aspect2Num INT, bothNum INT, pmi DOUBLE)'''**)  
 self.conn\_pmi\_sentence.commit()  
  
 **def** create\_aspects\_one\_word\_db(self):  
 self.cursor\_aspects\_one\_word.execute(**'''CREATE TABLE IF NOT EXISTS Aspects  
 (article TEXT, advantageAspects TEXT, disadvantageAspects TEXT, commentAspects TEXT)'''**)  
 self.conn\_aspects\_one\_word.commit()  
  
 **def** create\_reviews\_one\_word\_db(self):  
 self.cursor\_reviews\_one\_word.execute(**'''CREATE TABLE IF NOT EXISTS Reviews  
 (article TEXT, advantageAspects TEXT, disadvantageAspects TEXT, commentAspects TEXT)'''**)  
 self.conn\_reviews\_one\_word.commit()  
  
 **def** create\_sentences\_one\_word\_db(self):  
 self.cursor\_sentences\_one\_word.execute(**'''CREATE TABLE IF NOT EXISTS Sentences (article TEXT, sentence TEXT)'''**)  
 self.conn\_sentences\_one\_word.commit()  
  
 **def** create\_aspects\_db(self):  
 self.cursor\_aspects.execute(**'''CREATE TABLE IF NOT EXISTS Aspects  
 (article TEXT, advantageAspects TEXT, disadvantageAspects TEXT, commentAspects TEXT)'''**)  
 self.conn\_aspects.commit()  
  
 **def** create\_sentence\_db(self):  
 self.cursor\_sentence.execute(**'''CREATE TABLE IF NOT EXISTS Sentences (article TEXT, sentence TEXT)'''**)  
 self.conn\_sentence.commit()  
  
 **def** add\_semantic\_distance\_real(self, aspect1, aspect2, distance):  
 self.cursor\_semantic\_distance\_real.execute(**'INSERT INTO Distance (aspect1, aspect2, distance) VALUES (?, ?, ?)'**,  
 (aspect1, aspect2, distance))  
  
 **def** add\_hierarchy\_real(self, parent, child):  
 self.cursor\_hierarchy\_real.execute(**'INSERT INTO Hierarchy (parent, child) VALUES (?, ?)'**, (parent, child))  
  
 **def** add\_ideal\_full(self, filename, aspect1, aspect2, pmi\_review, pmi\_sentence, lexical, syntactic, local\_context, global\_context, weight):  
 self.cursor\_ideal\_full.execute(**'INSERT INTO Ideal (filename, aspect1, aspect2, pmi\_review, pmi\_sentence, lexical, syntactic, local\_context, global\_context, weight) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)'**, (filename, aspect1, aspect2, pmi\_review, pmi\_sentence, lexical, syntactic, local\_context, global\_context, weight,))  
  
 **def** add\_frequent(self, word, number):  
 self.cursor\_frequent.execute(**'INSERT INTO Frequent (word, number) VALUES (?, ?)'**, (word, number))  
  
 **def** add\_hierarchy(self, parent, child):  
 self.cursor\_hierarchy.execute(**'INSERT INTO Hierarchy (parent, child) VALUES (?, ?)'**, (parent, child))  
  
 **def** add\_tree(self, sentence, tree):  
 self.cursor\_tree.execute(**'INSERT INTO Tree (sentence, tree) VALUES (?, ?)'**, (sentence, tree))  
  
 **def** add\_syntactic\_ideal(self, aspect1, aspect2, syntactic\_path):  
 self.cursor\_syntactic\_ideal.execute(**'INSERT INTO Syntactic (aspect1, aspect2, syntactic\_path) VALUES (?, ?, ?)'**,  
 (aspect1, aspect2, syntactic\_path))  
  
 **def** add\_syntactic(self, aspect1, aspect2, syntactic\_path):  
 self.cursor\_syntactic.execute(**'INSERT INTO Syntactic (aspect1, aspect2, syntactic\_path) VALUES (?, ?, ?)'**,  
 (aspect1, aspect2, syntactic\_path))  
  
 **def** add\_lexical(self, aspect1, aspect2, length\_difference):  
 self.cursor\_lexical.execute(**'INSERT INTO Lexical (aspect1, aspect2, length\_difference) VALUES (?, ?, ?)'**,  
 (aspect1, aspect2, length\_difference))  
  
 **def** add\_lexical\_ideal(self, aspect1, aspect2, length\_difference):  
 self.cursor\_lexical\_ideal.execute(**'INSERT INTO Lexical (aspect1, aspect2, length\_difference) VALUES (?, ?, ?)'**,  
 (aspect1, aspect2, length\_difference))  
  
 **def** add\_context\_global\_ideal(self, aspect1, aspect2, kl\_divergence):  
 self.cursor\_global\_context\_ideal.execute(**'INSERT INTO Context (aspect1, aspect2, kl\_divergence) VALUES (?, ?, ?)'**,  
 (aspect1, aspect2, kl\_divergence))  
  
 **def** add\_context\_local\_ideal(self, aspect1, aspect2, kl\_divergence):  
 self.cursor\_local\_context\_ideal.execute(**'INSERT INTO Context (aspect1, aspect2, kl\_divergence) VALUES (?, ?, ?)'**,  
 (aspect1, aspect2, kl\_divergence))  
  
 **def** add\_context\_global(self, aspect1, aspect2, kl\_divergence):  
 self.cursor\_global\_context.execute(**'INSERT INTO Context (aspect1, aspect2, kl\_divergence) VALUES (?, ?, ?)'**,  
 (aspect1, aspect2, kl\_divergence))  
  
 **def** add\_context\_local(self, aspect1, aspect2, kl\_divergence):  
 self.cursor\_local\_context.execute(**'INSERT INTO Context (aspect1, aspect2, kl\_divergence) VALUES (?, ?, ?)'**,  
 (aspect1, aspect2, kl\_divergence))  
  
 **def** add\_context\_local\_prepare(self, aspect, context):  
 self.cursor\_local\_context\_prepare.execute(**'INSERT INTO Context (aspect, context) VALUES (?, ?)'**, (aspect, context))  
  
 **def** add\_context\_local\_prepare\_ideal(self, aspect, context):  
 self.cursor\_local\_context\_prepare\_ideal.execute(**'INSERT INTO Context (aspect, context) VALUES (?, ?)'**, (aspect, context))  
  
 **def** add\_context\_global\_prepare\_ideal(self, aspect, review):  
 self.cursor\_global\_context\_prepare\_ideal.execute(**'INSERT INTO Context (aspect, review) VALUES (?, ?)'**,  
 (aspect, review))  
  
 **def** add\_context\_global\_prepare\_extra\_ideal(self, aspect, context):  
 self.cursor\_global\_context\_prepare\_extra\_ideal.execute(**'INSERT INTO Context (aspect, context) VALUES (?, ?)'**,  
 (aspect, context))  
  
 **def** add\_context\_global\_prepare(self, aspect, review):  
 self.cursor\_global\_context\_prepare.execute(**'INSERT INTO Context (aspect, review) VALUES (?, ?)'**, (aspect, review))  
  
 **def** add\_context\_global\_prepare\_extra(self, aspect, context):  
 self.cursor\_global\_context\_prepare\_extra.execute(**'INSERT INTO Context (aspect, context) VALUES (?, ?)'**,  
 (aspect, context))  
  
 **def** add\_semantic\_distance\_ideal(self, aspect1, aspect2, distance):  
 self.cursor\_semantic\_distance\_ideal.execute(**'INSERT INTO Distance (aspect1, aspect2, distance) VALUES (?, ?, ?)'**,  
 (aspect1, aspect2, distance))  
  
 **def** add\_semantic\_distance(self, aspect1, aspect2, distance):  
 self.cursor\_semantic\_distance.execute(**'INSERT INTO Distance (aspect1, aspect2, distance) VALUES (?, ?, ?)'**,  
 (aspect1, aspect2, distance))  
  
 **def** add\_path\_weight(self, filename, aspect1, aspect2, weight):  
 self.cursor\_path\_weight.execute(**'INSERT INTO Weight (filename, aspect1, aspect2, weight) VALUES (?, ?, ?, ?)'**,  
 (filename, aspect1, aspect2, weight))  
  
 **def** add\_pmi\_ideal\_review(self, aspect1, aspect2, num1, num2, both\_num, pmi):  
 self.cursor\_pmi\_ideal\_review.execute(  
 **'INSERT INTO PMI (aspect1, aspect2, aspect1Num, aspect2Num, bothNum, pmi) VALUES (?, ?, ?, ?, ?, ?)'**,  
 (aspect1, aspect2, num1, num2, both\_num, pmi))  
  
 **def** add\_pmi\_ideal\_sentence(self, aspect1, aspect2, num1, num2, both\_num, pmi):  
 self.cursor\_pmi\_ideal\_sentence.execute(  
 **'INSERT INTO PMI (aspect1, aspect2, aspect1Num, aspect2Num, bothNum, pmi) VALUES (?, ?, ?, ?, ?, ?)'**,  
 (aspect1, aspect2, num1, num2, both\_num, pmi))  
  
 **def** add\_pmi\_review(self, aspect1, aspect2, num1, num2, both\_num, pmi):  
 self.cursor\_pmi\_review.execute(  
 **'INSERT INTO PMI (aspect1, aspect2, aspect1Num, aspect2Num, bothNum, pmi) VALUES (?, ?, ?, ?, ?, ?)'**,  
 (aspect1, aspect2, num1, num2, both\_num, pmi))  
  
 **def** add\_pmi\_sentence(self, aspect1, aspect2, num1, num2, both\_num, pmi):  
 self.cursor\_pmi\_sentence.execute(  
 **'INSERT INTO PMI (aspect1, aspect2, aspect1Num, aspect2Num, bothNum, pmi) VALUES (?, ?, ?, ?, ?, ?)'**,  
 (aspect1, aspect2, num1, num2, both\_num, pmi))  
  
 **def** add\_sentence(self, article, sentence):  
 self.cursor\_sentence.execute(**'INSERT INTO Sentences (article, sentence) VALUES (?, ?)'**, (article, sentence))  
  
 **def** add\_review(self, article, advantage\_aspects, disadvantage\_aspects, comment\_aspects):  
 self.cursor\_aspects.execute(  
 **'INSERT INTO Aspects (article, advantageAspects, disadvantageAspects, commentAspects) VALUES (?, ?, ?, ?)'**,  
 (article, advantage\_aspects, disadvantage\_aspects, comment\_aspects))  
  
 **def** add\_one\_word\_aspects(self, article, advantage\_aspects, disadvantage\_aspects, comment\_aspects):  
 self.conn\_aspects\_one\_word.execute(  
 **'INSERT INTO Aspects (article, advantageAspects, disadvantageAspects, commentAspects) VALUES (?, ?, ?, ?)'**,  
 (article, advantage\_aspects, disadvantage\_aspects, comment\_aspects))  
  
 **def** add\_one\_word\_review(self, article, advantage\_aspects, disadvantage\_aspects, comment\_aspects):  
 self.conn\_reviews\_one\_word.execute(  
 **'INSERT INTO Reviews (article, advantageAspects, disadvantageAspects, commentAspects) VALUES (?, ?, ?, ?)'**,  
 (article, advantage\_aspects, disadvantage\_aspects, comment\_aspects))  
  
 **def** add\_one\_word\_sentence(self, article, sentence):  
 self.conn\_sentences\_one\_word.execute(**'INSERT INTO Sentences (article, sentence) VALUES (?, ?)'**,  
 (article, sentence))  
  
 *# destructor - close connection* **def** \_\_del\_\_(self):  
 self.conn\_aspects.close()  
 self.conn\_reviews.close()  
  
 **def** delete\_aspects(self, article):  
 self.cursor\_aspects.execute(**'DELETE FROM Aspects WHERE article = '** + str(article))  
 self.conn\_aspects.commit()

# FrequentAspects.py

**class FrequentAspects**:  
  
 **def** process(self, db):  
 db.create\_frequent\_db()  
 row\_aspect = db.cursor\_aspects\_one\_word.execute(**'SELECT** *\** **FROM Aspects'**).fetchone()  
 count = 0  
 dict = {}  
 **while** row\_aspect **is not None**: *# iterate through all reviews* print(count)  
 count += 1  
 adv = str(row\_aspect[1])  
 dis = str(row\_aspect[2])  
 com = str(row\_aspect[3])  
 **if** len(adv) != 0:  
 words = adv.split(**";"**)  
 **for** word **in** words:  
 word = self.replacer(word)  
 **if** word **not in** dict:  
 dict[word] = 1  
 **else**:  
 dict[word] += 1  
 **if** len(dis) != 0:  
 words = dis.split(**";"**)  
 **for** word **in** words:  
 word = self.replacer(word)  
 **if** word **not in** dict:  
 dict[word] = 1  
 **else**:  
 dict[word] += 1  
 **if** len(com) != 0:  
 words = com.split(**";"**)  
 **for** word **in** words:  
 word = self.replacer(word)  
 **if** word **not in** dict:  
 dict[word] = 1  
 **else**:  
 dict[word] += 1  
 row\_aspect = db.cursor\_aspects\_one\_word.fetchone()  
 *# have a dictionary with aspects and their numbers* **import** operator  
 sorted\_dict = sorted(dict.items(), key=operator.itemgetter(1), reverse=**True**)  
 count = 0  
 **for** word **in** sorted\_dict:  
 **if** count < 1000:  
 db.add\_frequent(word[0], word[1])  
 **else**:  
 **break** count += 1  
 db.conn\_frequent.commit()  
  
 @staticmethod  
 **def** replacer(item):  
 item = item.replace(**"\r"**, **" "**)  
 item = item.replace(**"\t"**, **" "**)  
 item = item.replace(**","**, **""**)  
 item = item.replace(**"."**, **""**)  
 item = item.replace(**"•"**, **""**)  
 item = item.replace(**";"**, **""**)  
 item = item.replace(**":"**, **""**)  
 item = item.replace(**"!"**, **""**)  
 item = item.replace(**"?"**, **""**)  
 item = item.replace(**")"**, **""**)  
 item = item.replace(**"("**, **""**)  
 item = item.replace(**"™"**, **""**)  
 item = item.replace(**"®"**, **""**)  
 item = item.replace(**"\*"**, **""**)  
 item = item.replace(**"\""**, **""**)  
 item = item.replace(**"~"**, **""**)  
 item = item.replace(**"'"**, **""**)  
 **return** item.lower()

# HierarchyBuilder.py

**import** codecs  
  
  
**from** aspects.FrequentAspects **import** FrequentAspects  
  
  
**class** HierarchyBuilder:  
 @staticmethod  
 **def** calculate\_average\_semantic\_distance\_ideal\_tree(db):  
 row\_semantic\_distance\_ideal = db.cursor\_semantic\_distance\_ideal.execute(**'SELECT** *\** **FROM Distance'**).fetchone()  
 avg = 0  
 count = 0  
 **while** row\_semantic\_distance\_ideal **is not None**:  
 count += 1  
 avg += float(row\_semantic\_distance\_ideal[2])  
 row\_semantic\_distance\_ideal = db.cursor\_semantic\_distance\_ideal.fetchone()  
 average\_semantic\_distance\_ideal = avg / count  
 **return** average\_semantic\_distance\_ideal  
  
 **def** process\_article\_algo(self, db, average\_semantic\_distance\_ideal):  
 db.create\_hierarchy\_db()  
 self.build\_hierarchy(db, average\_semantic\_distance\_ideal\*\*(1/2.3))  
  
 **def** process\_random\_classifier\_algo(self, db, average\_semantic\_distance\_ideal\_real):  
 db.create\_hierarchy\_real\_db()  
 self.build\_hierarchy\_real(db, average\_semantic\_distance\_ideal\_real)  
  
 **def** build\_hierarchy(self, db, average\_semantic\_distance\_ideal):  
 free\_nodes = self.find\_free\_nodes()  
 count = 0  
 **for** node **in** free\_nodes:  
 print(count)  
 count += 1  
 str = FrequentAspects.replacer(node).replace(**" "**, **"\_"**)  
 row\_semantic\_distance = db.cursor\_semantic\_distance.execute(  
 **'SELECT** *\** **FROM Distance WHERE aspect1 = ? OR aspect2 = ?'**, (str, str,)).fetchone()  
 **while** row\_semantic\_distance **is not None**:  
 *# add as children those whose semantic distance is less them average distance in ideal tree* **if** row\_semantic\_distance[2] <= average\_semantic\_distance\_ideal:  
 aspect1 = row\_semantic\_distance[0]  
 aspect2 = row\_semantic\_distance[1]  
 **if** aspect1 == str:  
 row = db.cursor\_hierarchy.execute(**'SELECT** *\** **FROM Hierarchy WHERE parent = ? AND child = ?'**,  
 (str, aspect2)).fetchone()  
 **if** row **is None**:  
 db.add\_hierarchy(str, aspect2)  
 free\_nodes.append(aspect2)  
 **else**:  
 row = db.cursor\_hierarchy.execute(**'SELECT** *\** **FROM Hierarchy WHERE parent = ? AND child = ?'**,  
 (str, aspect1)).fetchone()  
 **if** row **is None**:  
 db.add\_hierarchy(str, aspect1)  
 free\_nodes.append(aspect1)  
 row\_semantic\_distance = db.cursor\_semantic\_distance.fetchone()  
 db.conn\_hierarchy.commit()  
 print(len(free\_nodes))  
  
 **def** build\_hierarchy\_real(self, db, average\_semantic\_distance\_ideal\_real):  
 free\_nodes = self.find\_free\_nodes()  
 count = 0  
 **for** node **in** free\_nodes:  
 print(count)  
 count += 1  
 str = FrequentAspects.replacer(node).replace(**" "**, **"\_"**)  
 row\_semantic\_distance = db.cursor\_semantic\_distance\_real.execute(  
 **'SELECT** *\** **FROM Distance WHERE aspect1 = ? OR aspect2 = ?'**, (str, str,)).fetchone()  
 **while** row\_semantic\_distance **is not None**:  
 *# add as children those whose semantic distance is less them average distance in ideal tree* **if** row\_semantic\_distance[2] <= average\_semantic\_distance\_ideal\_real:  
 aspect1 = row\_semantic\_distance[0]  
 aspect2 = row\_semantic\_distance[1]  
 **if** aspect1 == str:  
 row = db.cursor\_hierarchy\_real.execute(**'SELECT** *\** **FROM Hierarchy WHERE parent = ? AND child = ?'**,  
 (str, aspect2)).fetchone()  
 **if** row **is None**:  
 db.add\_hierarchy\_real(str, aspect2)  
 free\_nodes.append(aspect2)  
 **else**:  
 row = db.cursor\_hierarchy\_real.execute(**'SELECT** *\** **FROM Hierarchy WHERE parent = ? AND child = ?'**,  
 (str, aspect1)).fetchone()  
 **if** row **is None**:  
 db.add\_hierarchy\_real(str, aspect1)  
 free\_nodes.append(aspect1)  
 row\_semantic\_distance = db.cursor\_semantic\_distance\_real.fetchone()  
 db.conn\_hierarchy\_real.commit()  
 print(len(free\_nodes))  
  
 @staticmethod  
 **def** find\_free\_nodes():  
 **import** os  
 free\_nodes = []  
 path = os.getcwd()  
 filenames = os.listdir(path + **"/../productTrees/Tree"**)  
 os.chdir(path + **"/../productTrees/Tree"**)  
 filenames.remove(**".DS\_Store"**)  
 filenames.remove(**"Subcategories.txt"**)  
 **for** filename **in** filenames:  
 lines = codecs.open(filename, **'r'**, **'cp1251'**).readlines()  
 max\_ind = 2 *# max depth index of file* **for** line **in** lines:  
 arr = line.split(**";"**)  
 arr[2] = arr[2].replace(**"\r\n"**, **""**)  
 **if** str(max\_ind) **in** arr **and** arr[0] **not in** free\_nodes: *# the node is the free one for concrete file* free\_nodes.append(arr[0])  
 **return** free\_nodes  
  
 @staticmethod  
 **def** calculate\_average\_semantic\_distance\_ideal\_tree\_real(db):  
 row\_semantic\_distance\_ideal = db.cursor\_path\_weight.execute(**'SELECT** *\** **FROM Weight'**).fetchone()  
 avg = 0  
 count = 0  
 **while** row\_semantic\_distance\_ideal **is not None**:  
 count += 1  
 avg += float(row\_semantic\_distance\_ideal[3])  
 row\_semantic\_distance\_ideal = db.cursor\_path\_weight.fetchone()  
 average\_semantic\_distance\_ideal = avg / count  
 **return** average\_semantic\_distance\_ideal

# IdealAspectsDB.py

**import** sqlite3  
**import** os  
  
  
**class** IdealAspectsDB:  
 conn\_aspects = **None** cursor\_aspects = **None** cursor\_aspects\_update = **None** db\_aspects\_name = **'IdealAspects\_Ulmart.db'  
  
 def** \_\_init\_\_(self):  
 path = os.getcwd()  
 self.conn\_aspects = sqlite3.connect(path + **"\\..\\db\\"** + self.db\_aspects\_name)  
 self.conn\_aspects\_trees = sqlite3.connect(path + **"\\..\\db\\"** + self.db\_aspects\_name\_trees)  
 self.cursor\_aspects = self.conn\_aspects.cursor()  
 self.cursor\_aspects\_update = self.conn\_aspects.cursor()  
 self.cursor\_trees = self.conn\_aspects\_trees.cursor()  
 self.create\_aspects\_db()  
  
 *# Create table* **def** create\_aspects\_db(self):  
 self.conn\_aspects.execute(**'''CREATE TABLE IF NOT EXISTS IdealAspects  
 (article TEXT, advantageAspects TEXT, disadvantageAspects TEXT, commentAspects TEXT)'''**)  
 self.conn\_aspects.commit()  
  
 *# Insert new review to DB* **def** add\_review(self, article, advantage\_aspects, disadvantage\_aspects, comment\_aspects):  
 self.conn\_aspects.execute(  
 **'INSERT INTO IdealAspects (article, advantageAspects, disadvantageAspects, commentAspects) '  
 'VALUES (?, ?, ?, ?)'**,  
 (article, advantage\_aspects, disadvantage\_aspects, comment\_aspects))  
  
 *# destructor - close connection* **def** \_\_del\_\_(self):  
 self.conn\_aspects\_trees.close()  
 self.conn\_aspects.close()  
  
 **def** remove\_duplicates(self):  
 row = self.cursor\_aspects.execute(**'SELECT** *\** **FROM IdealAspects'**).fetchone()  
 count = 0  
 **while** row **is not None**:  
 print(count)  
 count += 1  
 article = str(row[0])  
 adv = str(row[1])  
 dis = str(row[2])  
 com = str(row[3])  
 new\_str = self.process(adv)  
 **if** new\_str != adv:  
 self.cursor\_aspects\_update.execute(  
 **'UPDATE IdealAspects SET advantageAspects = ? WHERE article = ? AND advantageAspects = ?'**,  
 (new\_str, article, adv,))  
 self.conn\_aspects.commit()  
 new\_str = self.process(dis)  
 **if** new\_str != dis:  
 self.cursor\_aspects\_update.execute(  
 **'UPDATE IdealAspects SET disadvantageAspects = ? WHERE article = ? AND disadvantageAspects = ?'**,  
 (new\_str, article, dis,))  
 self.conn\_aspects.commit()  
 new\_str = self.process(com)  
 **if** new\_str != com:  
 self.cursor\_aspects\_update.execute(  
 **'UPDATE IdealAspects SET commentAspects = ? WHERE article = ? AND commentAspects = ?'**,  
 (new\_str, article, com,))  
 self.conn\_aspects.commit()  
 row = self.cursor\_aspects.fetchone()  
  
 @staticmethod  
 **def** process(part):  
 **if** len(part) != 0:  
 new\_str = **""** arr = part.split(**";"**)  
 **for** i **in** range(len(arr)):  
 flag = **False  
 for** j **in** range(i + 1, len(arr)):  
 **if** arr[i] == arr[j]:  
 flag = **True  
 if not** flag:  
 new\_str += arr[i] + **";"** new\_str = new\_str[0:len(new\_str) - 1]  
 **return** new\_str  
 **return ""  
  
 def** count\_aspects(self):  
 row = self.cursor\_aspects.execute(**'SELECT** *\** **FROM IdealAspects'**).fetchone()  
 count = 0  
 aspects\_num = 0  
 **while** row **is not None**:  
 print(count)  
 count += 1  
 adv = str(row[1])  
 dis = str(row[2])  
 com = str(row[3])  
 **if** len(adv) != 0:  
 arr = adv.split(**";"**)  
 aspects\_num += len(arr)  
 **if** len(dis) != 0:  
 arr = dis.split(**";"**)  
 aspects\_num += len(arr)  
 **if** len(com) != 0:  
 arr = com.split(**";"**)  
 aspects\_num += len(arr)  
 row = self.cursor\_aspects.fetchone()  
 print(aspects\_num)

# Lexical.py

**class** Lexical:  
  
 @staticmethod  
 **def** process(aspects, db):  
 db.create\_lexical\_db()  
 only\_aspect\_words = []  
 **for** aspect **in** aspects:  
 only\_aspect\_words.append(aspect)  
 **for** i **in** range(len(only\_aspect\_words)):  
 print(i)  
 **for** j **in** range(i + 1, len(only\_aspect\_words)):  
 length\_difference = abs(len(only\_aspect\_words[i]) - len(only\_aspect\_words[j]))  
 db.add\_lexical(only\_aspect\_words[i], only\_aspect\_words[j], length\_difference)  
 db.conn\_lexical.commit()  
  
 @staticmethod  
 **def** process\_ideal(db):  
 db.create\_lexical\_ideal\_db()  
 row\_ideal = db.cursor\_pmi\_ideal\_review.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 **while** row\_ideal **is not None**:  
 aspect1 = str(row\_ideal[0])  
 aspect2 = str(row\_ideal[1])  
 length\_difference = abs(len(aspect1) - len(aspect2))  
 db.add\_lexical\_ideal(aspect1, aspect2, length\_difference)  
 row\_ideal = db.cursor\_pmi\_ideal\_review.fetchone()  
 db.conn\_lexical\_ideal.commit()

# Main.py

**from** sklearn.model\_selection **import** train\_test\_split  
  
**from** aspects.Aspects **import** Aspects  
**from** aspects.Context **import** Context  
**from** aspects.DB **import** DB  
**from** aspects.RandomForest **import** RandomForest  
**from** aspects.FrequentAspects **import** FrequentAspects  
**from** aspects.HierarchyBuilder **import** HierarchyBuilder  
**from** aspects.IdealAspectsDB **import** IdealAspectsDB  
**from** aspects.Lexical **import** Lexical  
**from** aspects.OneClassSVM **import** OneClassSVM  
**from** aspects.PMI **import** PMI  
**from** aspects.SemanticDistanceLearning **import** SemanticDistanceLearning  
**from** aspects.Sentence **import** Sentence  
**from** aspects.Splitter **import** Splitter  
**from** aspects.Syntactic **import** Syntactic  
  
  
**class** Main:  
 vocabulary = **None** db = **None** aspect = **None  
  
 def** \_\_init\_\_(self):  
 self.db = DB() *# data base* self.aspect = Aspects()  
 self.aspects\_process()  
 self.sentence\_process()  
 self.split\_process()  
 self.frequent()  
 self.pmi\_process()  
 self.contextual\_features()  
 self.lexical\_features()  
 self.syntactic\_features()  
 self.semantic\_learning\_process()  
 self.ensemble\_classifier()  
 self.hierarchy()  
  
 **def** aspects\_process(self):  
 self.aspect.process(self.aspect, self.db) *# find aspects with the help of ISP RAS API  
 # clean the data with the help of unnecessary class* one\_class\_svm = OneClassSVM()  
 data = one\_class\_svm.get\_data(self.db) *# get only aspects from data base  
 # get labels for all the aspects depends on their ideality* labels = one\_class\_svm.get\_labels(data, self.db)  
 *# split the data (80% for training)* train\_data, test\_data, train\_labels, test\_labels = train\_test\_split(data, labels, test\_size=0.2)  
 *# unarray the 2D arrays and make them 1D* test\_data\_unarrayed = one\_class\_svm.unarray(test\_data)  
 train\_data\_unarrayed = one\_class\_svm.unarray(train\_data)  
 train\_labels\_unarrayed = one\_class\_svm.unarray(train\_labels)  
 *# get only ideal aspects from aspects list (label = 1) for train data* train\_data\_unarrayed = one\_class\_svm.get\_ideal\_data(train\_data\_unarrayed, train\_labels\_unarrayed)  
 *# train the one-class SVM and predict the aspects* test\_labels\_unarrayed = one\_class\_svm.train\_and\_predict(train\_data\_unarrayed, test\_data\_unarrayed)  
 *# get only ideal aspects from aspects list (label = 1) for test data* test\_data\_unarrayed = one\_class\_svm.get\_ideal\_data(test\_data\_unarrayed, test\_labels\_unarrayed)  
 *# now the sum of test\_data\_unarrayed and train\_data\_unarrayed have only ideal aspects* ideal\_aspects = test\_data\_unarrayed + train\_data\_unarrayed  
 ideal = IdealAspectsDB()  
 ideal.count\_aspects() *# number of ideal aspects  
 # got only ideal aspects in the db* self.aspect.move\_ideal\_aspects(ideal, ideal\_aspects, self.db)  
 *# clean the data with the help of unnecessary class* synonyms = Synonyms()  
 synonyms.find\_synonyms(ideal) *# find and group the synonyms* **def** sentence\_process(self):  
 sentence = Sentence()  
 sentence.process(self.db, self.aspect) *# create a db with all sentences from reviews* **def** split\_process(self):  
 sp = Splitter() *# split the words with multiple \_* sp.process\_reviews(self.db)  
 sp.process\_sentences(self.db)  
  
 **def** pmi\_process(self):  
 pmi = PMI()  
 reviews\_corpus = pmi.get\_all\_reviews\_corpus(self.db) *# reviews* sentences\_corpus = pmi.get\_all\_sentences\_corpus(self.db) *# sentences* self.pmi\_ideal(pmi, reviews\_corpus, sentences\_corpus) *# ideal* self.vocabulary = pmi.get\_vocabulary(self.db)  
 self.vocabulary = pmi.get\_frequent\_vocabulary(self.db)  
 self.pmi\_review(pmi, reviews\_corpus, self.vocabulary)  
 self.pmi\_sentence(pmi, sentences\_corpus, self.vocabulary)  
  
 **def** pmi\_review(self, pmi, reviews\_corpus, vocabulary):  
 self.db.create\_pmi\_review\_db()  
 pmi.calculate\_pmi(reviews\_corpus, 0, vocabulary, self.db)  
 self.db.conn\_pmi\_review.commit()  
  
 **def** pmi\_sentence(self, pmi, sentences\_corpus, vocabulary):  
 self.db.create\_pmi\_sentence\_db()  
 pmi.calculate\_pmi(sentences\_corpus, 1, vocabulary, self.db)  
 self.db.conn\_pmi\_sentence.commit()  
  
 **def** pmi\_ideal(self, pmi, reviews\_corpus, sentences\_corpus):  
 self.db.create\_pmi\_ideal\_review\_db()  
 self.db.create\_pmi\_ideal\_sentence\_db()  
 pmi.iterate\_ideal\_aspects\_files(pmi, reviews\_corpus, sentences\_corpus, self.db)  
  
 **def** semantic\_learning\_process(self):  
 semantic\_learning = SemanticDistanceLearning()  
 self.db.create\_path\_weight\_db()  
 semantic\_learning.calculate\_ground\_truth\_distance(self.db)  
 semantic\_learning.process\_semantic\_distance\_learning\_ideal(self.db)  
 semantic\_learning.process\_semantic\_distance\_learning(self.db)  
 semantic\_learning.print\_data(self.db)  
  
 **def** contextual\_features(self):  
 context = Context()  
 context.process(self.db, self.vocabulary)  
 context.process\_ideal(self.db)  
  
 **def** lexical\_features(self):  
 lexical = Lexical()  
 lexical.process(self.vocabulary, self.db)  
 lexical.process\_ideal(self.db)  
  
 **def** syntactic\_features(self):  
 syntactic = Syntactic()  
 syntactic.process\_ideal(self.db)  
 syntactic.process(self.db, self.vocabulary, self.aspect)  
  
 **def** frequent(self):  
 f = FrequentAspects()  
 f.process(self.db)  
  
 **def** hierarchy(self):  
 h = HierarchyBuilder()  
 average\_semantic\_distance\_ideal\_calc = h.calculate\_average\_semantic\_distance\_ideal\_tree(self.db) *# 51.71624897381593* average\_semantic\_distance\_ideal\_real = h.calculate\_average\_semantic\_distance\_ideal\_tree\_real(self.db) *# 3.150705186533212* h.process\_article\_algo(self.db, average\_semantic\_distance\_ideal\_calc)  
 h.process\_random\_classifier\_algo(self.db, average\_semantic\_distance\_ideal\_real)  
  
 **def** ensemble\_classifier(self):  
 en = RandomForest()  
 en.form\_train(self.db)  
 x\_train, y\_train = en.get\_train(self.db)  
 x\_test = en.get\_test(self.db)  
 y\_test = en.process(x\_test, x\_train, y\_train)  
 sem = SemanticDistanceLearning()  
 sem.create\_semantic\_distance\_random\_forest\_classifier(self.db, y\_test)  
  
main = Main()  
  
  
*# Sizes data  
# len(data, labels) = 24093  
# len(train\_data, train\_labels) = 19274  
# len(test\_data, test\_labels) = 4819  
# len(train\_data\_unarrayed) = 619286  
# len(test\_data\_unarrayed) = 154951  
# len(all\_aspects) = 774237  
# len(ideal\_train\_data\_unarrayed) = 46149  
# len(ideal\_test\_data\_unarrayed) = 124709  
# len(ideal\_aspects\_dictionary) = 170858  
# len(ideal\_aspects) = 540571  
# len(grouped aspects) = 421715  
# len(vocabulary) = 45435  
# len(pmi\_review) = 1 032 146 895*

# OneClassSVM.py

**class** OneClassSVM:  
  
 @staticmethod  
 **def** get\_data(db):  
 row = db.cursor\_aspects.execute(**'SELECT** *\** **FROM Aspects'**).fetchone()  
 data = []  
 **while** row **is not None**: *# iterate through all reviews* aspect\_arr = []  
 **if** len(str(row[1])) != 0:  
 adv = str(row[1]).split(**";"**)  
 **for** item **in** adv:  
 aspect\_arr.append(item)  
 **if** len(str(row[2])) != 0:  
 dis = str(row[2]).split(**";"**)  
 **for** item **in** dis:  
 aspect\_arr.append(item)  
 **if** len(str(row[3])) != 0:  
 com = str(row[3]).split(**";"**)  
 **for** item **in** com:  
 aspect\_arr.append(item)  
 data.append(aspect\_arr)  
 row = db.cursor\_aspects.fetchone()  
 **return** data  
  
 @staticmethod  
 **def** get\_labels(data, db):  
 row\_review = db.cursor\_reviews.execute(**'SELECT** *\** **FROM Review'**).fetchone()  
 **import** os  
 path = os.getcwd()  
 train\_labels = []  
 count = 0  
 **while** row\_review **is not None**:  
 subcat\_name = str(row\_review[1])  
 file\_path = path + **"\\..\\productTrees\\Subcategories\\"** + subcat\_name + **".txt"** ideal\_labels = []  
 labels = []  
 **with** open(file\_path) **as** f:  
 ideal\_labels.append(f.readlines())  
 ideal\_labels[0][0] = ideal\_labels[0][0].lower()  
 **for** item **in** data[count]:  
 **if** item **in** ideal\_labels[0][0]:  
 labels.append(1)  
 **else**:  
 labels.append(-1)  
 count += 1  
 train\_labels.append(labels)  
 row\_review = db.cursor\_reviews.fetchone()  
 **return** train\_labels  
  
 @staticmethod  
 **def** get\_ideal\_data(data, labels):  
 ideal\_data = []  
 **for** i **in** range(len(labels)):  
 **if** labels[i] == 1:  
 ideal\_data.append(data[i])  
 **return** ideal\_data  
  
 @staticmethod  
 **def** unarray(data):  
 unarrayed\_data = []  
 **for** i **in** range(len(data)):  
 **for** item **in** data[i]:  
 unarrayed\_data.append(item)  
 **return** unarrayed\_data  
  
 @staticmethod  
 **def** train\_and\_predict(train\_data, test\_data):  
 **from** sklearn.feature\_extraction.text **import** TfidfVectorizer  
 vectorizer = TfidfVectorizer(min\_df=5,  
 max\_df=0.8,  
 sublinear\_tf=**True**,  
 use\_idf=**True**)  
 train\_vectors = vectorizer.fit\_transform(train\_data)  
 test\_vectors = vectorizer.transform(test\_data)  
 **from** sklearn **import** svm  
 classifier\_rbf = svm.OneClassSVM(nu=0.1, kernel=**"rbf"**, gamma=0.1)  
 classifier\_rbf.fit(train\_vectors)  
 prediction\_rbf = classifier\_rbf.predict(test\_vectors)  
 **return** prediction\_rbf

# PMI.py

**from** datetime **import** datetime  
  
**import** numpy **as** np  
**from** sklearn.feature\_extraction.text **import** CountVectorizer  
  
  
**class** PMI:  
 @staticmethod  
 **def** get\_all\_ideal\_aspects\_from\_train\_files():  
 **import** os  
 ideal\_aspects\_from\_file = {}  
 path = os.getcwd()  
 filenames = os.listdir(path + **"/../productTrees/Subcategories"**)  
 os.chdir(path + **"/../productTrees/Subcategories"**)  
 count = 0  
 **for** filename **in** filenames:  
 **with** open(filename) **as** f:  
 lines = f.readlines()  
 arr = lines[0].split(**";"**)  
 **for** val **in** arr:  
 val = val.replace(**" "**, **"\_"**).lower()  
 val = val.replace(**",\_"**, **"\_"**)  
 **if** val **not** in ideal\_aspects\_from\_file:  
 ideal\_aspects\_from\_file[val] = count  
 count += 1  
 **return** ideal\_aspects\_from\_file  
  
 @staticmethod  
 **def** iterate\_ideal\_aspects\_files(pmi, reviews\_corpus, sentences\_corpus, db):  
 **import** os  
 path = os.getcwd()  
 filenames = os.listdir(path + **"/../productTrees/Subcategories"**)  
 os.chdir(path + **"/../productTrees/Subcategories"**)  
 **for** filename **in** filenames:  
 print(filename)  
 count = 0  
 ideal\_aspects\_from\_file = {}  
 **with** open(filename) **as** f:  
 lines = f.readlines()  
 arr = lines[0].split(**";"**)  
 **for** val **in** arr:  
 val = val.replace(**" "**, **"\_"**).lower()  
 val = val.replace(**",\_"**, **"\_"**)  
 **if** val **not in** ideal\_aspects\_from\_file:  
 ideal\_aspects\_from\_file[val] = count  
 count += 1  
 *# ideal aspects from file + reviews* pmi.calculate\_pmi(reviews\_corpus, 2, ideal\_aspects\_from\_file, db)  
 db.conn\_pmi\_ideal\_review.commit()  
 *# ideal aspects from file + sentences* pmi.calculate\_pmi(sentences\_corpus, 3, ideal\_aspects\_from\_file, db)  
 db.conn\_pmi\_ideal\_sentence.commit()  
  
 @staticmethod  
 **def** get\_all\_reviews\_corpus(db):  
 reviews = []  
 row\_review = db.cursor\_reviews\_one\_word.execute(**'SELECT** *\** **FROM Reviews'**).fetchone()  
 **while** row\_review **is not None**:  
 adv = str(row\_review[1])  
 dis = str(row\_review[2])  
 com = str(row\_review[3])  
 review = adv + **" "** + dis + **" "** + com  
 reviews.append(review)  
 row\_review = db.cursor\_reviews\_one\_word.fetchone()  
 **return** reviews  
  
 @staticmethod  
 **def** get\_all\_sentences\_corpus(db):  
 sentences = []  
 row\_sentence = db.cursor\_sentences\_one\_word.execute(**'SELECT** *\** **FROM Sentences'**).fetchone()  
 **while** row\_sentence **is not None**:  
 sentence = str(row\_sentence[1])  
 sentences.append(sentence)  
 row\_sentence = db.cursor\_sentences\_one\_word.fetchone()  
 **return** sentences  
  
 @staticmethod  
 **def** get\_vocabulary(db):  
 vocabulary = {}  
 row = db.cursor\_aspects\_one\_word.execute(**'SELECT** *\** **FROM Aspects'**).fetchone()  
 count = 0  
 **while** row **is not None**:  
 adv = str(row[1]).strip()  
 **if** len(adv) != 0:  
 items = adv.split(**";"**)  
 **for** item **in** items:  
 **if** item **not in** vocabulary **and** len(item) > 0:  
 vocabulary[item] = count  
 count += 1  
 dis = str(row[2]).strip()  
 **if** len(dis) != 0:  
 items = dis.split(**";"**)  
 **for** item **in** items:  
 **if** item **not in** vocabulary **and** len(item) > 0:  
 vocabulary[item] = count  
 count += 1  
 com = str(row[3]).strip()  
 **if** len(com) != 0:  
 items = com.split(**";"**)  
 **for** item **in** items:  
 **if** item **not in** vocabulary **and** len(item) > 0:  
 vocabulary[item] = count  
 count += 1  
 row = db.cursor\_aspects\_one\_word.fetchone()  
 **return** vocabulary  
  
 @staticmethod  
 **def** get\_frequent\_vocabulary(db):  
 vocabulary = {}  
 count = 0  
 row\_aspect = db.cursor\_frequent.execute(**'SELECT** *\** **FROM Frequent'**).fetchone()  
 **while** row\_aspect **is not None**:  
 vocabulary[row\_aspect[0]] = count  
 row\_aspect = db.cursor\_frequent.fetchone()  
 count += 1  
 **return** vocabulary  
  
 **def** one\_word\_aspects(self, ideal, db):  
 row\_aspect = ideal.cursor\_aspects.execute(**'SELECT** *\** **FROM IdealAspects'**).fetchone()  
 count = 0  
 **while** row\_aspect **is not None**:  
 print(count)  
 count += 1  
 article = str(row\_aspect[0])  
 adv = str(row\_aspect[1])  
 list\_adv\_aspects = self.create\_one\_word\_list(adv)  
 dis = str(row\_aspect[2])  
 list\_dis\_aspects = self.create\_one\_word\_list(dis)  
 com = str(row\_aspect[3])  
 list\_com\_aspects = self.create\_one\_word\_list(com)  
 str\_adv\_aspects = **';'**.join(list\_adv\_aspects)  
 str\_dis\_aspects = **';'**.join(list\_dis\_aspects)  
 str\_com\_aspects = **';'**.join(list\_com\_aspects)  
 db.add\_one\_word\_aspects(article, str\_adv\_aspects, str\_dis\_aspects, str\_com\_aspects)  
 row\_aspect = ideal.cursor\_aspects.fetchone()  
 db.conn\_aspects\_one\_word.commit()  
  
 **def** one\_word\_reviews(self, db):  
 row = db.cursor\_reviews.execute(**'SELECT** *\** **FROM Review'**).fetchone()  
 row\_aspect = db.cursor\_aspects\_one\_word.execute(**'SELECT** *\** **FROM Aspects'**).fetchone()  
 count = 0  
 **while** row **is not None**:  
 print(count)  
 count += 1  
 article = str(row[2])  
 adv = str(row[3]).lower()  
 adv\_aspect = str(row\_aspect[1])  
 adv = self.process\_review(adv, adv\_aspect)  
 dis = str(row[4]).lower()  
 dis\_aspect = str(row\_aspect[2])  
 dis = self.process\_review(dis, dis\_aspect)  
 com = str(row[5]).lower()  
 com\_aspect = str(row\_aspect[3])  
 com = self.process\_review(com, com\_aspect)  
 db.add\_one\_word\_review(article, adv, dis, com)  
 row = db.cursor\_reviews.fetchone()  
 row\_aspect = db.cursor\_aspects\_one\_word.fetchone()  
 db.conn\_reviews\_one\_word.commit()  
  
 @staticmethod  
 **def** process\_review(part, aspects):  
 **if** len(aspects) != 0:  
 items = aspects.split(**";"**)  
 **for** item **in** items:  
 old\_words = item.split(**"\_"**)  
 **if** len(old\_words) > 1:  
 **for** word **in** old\_words:  
 part = part.replace(word, **""**, 1) *# remove the 1st entry of aspect word* **else**:  
 part = part.replace(old\_words[0], **""**, 1)  
 **if** len(part) == 0 **or** part[len(part) - 1] == **" " or** part[len(part) - 1] == **"\_"**:  
 part += item  
 **else**:  
 part += **" "** + item  
 **return** part  
  
 @staticmethod  
 **def** create\_one\_word\_list(part):  
 **if** len(part) != 0:  
 arr = []  
 items = part.split(**";"**)  
 **for** item **in** items:  
 words = item.split(**" "**)  
 **if** len(words) > 1:  
 arr.append(**"\_"**.join(words))  
 **else**:  
 arr.append(words[0])  
 **return** arr  
 **return ""  
  
 def** calculate\_pmi(self, corpus, which\_part, vocabulary, db):  
 vectorizer = CountVectorizer(min\_df=5, max\_df=0.8, vocabulary=vocabulary)  
 matrix = vectorizer.fit\_transform(corpus)  
 count = 0  
 matrix\_terms = np.array(vectorizer.get\_feature\_names()) *# unique aspects - keys* matrix\_freq = np.asarray(matrix.sum(axis=0)).ravel() *# number of each aspect* final\_matrix = np.array([matrix\_terms, matrix\_freq])  
 col\_array = self.create\_col\_array(matrix, len(matrix\_terms))  
 col\_array = np.array(col\_array)  
 **from** math **import** log  
 **for** i **in** range(len(matrix\_terms)):  
 print(count)  
 count += 1  
 start = datetime.now()  
 **for** j **in** range(i + 1, len(matrix\_terms)):  
 col1 = col\_array[i]  
 col2 = col\_array[j]  
 both\_num = np.count\_nonzero(col1 \* col2)  
 **if** both\_num == 0: *# independent* pmi\_val = 0  
 **else**:  
 pmi\_val = log(both\_num / (int(final\_matrix[1][i]) \* int(final\_matrix[1][j])))  
 **if** which\_part == 0:  
 db.add\_pmi\_review(matrix\_terms[i], matrix\_terms[j], final\_matrix[1][i], final\_matrix[1][j],  
 both\_num, pmi\_val)  
 **elif** which\_part == 1:  
 db.add\_pmi\_sentence(matrix\_terms[i], matrix\_terms[j], final\_matrix[1][i], final\_matrix[1][j],  
 both\_num, pmi\_val)  
 **elif** which\_part == 2:  
 db.add\_pmi\_ideal\_review(matrix\_terms[i], matrix\_terms[j], final\_matrix[1][i], final\_matrix[1][j],  
 both\_num, pmi\_val)  
 **else**:  
 db.add\_pmi\_ideal\_sentence(matrix\_terms[i], matrix\_terms[j], final\_matrix[1][i], final\_matrix[1][j],  
 both\_num, pmi\_val)  
 print(datetime.now() - start)  
 **if** count % 100 == 0:  
 db.conn\_pmi\_review.commit()  
 db.conn\_pmi\_sentence.commit()  
  
 @staticmethod  
 **def** create\_col\_array(matrix, matrix\_terms\_len):  
 array = []  
 **from** nltk.compat **import** xrange  
 **for** i **in** xrange(matrix\_terms\_len):  
 col = np.array(matrix[:, i].T.toarray())  
 array.append(col)  
 **return** array

# RandomForest.py

**from** sklearn.ensemble **import** RandomForestRegressor  
  
  
**class** RandomForest:  
 **def** \_\_init\_\_(self):  
 self.model = RandomForestRegressor(criterion=**'mse'**, max\_depth=8, min\_samples\_leaf=10, n\_estimators=250)  
  
 **def** process(self, x\_test, x\_train, y\_train): *# x\_train - characteristic(6 item), y\_train - semantic distance* self.model.fit(x\_train, y\_train)  
 *# from sklearn.externals import joblib  
 # joblib.dump(self.model, "train\_dump.pkl")  
 # self.model = joblib.load("train\_dump.pkl")* **return** self.model.predict(x\_test)  
  
 @staticmethod  
 **def** get\_train(db):  
 count = 0  
 x\_train = []  
 y\_train = []  
 row = db.cursor\_ideal\_full.execute(**'SELECT** *\** **FROM Ideal'**).fetchone()  
 **while** row **is not None**:  
 print(count)  
 count += 1  
 x\_train.append([row[3], row[4], row[5], row[6], row[7], row[8]])  
 y\_train.append(row[9])  
 row = db.cursor\_ideal\_full.fetchone()  
 **return** x\_train, y\_train  
  
 @staticmethod  
 **def** form\_train(db):  
 db.create\_ideal\_full\_db()  
 row\_review\_ideal = db.cursor\_pmi\_ideal\_review.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 row\_sentence\_ideal = db.cursor\_pmi\_ideal\_sentence.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 row\_lexical\_ideal = db.cursor\_lexical\_ideal.execute(**'SELECT** *\** **FROM Lexical'**).fetchone()  
 count = 0  
 row = db.cursor\_path\_weight.execute(**'SELECT** *\** **FROM Weight'**).fetchone()  
 **while** row\_review\_ideal **is not None**:  
 print(count)  
 count += 1  
 pmi\_review = float(row\_review\_ideal[5])  
 pmi\_sentence = float(row\_sentence\_ideal[5])  
 lexical = int(row\_lexical\_ideal[2])  
 aspect1 = row\_lexical\_ideal[0].replace(**"\_"**, **" "**)  
 aspect2 = row\_lexical\_ideal[1].replace(**"\_"**, **" "**)  
 *# local* row\_local\_ideal = db.cursor\_local\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect1, aspect2,)).fetchone()  
 **try**:  
 local\_context = float(row\_local\_ideal[2])  
 **except**: *# we can have (1,0) or (0,1) so need to find the correct one* row\_local\_ideal = db.cursor\_local\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect2, aspect1,)).fetchone()  
 local\_context = float(row\_local\_ideal[2])  
 *# global* row\_global\_ideal = db.cursor\_global\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect1, aspect2,)).fetchone()  
 **try**:  
 global\_context = float(row\_global\_ideal[2])  
 **except**: *# we can have (1,0) or (0,1) so need to find the correct one* row\_global\_ideal = db.cursor\_global\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect2, aspect1,)).fetchone()  
 global\_context = float(row\_global\_ideal[2])  
 *# syntactic* row\_syntactic\_ideal = db.cursor\_syntactic\_ideal.execute(  
 **'SELECT** *\** **FROM Syntactic WHERE aspect1 = ? AND aspect2 = ?'**,  
 (row\_lexical\_ideal[0], row\_lexical\_ideal[1],)).fetchone()  
 **try**:  
 syntactic = int(row\_syntactic\_ideal[2])  
 **except**: *# we can have (1,0) or (0,1) so need to find the correct one* row\_syntactic\_ideal = db.cursor\_syntactic\_ideal.execute(  
 **'SELECT** *\** **FROM Syntactic WHERE aspect1 = ? AND aspect2 = ?'**,  
 (row\_lexical\_ideal[1], row\_lexical\_ideal[0],)).fetchone()  
 syntactic = int(row\_syntactic\_ideal[2])  
 **if** syntactic == -1:  
 syntactic = 0  
 db.add\_ideal\_full(row[0], row\_lexical\_ideal[0], row\_lexical\_ideal[1], pmi\_review, pmi\_sentence, lexical, syntactic, local\_context, global\_context, row[3])  
 row = db.cursor\_path\_weight.fetchone()  
 row\_review\_ideal = db.cursor\_pmi\_ideal\_review.fetchone()  
 row\_sentence\_ideal = db.cursor\_pmi\_ideal\_sentence.fetchone()  
 row\_lexical\_ideal = db.cursor\_lexical\_ideal.fetchone()  
 db.conn\_ideal\_full.commit()  
  
 @staticmethod  
 **def** get\_test(db):  
 row\_review = db.cursor\_pmi\_review.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 row\_sentence = db.cursor\_pmi\_sentence.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 row\_lexical = db.cursor\_lexical.execute(**'SELECT** *\** **FROM Lexical'**).fetchone()  
 row\_syntactic = db.cursor\_syntactic.execute(**'SELECT** *\** **FROM Syntactic'**).fetchone()  
 row\_local = db.cursor\_local\_context.execute(**'SELECT** *\** **FROM Context'**).fetchone()  
 row\_global = db.cursor\_global\_context.execute(**'SELECT** *\** **FROM Context'**).fetchone()  
 count = 0  
 x\_test = []  
 **while** row\_review **is not None**:  
 print(count)  
 count += 1  
 pmi\_review = float(row\_review[5])  
 pmi\_sentence = float(row\_sentence[5])  
 lexical = int(row\_lexical[2])  
 syntactic = int(row\_syntactic[2])  
 local\_context = float(row\_local[2])  
 global\_context = float(row\_global[2])  
 x\_test.append([pmi\_review, pmi\_sentence, lexical, syntactic, local\_context, global\_context])  
 row\_review = db.cursor\_pmi\_review.fetchone()  
 row\_sentence = db.cursor\_pmi\_sentence.fetchone()  
 row\_lexical = db.cursor\_lexical.fetchone()  
 row\_local = db.cursor\_local\_context.fetchone()  
 row\_global = db.cursor\_global\_context.fetchone()  
 **return** x\_test

# SemanticDistanceLearning.py

**import** numpy **as** np  
  
  
**class** SemanticDistanceLearning:  
 **def** calculate\_ground\_truth\_distance(self, db):  
 **import** os  
 path = os.getcwd()  
 filenames = os.listdir(path + **"/../productTrees/Subcategories old"**)  
 os.chdir(path + **"/../productTrees/Subcategories"**)  
 all\_files\_content = []  
 **for** filename **in** filenames: *# load the aspects from all files* **with** open(filename) **as** f:  
 all\_files\_content.append(f.readlines())  
 os.chdir(path + **"/../productTrees/Subcategories old"**)  
 count = 0  
 **for** filename **in** filenames: *# iterate through all the files to calculate the path weights* f = open(filename)  
 print(filename)  
 file\_content = str(all\_files\_content[count]).split(**";"**) *# list with ideal aspects for concrete topic* file\_content[0] = file\_content[0][2:]  
 file\_content[len(file\_content) - 1] = file\_content[len(file\_content) - 1][  
 :len(file\_content[len(file\_content) - 1]) - 2]  
 **for** i **in** range(0, len(file\_content)):  
 node = file\_content[i]  
 **for** j **in** range(i + 1, len(file\_content)):  
 next\_node = file\_content[j]  
 path\_weight = self.find\_path(node, next\_node, filename) *# the min path weight between 2 words* db.add\_path\_weight(filename, node, next\_node, path\_weight)  
 db.conn\_path\_weight.commit()  
 count += 1  
 f.close()  
  
 @staticmethod  
 **def** find\_path(node, next\_node, filename):  
 **with** open(filename) **as** f:  
 content = f.readlines()  
 parent\_name = **""** parent\_name\_next = **""** deep\_num\_node = 0  
 deep\_num\_node\_next = 0  
 **for** line **in** content:  
 arr = line.split(**";"**)  
 word1 = arr[0]  
 word2 = arr[1]  
 deep\_num = int(arr[2].replace(**"\n"**, **""**))  
 **if** node == word1:  
 parent\_name = word2  
 deep\_num\_node = deep\_num  
 **if** next\_node == word1:  
 parent\_name\_next = word2  
 deep\_num\_node\_next = deep\_num  
 **if** len(parent\_name) == 0:  
 deep\_num\_node = 1  
 **if** len(parent\_name\_next) == 0:  
 deep\_num\_node\_next = 1  
 **if** filename.replace(**".txt"**, **""**) == next\_node:  
 deep\_num\_node\_next = 0  
 **if** filename.replace(**".txt"**, **""**) == node:  
 deep\_num\_node = 0  
 **if** parent\_name == parent\_name\_next:  
 **return** 2  
 **elif** parent\_name\_next == node:  
 **return** 1  
 **else**:  
 **return** deep\_num\_node + deep\_num\_node\_next  
  
 **def** calculate\_distance(self, db):  
 row\_review\_ideal = db.cursor\_pmi\_ideal\_review.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 row\_sentence\_ideal = db.cursor\_pmi\_ideal\_sentence.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 row\_lexical\_ideal = db.cursor\_lexical\_ideal.execute(**'SELECT** *\** **FROM Lexical'**).fetchone()  
 features\_arr = []  
 **while** row\_review\_ideal **is not None**:  
 pair\_features = []  
 pmi\_r = float(row\_review\_ideal[5])  
 pmi\_s = float(row\_sentence\_ideal[5])  
 lexical = int(row\_lexical\_ideal[2])  
 print(row\_lexical\_ideal[0] + **" "** + row\_lexical\_ideal[1])  
 aspect1 = row\_lexical\_ideal[0].replace(**"\_"**, **" "**)  
 aspect2 = row\_lexical\_ideal[1].replace(**"\_"**, **" "**)  
 *# local* row\_local\_ideal = db.cursor\_local\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect1, aspect2,)).fetchone()  
 **try**:  
 local\_context = float(row\_local\_ideal[2])  
 **except**: *# we can have (1,0) or (0,1) so need to find the correct one* row\_local\_ideal = db.cursor\_local\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect2, aspect1,)).fetchone()  
 local\_context = float(row\_local\_ideal[2])  
 *# global* row\_global\_ideal = db.cursor\_global\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect1, aspect2,)).fetchone()  
 **try**:  
 global\_context = float(row\_global\_ideal[2])  
 **except**: *# we can have (1,0) or (0,1) so need to find the correct one* row\_global\_ideal = db.cursor\_global\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect2, aspect1,)).fetchone()  
 global\_context = float(row\_global\_ideal[2])  
 *# syntactic* row\_syntactic\_ideal = db.cursor\_syntactic\_ideal.execute(  
 **'SELECT** *\** **FROM Syntactic WHERE aspect1 = ? AND aspect2 = ?'**,  
 (row\_lexical\_ideal[0], row\_lexical\_ideal[1],)).fetchone()  
 **try**:  
 syntactic = int(row\_syntactic\_ideal[2])  
 **except**: *# we can have (1,0) or (0,1) so need to find the correct one* row\_syntactic\_ideal = db.cursor\_syntactic\_ideal.execute(  
 **'SELECT** *\** **FROM Syntactic WHERE aspect1 = ? AND aspect2 = ?'**,  
 (row\_lexical\_ideal[1], row\_lexical\_ideal[0],)).fetchone()  
 syntactic = int(row\_syntactic\_ideal[2])  
 pair\_features.append(pmi\_r)  
 pair\_features.append(pmi\_s)  
 pair\_features.append(lexical)  
 pair\_features.append(syntactic)  
 pair\_features.append(local\_context)  
 pair\_features.append(global\_context)  
 features\_arr.append(pair\_features)  
 row\_review\_ideal = db.cursor\_pmi\_ideal\_review.fetchone()  
 row\_sentence\_ideal = db.cursor\_pmi\_ideal\_sentence.fetchone()  
 row\_lexical\_ideal = db.cursor\_lexical\_ideal.fetchone()  
 f = np.array(features\_arr) *# features's vector* d = np.array(self.vector\_with\_ground\_truth\_distances(db))  
 matrix\_size = 6 *# the features num* i = np.matrix(np.identity(matrix\_size)) *# identity metric* nu = 0.4  
 w = np.dot(np.power(np.dot(f.T, f) + nu \* i, -1), np.dot(f.T, d))  
 **return** w *# [ -5.18461604 -7.25855391 2.49603805 -1.122215 -2.29888273 41.45422735]* @staticmethod  
 **def** vector\_with\_ground\_truth\_distances(db):  
 row = db.cursor\_path\_weight.execute(**"SELECT** *\** **FROM Weight"**).fetchone()  
 vector = []  
 **while** row **is not None**:  
 vector.append(int(row[3]))  
 row = db.cursor\_path\_weight.fetchone()  
 **return** vector  
  
 **def** process\_semantic\_distance\_learning(self, db):  
 db.create\_semantic\_distance\_db()  
 row\_review = db.cursor\_pmi\_review.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 row\_sentence = db.cursor\_pmi\_sentence.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 row\_lexical = db.cursor\_lexical.execute(**'SELECT** *\** **FROM Lexical'**).fetchone()  
 row\_syntactic = db.cursor\_syntactic.execute(**'SELECT** *\** **FROM Syntactic'**).fetchone()  
 row\_local = db.cursor\_local\_context.execute(**'SELECT** *\** **FROM Context'**).fetchone()  
 row\_global = db.cursor\_global\_context.execute(**'SELECT** *\** **FROM Context'**).fetchone()  
 *# w = np.array(self.calculate\_distance(db))[0] # will return a vector with feature values  
 # w = [-7.17434844, -9.98536379, 2.89004706, -7.00041128, 102.75428241, 50.38338244, 22.32963088] # bayes* w = [-5.18461604, -7.25855391, 2.49603805, -1.122215, -2.29888273, 41.45422735]  
 count = 0  
 **while** row\_review **is not None**:  
 print(count)  
 count += 1  
 pmi\_review = float(row\_review[5])  
 pmi\_sentence = float(row\_sentence[5])  
 lexical = int(row\_lexical[2])  
 syntactic = int(row\_syntactic[2])  
 local\_context = float(row\_local[2])  
 global\_context = float(row\_global[2])  
 d = w[0] \* pmi\_review + w[1] \* pmi\_sentence + w[2] \* lexical + w[3] \* syntactic + w[4] \* local\_context + w[5] \* global\_context  
 db.add\_semantic\_distance(str(row\_review[0]), str(row\_review[1]), d)  
 row\_review = db.cursor\_pmi\_review.fetchone()  
 row\_sentence = db.cursor\_pmi\_sentence.fetchone()  
 row\_lexical = db.cursor\_lexical.fetchone()  
 row\_local = db.cursor\_local\_context.fetchone()  
 row\_global = db.cursor\_global\_context.fetchone()  
 **if** count % 1000 == 0:  
 db.conn\_semantic\_distance.commit()  
 db.conn\_semantic\_distance.commit()  
  
 **def** process\_semantic\_distance\_learning\_ideal(self, db):  
 db.create\_semantic\_distance\_ideal\_db()  
 row\_review\_ideal = db.cursor\_pmi\_ideal\_review.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 row\_sentence\_ideal = db.cursor\_pmi\_ideal\_sentence.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 row\_lexical\_ideal = db.cursor\_lexical\_ideal.execute(**'SELECT** *\** **FROM Lexical'**).fetchone()  
 *# w = np.array(self.calculate\_distance(db))[0] # will return a vector with feature values* w = [-7.17434844, -9.98536379, 2.89004706, -7.00041128, 102.75428241, 50.38338244, 22.32963088] *# bayes* w = [-5.18461604, -7.25855391, 2.49603805, -1.122215, -2.29888273, 41.45422735] *# 0.4  
 # [-5.18461655 - 7.25855449 2.49603811 - 1.12222121 - 2.20044153 41.45442079] 0  
 # [ -5.18460385 -7.25854018 2.49603675 -1.12206601 -4.40758755 41.44959215] 10* count = 0  
 **while** row\_review\_ideal **is not None**:  
 print(count)  
 count += 1  
 pmi\_review = float(row\_review\_ideal[5])  
 pmi\_sentence = float(row\_sentence\_ideal[5])  
 lexical = int(row\_lexical\_ideal[2])  
 aspect1 = row\_lexical\_ideal[0].replace(**"\_"**, **" "**)  
 aspect2 = row\_lexical\_ideal[1].replace(**"\_"**, **" "**)  
 *# local* row\_local\_ideal = db.cursor\_local\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect1, aspect2,)).fetchone()  
 **try**:  
 local\_context = float(row\_local\_ideal[2])  
 **except**: *# we can have (1,0) or (0,1) so need to find the correct one* row\_local\_ideal = db.cursor\_local\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect2, aspect1,)).fetchone()  
 local\_context = float(row\_local\_ideal[2])  
 *# global* row\_global\_ideal = db.cursor\_global\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect1, aspect2,)).fetchone()  
 **try**:  
 global\_context = float(row\_global\_ideal[2])  
 **except**: *# we can have (1,0) or (0,1) so need to find the correct one* row\_global\_ideal = db.cursor\_global\_context\_ideal.execute(  
 **'SELECT** *\** **FROM Context WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect2, aspect1,)).fetchone()  
 global\_context = float(row\_global\_ideal[2])  
 *# syntactic* row\_syntactic\_ideal = db.cursor\_syntactic\_ideal.execute(  
 **'SELECT** *\** **FROM Syntactic WHERE aspect1 = ? AND aspect2 = ?'**,  
 (row\_lexical\_ideal[0], row\_lexical\_ideal[1],)).fetchone()  
 **try**:  
 syntactic = int(row\_syntactic\_ideal[2])  
 **except**: *# we can have (1,0) or (0,1) so need to find the correct one* row\_syntactic\_ideal = db.cursor\_syntactic\_ideal.execute(  
 **'SELECT** *\** **FROM Syntactic WHERE aspect1 = ? AND aspect2 = ?'**,  
 (row\_lexical\_ideal[1], row\_lexical\_ideal[0],)).fetchone()  
 syntactic = int(row\_syntactic\_ideal[2])  
 **if** syntactic == -1:  
 syntactic = 0  
 d = w[0] \* pmi\_review + w[1] \* pmi\_sentence + w[2] \* lexical + w[3] \* syntactic + w[4] \* local\_context + w[  
 5] \* global\_context  
 db.add\_semantic\_distance\_ideal(row\_lexical\_ideal[0], row\_lexical\_ideal[1], d)  
 row\_review\_ideal = db.cursor\_pmi\_ideal\_review.fetchone()  
 row\_sentence\_ideal = db.cursor\_pmi\_ideal\_sentence.fetchone()  
 row\_lexical\_ideal = db.cursor\_lexical\_ideal.fetchone()  
 db.conn\_semantic\_distance\_ideal.commit()  
  
 @staticmethod  
 **def** create\_semantic\_distance\_random\_forest\_classifier(db, y\_test):  
 db.create\_semantic\_distance\_real\_db()  
 row\_review = db.cursor\_pmi\_review.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 count = 0  
 **while** row\_review **is not None**:  
 print(count)  
 aspect1 = row\_review[0]  
 aspect2 = row\_review[1]  
 db.add\_semantic\_distance\_real(aspect1, aspect2, y\_test[count])  
 row\_review = db.cursor\_pmi\_review.fetchone()  
 count += 1  
 db.conn\_semantic\_distance\_real.commit()  
  
 @staticmethod  
 **def** print\_data(db):  
 row\_path\_weight = db.cursor\_path\_weight.execute(**'SELECT** *\** **FROM Weight'**).fetchone()  
 print(**"aspect1\t\taspect2\t\tsemantic\tweight"**)  
 **while** row\_path\_weight **is not None**:  
 **from** aspects.Context **import** Context  
 aspect1 = Context.replacer(row\_path\_weight[1]).replace(**" "**, **"\_"**)  
 aspect2 = Context.replacer(row\_path\_weight[2]).replace(**" "**, **"\_"**)  
 row\_semantic = db.cursor\_semantic\_distance\_ideal.execute(  
 **'SELECT** *\** **FROM Distance WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect1, aspect2,)).fetchone()  
 **if** row\_semantic **is None**:  
 row\_semantic = db.cursor\_semantic\_distance\_ideal.execute(  
 **'SELECT** *\** **FROM Distance WHERE aspect1 = ? AND aspect2 = ?'**,  
 (aspect2, aspect1,)).fetchone()  
 print(**'{:s}\t{:s}\t{:4.2f}\t{:4.2f}'**.format(row\_semantic[0], row\_semantic[1], row\_semantic[2],  
 float(row\_path\_weight[3])))  
 row\_path\_weight = db.cursor\_path\_weight.fetchone()

# Sentence.py

**import** json  
  
**import** requests  
**import** re  
  
  
**class** Sentence:  
 **def** process(self, db, aspect):  
 row\_review = db.cursor\_reviews.execute(**'SELECT** *\** **FROM Review'**).fetchone()  
 count = 0  
 **while** row\_review **is not None**: *# iterate through all reviews* print(count)  
 count += 1  
 article = str(row\_review[2])  
 adv = str(row\_review[3]).strip()  
 review = **""  
 if** len(adv) != 0:  
 review += adv  
 **if** adv[len(adv) - 1] != **"." and** adv[len(adv) - 1] != **"!" and** adv[len(adv) - 1] != **"?"**:  
 review += **"."** dis = str(row\_review[4]).strip()  
 **if** len(dis) != 0:  
 **if** len(review) != 0:  
 review += **" "** review += dis  
 **if** dis[len(dis) - 1] != **"." and** dis[len(dis) - 1] != **"!" and** dis[len(dis) - 1] != **"?"**:  
 review += **"."** com = str(row\_review[5]).strip()  
 **if** len(com) != 0:  
 **if** len(review) != 0:  
 review += **" "** review += com  
 **if** com[len(com) - 1] != **"." and** com[len(com) - 1] != **"!" and** com[len(com) - 1] != **"?"**:  
 review += **"."** sentences\_from\_api = self.ask\_api(review.lower(), aspect)  
 sentences = self.clean\_sentences(sentences\_from\_api)  
 **for** sentence **in** sentences:  
 **if** sentence != **"."**:  
 db.add\_sentence(article, sentence)  
 row\_review = db.cursor\_reviews.fetchone()  
 db.conn\_sentence.commit()  
  
 **def** process\_one\_word(self, db, aspect):  
 row\_review = db.cursor\_reviews.execute(**'SELECT** *\** **FROM Review'**).fetchone()  
 row\_aspect = db.cursor\_aspects\_one\_word.execute(**'SELECT** *\** **FROM Aspects'**).fetchone()  
 count = 0  
 **while** row\_review **is not None**: *# iterate through all reviews* print(count)  
 count += 1  
 article = str(row\_review[2])  
 review = **""** adv = str(row\_review[3]).strip()  
 adv\_aspect = str(row\_aspect[1])  
 **if** len(adv) != 0:  
 **if** adv[len(adv) - 1] == **"." or** adv[len(adv) - 1] == **"!" or** adv[len(adv) - 1] == **"?"**:  
 adv = adv[0:len(adv) - 1]  
 adv = self.process\_review(adv, adv\_aspect)  
 **if** len(adv) != 0:  
 **if** len(review) != 0:  
 review += **" "** review += adv  
 **if** adv[len(adv) - 1] != **"." and** adv[len(adv) - 1] != **"!" and** adv[len(adv) - 1] != **"?"**:  
 review += **"."** dis = str(row\_review[4]).strip()  
 dis\_aspect = str(row\_aspect[2])  
 **if** len(dis) != 0:  
 **if** dis[len(dis) - 1] == **"." or** dis[len(dis) - 1] == **"!" or** dis[len(dis) - 1] == **"?"**:  
 dis = dis[0:len(dis) - 1]  
 dis = self.process\_review(dis, dis\_aspect)  
 **if** len(dis) != 0:  
 **if** len(review) != 0:  
 review += **" "** review += dis  
 **if** dis[len(dis) - 1] != **"." and** dis[len(dis) - 1] != **"!" and** dis[len(dis) - 1] != **"?"**:  
 review += **"."** com = str(row\_review[5]).strip()  
 com\_aspect = str(row\_aspect[3])  
 **if** len(com) != 0:  
 **if** com[len(com) - 1] == **"." or** com[len(com) - 1] == **"!" or** com[len(com) - 1] == **"?"**:  
 com = com[0:len(com) - 1]  
 com = self.process\_review(com, com\_aspect)  
 **if** len(com) != 0:  
 **if** len(review) != 0:  
 review += **" "** review += com  
 **if** com[len(com) - 1] != **"." and** com[len(com) - 1] != **"!" and** com[len(com) - 1] != **"?"**:  
 review += **"."** review = review.upper()  
 review = re.sub(**' +'**, **' '**, review)  
 sentences\_from\_api = self.ask\_api(review, aspect)  
 sentences = self.clean\_sentences(sentences\_from\_api)  
 **for** sentence **in** sentences:  
 **if** sentence != **"."**:  
 db.add\_one\_word\_sentence(article, sentence.lower())  
 row\_review = db.cursor\_reviews.fetchone()  
 row\_aspect = db.cursor\_aspects\_one\_word.fetchone()  
 db.conn\_sentences\_one\_word.commit()  
  
 @staticmethod  
 **def** process\_review(part, aspects):  
 **if** len(aspects) != 0:  
 items = aspects.split(**";"**)  
 **for** item **in** items:  
 old\_words = item.split(**"\_"**)  
 **if** len(old\_words) > 1:  
 **for** word **in** old\_words:  
 rep = re.compile(re.escape(word), re.IGNORECASE)  
 part = rep.sub(**""**, part, 1) *# remove the 1st entry of aspect word* **else**:  
 rep = re.compile(re.escape(old\_words[0]), re.IGNORECASE)  
 part = rep.sub(**""**, part, 1)  
 **if** len(part) == 0 **or** part[len(part) - 1] == **" " or** part[len(part) - 1] == **"\_"**:  
 part += item  
 **else**:  
 part += **" "** + item  
 **return** part  
  
 @staticmethod  
 **def** clean\_sentences(sentences\_from\_api):  
 result = json.loads(sentences\_from\_api)  
 sentence\_items = result[**'annotations'**][**'sentence'**]  
 sentences = []  
 **for** item **in** sentence\_items:  
 text = result[**'text'**][item[**'start'**]:item[**'end'**]]  
 sentences.append(text)  
 **return** sentences  
  
 @staticmethod  
 **def** ask\_api(review, aspect):  
 payload = {**'text'**: str(review)}  
 headers = {**'Accept'**: **'application/json'**}  
 r = requests.post(aspect.url\_sentence, data=payload, headers=headers)  
 **while** r.status\_code != 200:  
 r = requests.post(aspect.url\_pos, data=payload, headers=headers)  
 **return** r.content.decode(**'utf8'**)

# Splitter.py

**class** Splitter:  
 **def** process\_reviews(self, db):  
 row\_review = db.cursor\_reviews\_one\_word.execute(**'SELECT** *\** **FROM Reviews'**).fetchone()  
 count = 0  
 **while** row\_review **is not None**:  
 print(count)  
 count += 1  
 adv\_before = str(row\_review[1])  
 adv = self.clean(adv\_before)  
 adv = **" "**.join(adv.split())  
 **if** adv\_before != adv:  
 db.cursor\_reviews\_one\_word\_update.execute(  
 **'UPDATE Reviews SET advantageAspects = ? WHERE advantageAspects = ?'**,  
 (adv, adv\_before,))  
 db.conn\_reviews\_one\_word.commit()  
 dis\_before = str(row\_review[2])  
 dis = self.clean(dis\_before)  
 dis = **" "**.join(dis.split())  
 **if** dis\_before != dis:  
 db.cursor\_reviews\_one\_word\_update.execute(  
 **'UPDATE Reviews SET disadvantageAspects = ? WHERE disadvantageAspects = ?'**,  
 (dis, dis\_before,))  
 db.conn\_reviews\_one\_word.commit()  
 com\_before = str(row\_review[3])  
 com = self.clean(com\_before)  
 com = **" "**.join(com.split())  
 **if** com\_before != com:  
 db.cursor\_reviews\_one\_word\_update.execute(  
 **'UPDATE Reviews SET commentAspects = ? WHERE commentAspects = ?'**,  
 (com, com\_before,))  
 db.conn\_reviews\_one\_word.commit()  
 row\_review = db.cursor\_reviews\_one\_word.fetchone()  
  
 **def** process\_sentences(self, db):  
 row\_sentence = db.cursor\_sentences\_one\_word.execute(**'SELECT** *\** **FROM Sentences'**).fetchone()  
 count = 0  
 **while** row\_sentence **is not None**:  
 print(count)  
 count += 1  
 sentence\_before = str(row\_sentence[1])  
 sentence = self.clean(sentence\_before)  
 sentence = **" "**.join(sentence.split())  
 **if** sentence\_before != sentence:  
 db.cursor\_sentences\_one\_word\_update.execute(  
 **'UPDATE Sentences SET sentence = ? WHERE sentence = ?'**,  
 (sentence, sentence\_before,))  
 db.conn\_reviews\_one\_word.commit()  
 row\_sentence = db.cursor\_sentences\_one\_word.fetchone()  
  
 @staticmethod  
 **def** clean(part):  
 new\_part = **""** words = part.strip().split(**" "**)  
 words = filter(**None**, words)  
 **for** word **in** words:  
 **if** word[0] == **"\_"**:  
 **if** len(word) == 1:  
 word = **""  
 else**:  
 word = word[1:]  
 **if** len(word) > 0 **and** word[len(word) - 1] == **"\_"**:  
 word = word[0:len(word) - 1]  
 **if** word.isdigit():  
 word = **""  
 if** word.count(**"\_"**) > 1:  
 under\_words = word.split(**"\_"**)  
 **if** len(under\_words) == 3:  
 new\_str = under\_words[0] + **"\_"** + under\_words[1] + **" "** + under\_words[1] + **"\_"** + under\_words[2]  
 **elif** len(under\_words) == 4:  
 new\_str = under\_words[0] + **"\_"** + under\_words[1] + **" "** + under\_words[1] + **"\_"** + under\_words[  
 2] + **" "** + under\_words[2] + **"\_"** + under\_words[3]  
 **elif** len(under\_words) == 7:  
 new\_str = under\_words[0] + **"\_"** + under\_words[1] + **" "** + under\_words[1] + **"\_"** + under\_words[  
 2] + **" "** + under\_words[2] + **"\_"** + under\_words[3] + **" "** + under\_words[3] + **"\_"** + under\_words[  
 4] + **" "** + under\_words[4] + **"\_"** + under\_words[5] + **" "** + under\_words[5] + **"\_"** + \  
 under\_words[6]  
 **else**:  
 new\_str = **""** new\_part += new\_str + **" "  
 else**:  
 new\_part += word + **" "  
 return** new\_part.strip()

# Syntactic.py

**import** json  
**from** datetime **import** datetime  
**from** operator **import** itemgetter  
  
**import** numpy **as** np  
**from** sklearn.feature\_extraction.text **import** CountVectorizer  
  
**from** aspects.Aspects **import** Aspects  
**from** aspects.PMI **import** PMI  
  
**import** re  
  
  
**class** Syntactic:  
 **def** process(self, db, vocabulary, aspect\_class\_object):  
 db.create\_syntactic\_db()  
 *# db.create\_tree\_db() # load syntactic trees from api ispras* corpus = PMI.get\_all\_sentences\_corpus(db)  
 *# self.build\_tree(corpus, aspect\_class\_object, db)* vectorizer = CountVectorizer(min\_df=5, max\_df=0.8, vocabulary=vocabulary)  
 matrix = vectorizer.fit\_transform(corpus)  
 matrix\_terms = np.array(vectorizer.get\_feature\_names()) *# unique aspects - keys* col\_array = PMI.create\_col\_array(matrix, len(matrix\_terms))  
 col\_array = np.array(col\_array)  
 **for** i **in** range(len(matrix\_terms)):  
 start = datetime.now()  
 **for** j **in** range(i + 1, len(matrix\_terms)):  
 print(j)  
 col1 = col\_array[i]  
 col2 = col\_array[j]  
 non\_zero\_sentences\_indexes = np.nonzero(col1 \* col2)[1]  
 **if** len(non\_zero\_sentences\_indexes) == 0: *# independent* syntactic = -1  
 **else**:  
 non\_zero\_sentences = itemgetter(\*non\_zero\_sentences\_indexes)(corpus)  
 **if** len(non\_zero\_sentences\_indexes) == 1:  
 syntactic = self.find\_path\_for\_sentence(non\_zero\_sentences, matrix\_terms[i], matrix\_terms[j], 0,  
 1, db)  
 **else**:  
 syntactic = self.calculate\_syntactic(matrix\_terms[i], matrix\_terms[j], non\_zero\_sentences, db)  
 db.add\_syntactic(matrix\_terms[i], matrix\_terms[j], syntactic)  
 print(datetime.now() - start)  
 **if** i % 100 == 0:  
 db.conn\_syntactic.commit()  
 db.conn\_syntactic.commit()  
  
 **def** process\_ideal(self, db):  
 db.create\_syntactic\_ideal\_db()  
 corpus = PMI.get\_all\_sentences\_corpus(db)  
 row\_ideal = db.cursor\_pmi\_ideal\_review.execute(**'SELECT** *\** **FROM PMI'**).fetchone()  
 vocabulary = []  
 **while** row\_ideal **is not None**:  
 aspect1 = str(row\_ideal[0])  
 aspect2 = str(row\_ideal[1])  
 **if** aspect1 **not in** vocabulary:  
 vocabulary.append(aspect1)  
 **if** aspect2 **not in** vocabulary:  
 vocabulary.append(aspect2)  
 row\_ideal = db.cursor\_pmi\_ideal\_review.fetchone()  
 vectorizer = CountVectorizer(min\_df=5, max\_df=0.8, vocabulary=vocabulary)  
 matrix = vectorizer.fit\_transform(corpus)  
 matrix\_terms = np.array(vectorizer.get\_feature\_names()) *# unique aspects - keys* col\_array = PMI.create\_col\_array(matrix, len(matrix\_terms))  
 **for** i **in** range(len(matrix\_terms)):  
 print(i)  
 start = datetime.now()  
 **for** j **in** range(i + 1, len(matrix\_terms)):  
 print(j)  
 col1 = col\_array[i]  
 col2 = col\_array[j]  
 non\_zero\_sentences\_indexes = np.nonzero(col1 \* col2)[1]  
 **if** len(non\_zero\_sentences\_indexes) == 0: *# independent* syntactic = -1  
 **else**:  
 non\_zero\_sentences = itemgetter(\*non\_zero\_sentences\_indexes)(corpus)  
 **if** len(non\_zero\_sentences\_indexes) == 1:  
 syntactic = self.find\_path\_for\_sentence(non\_zero\_sentences, matrix\_terms[i], matrix\_terms[j], 0,  
 1, db)  
 **else**:  
 syntactic = self.calculate\_syntactic(matrix\_terms[i], matrix\_terms[j], non\_zero\_sentences, db)  
 db.add\_syntactic\_ideal(matrix\_terms[i], matrix\_terms[j], syntactic)  
 print(datetime.now() - start)  
 db.conn\_syntactic\_ideal.commit()  
  
 **def** calculate\_syntactic(self, aspect1, aspect2, non\_zero\_sentences, db):  
 divider = len(non\_zero\_sentences)  
 syntactic\_paths\_sum = 0  
 dict = {}  
 **for** sentence **in** non\_zero\_sentences:  
 **if** sentence **in** dict:  
 syntactic\_paths\_sum += dict[sentence]  
 **else**:  
 sum\_before = syntactic\_paths\_sum  
 syntactic\_paths\_sum = self.find\_path\_for\_sentence(sentence, aspect1, aspect2, syntactic\_paths\_sum, divider,  
 db)  
 dict[sentence] = syntactic\_paths\_sum - sum\_before  
 **return** syntactic\_paths\_sum / divider  
  
 **def** find\_path(self, aspect1, aspect2, aspect1\_parent, aspect2\_parent, syntax\_relations, aspect1\_parents,  
 aspect2\_parents):  
 **while True**:  
 **if** aspect1\_parents[len(aspect1\_parents) - 1] == aspect2\_parents[  
 len(aspect2\_parents) - 1]: *# aspect parents have just intersected* **return** len(aspect1\_parents) + len(aspect2\_parents)  
 **if** aspect2\_parent **in** aspect1\_parents: *# next aspect2 parent has intersection with list of aspect1 parents* index = aspect1\_parents.index(aspect2\_parent) + 1  
 **return** index + len(aspect2\_parents)  
 **if** aspect1\_parent **in** aspect2\_parents: *# next aspect1 parent has intersection with list of aspect2 parents* index = aspect2\_parents.index(aspect1\_parent) + 1  
 **return** index + len(aspect1\_parents)  
 **if** aspect1\_parent == aspect2: *# the next aspect1 parent is out aspect2* **return** len(aspect1\_parents)  
 **if** aspect2\_parent == aspect1: *# the next aspect2 parent is out aspect1* **return** len(aspect2\_parents)  
 **if** aspect1\_parent == **"" and** aspect2\_parent == **""**: *# the words don't have intersection at all* **return** max(len(aspect1\_parents), len(aspect2\_parents))  
 **try**:  
 **if** aspect1\_parent != **""**:  
 aspect1\_parent = self.get\_parent(aspect1\_parent, syntax\_relations)[**"value"**][**"parent"**][**"start"**]  
 aspect1\_parents.append(aspect1\_parent)  
 **except**:  
 aspect1\_parent = **""** *# there is no next parent* **try**:  
 **if** aspect2\_parent != **""**:  
 aspect2\_parent = self.get\_parent(aspect2\_parent, syntax\_relations)[**"value"**][**"parent"**][**"start"**]  
 aspect2\_parents.append(aspect2\_parent)  
 **except**:  
 aspect2\_parent = **""** *# there is no next parent* @staticmethod  
 **def** get\_parent(start\_index, syntax\_relations):  
 **for** relation **in** syntax\_relations:  
 **if** start\_index == relation[**"start"**]:  
 **return** relation  
  
 **def** find\_path\_for\_sentence(self, sentence, aspect1, aspect2, syntactic\_paths\_sum, divider, db):  
 row\_syntactic\_tree = db.cursor\_tree.execute(**'SELECT** *\** **FROM Tree WHERE sentence = ?'**, (sentence,)).fetchone()  
 data = json.loads(str(row\_syntactic\_tree[1]))  
 aspect\_error\_happened = **False** syntax\_relations = data[**'annotations'**][**'syntax-relation'**]  
 **try**:  
 aspect1\_start\_index = re.search(**r'\b(%s)\b'** % aspect1, sentence).start()  
 **except**:  
 aspect1\_start\_index = 0  
 aspect\_error\_happened = **True  
 try**:  
 aspect2\_start\_index = re.search(**r'\b(%s)\b'** % aspect2, sentence).start()  
 **except**:  
 aspect2\_start\_index = 0  
 aspect\_error\_happened = **True** aspect1\_parents = []  
 aspect2\_parents = []  
 **try**:  
 aspect1\_parent = self.get\_parent(aspect1\_start\_index, syntax\_relations)[**"value"**][**"parent"**][**"start"**]  
 **except**:  
 aspect1\_parent = aspect1\_start\_index - 1 *# trying to fix the error later in method* aspect1\_parents.append(aspect1\_parent)  
 **try**:  
 aspect2\_parent = self.get\_parent(aspect2\_start\_index, syntax\_relations)[**"value"**][**"parent"**][**"start"**]  
 **except**:  
 aspect2\_parent = aspect2\_start\_index - 1 *# trying to fix the error later in method* aspect2\_parents.append(aspect2\_parent)  
 **if** aspect1\_parent == aspect2\_start\_index **or** aspect2\_parent == aspect1\_start\_index:  
 **if not** aspect\_error\_happened: *# one aspect is parent to another* syntactic\_paths\_sum += 1  
 **else**:  
 divider -= 1  
 **else**:  
 **if not** aspect\_error\_happened: *# don't need the results which was incorrectly included because of sklearn lib* syntactic\_paths\_sum += self.find\_path(aspect1\_start\_index, aspect2\_start\_index, aspect1\_parent,  
 aspect2\_parent, syntax\_relations, aspect1\_parents,  
 aspect2\_parents)  
 **else**:  
 divider -= 1  
 **return** syntactic\_paths\_sum  
  
 @staticmethod  
 **def** build\_tree(sentences, aspect\_class\_object, db):  
 **for** sentence **in** sentences:  
 row\_sentence = db.cursor\_tree.execute(**'SELECT** *\** **FROM Tree WHERE sentence = ?'**, (sentence,)).fetchone()  
 **if** row\_sentence **is None**: *# the sentence is not in db* syntactic\_tree = Aspects.syntatic\_parsing(sentence, aspect\_class\_object)  
 db.add\_tree(sentence, syntactic\_tree)  
 db.conn\_tree.commit()

# Unnecessary.py

**from** aspects.DB **import** DB  
**from** aspects.IdealAspectsDB **import** IdealAspectsDB  
  
  
**def** updater(part):  
 **if** len(part) != 0:  
 arr = part.split(";")  
 new\_aspect = **""  
 for** item **in** arr:  
 **if "http" not in** item:  
 new\_aspect += item + **";"** new\_aspect = new\_aspect[0:len(new\_aspect) - 1]  
 **return** new\_aspect  
 **return ''**ideal = IdealAspectsDB() *# aspects data base*row\_aspect = ideal.cursor\_aspects.execute(**'SELECT** *\** **FROM IdealAspects'**).fetchone()  
count = 0  
**while** row\_aspect **is not None**: *# iterate through all reviews* print(count)  
 count += 1  
 article = str(row\_aspect[0])  
 adv = str(row\_aspect[1])  
 dis = str(row\_aspect[2])  
 com = str(row\_aspect[3])  
 new\_adv = updater(adv)  
 **if** new\_adv != adv:  
 ideal.cursor\_aspects\_update.execute(  
 **'UPDATE IdealAspects SET advantageAspects = ? WHERE article = ? and advantageAspects = ?'**,  
 (new\_adv, article, adv,))  
 ideal.conn\_aspects.commit()  
 new\_dis = updater(dis)  
 **if** new\_dis != dis:  
 ideal.cursor\_aspects\_update.execute(  
 **'UPDATE IdealAspects SET disadvantageAspects = ? WHERE article = ? and disadvantageAspects = ?'**,  
 (new\_dis, article, dis,))  
 ideal.conn\_aspects.commit()  
 new\_com = updater(com)  
 **if** new\_com != com:  
 ideal.cursor\_aspects\_update.execute(  
 **'UPDATE IdealAspects SET commentAspects = ? WHERE article = ? and commentAspects = ?'**,  
 (new\_com, article, com,))  
 ideal.conn\_aspects.commit()  
 row\_aspect = ideal.cursor\_aspects.fetchone()

# CategoryNames.py

**import** sqlite3  
  
  
**class** CategoryNames:  
 conn = **None** *# database* c = **None** *# cursor* dbName = **'Review\_Ulmart.db'** *# constructor* **def** \_\_init\_\_(self):  
 **import** os  
 path = os.getcwd()  
 self.conn = sqlite3.connect(path + **"\\..\\db\\"** + self.dbName)  
 self.c = self.conn.cursor()  
  
 **def** site\_directories\_print(self):  
 self.c.execute(**'SELECT DISTINCT category\_name FROM Review'**)  
 row = self.c.fetchone()  
 **while** row **is not None**:  
 print(str(row[0]))  
 row = self.c.fetchone()  
  
 **def** sub\_categories\_print(self):  
 self.c.execute(**'SELECT DISTINCT subcategory\_name FROM Review'**)  
 row = self.c.fetchone()  
 **while** row **is not None**:  
 print(str(row[0]))  
 self.create(str(row[0]))  
 row = self.c.fetchone()  
  
 **def** create(self, name):  
 **try**:  
 name = **"D:\courseWork2016\productTrees\Subcategories files\\"** + name + **".txt"** file = open(name, **'a'**)  
 file.close()  
 **except**:  
 print(**"error occured"**)  
  
  
category\_names = CategoryNames()  
category\_names.site\_directories\_print()  
category\_names.sub\_categories\_print()

# DataBase\_Ulmart.py

**import** sqlite3  
  
  
**class** DataBase\_Ulmart:  
 conn = **None** *# database* c = **None** *# cursor* dbName = **'Review\_Ulmart.db'** *# constructor* **def** \_\_init\_\_(self):  
 **import** os  
 path = os.getcwd()  
 self.conn = sqlite3.connect(path + **"\\..\\db\\"** + self.dbName)  
 self.c = self.conn.cursor()  
 self.create\_db()  
  
 *# Create table* **def** create\_db(self):  
 self.c.execute(**'''CREATE TABLE IF NOT EXISTS Review  
 (category\_name TEXT, subcategory\_name TEXT, article TEXT, advantage TEXT, disadvantage TEXT, comment TEXT)'''**)  
 self.commit()  
  
 *# Insert new review to DB* **def** add\_review(self, category\_name, subcategory\_name, article, advantage, disadvantage, comment):  
 self.c.execute(  
 **'INSERT INTO Review (category\_name, subcategory\_name, article, advantage, disadvantage, comment) '  
 'VALUES (?, ?, ?, ?, ?, ?)'**,  
 (category\_name, subcategory\_name, article, advantage, disadvantage, comment))  
 self.commit()  
  
  
 **def** reviews\_num(self, article):  
 review\_num = self.c.execute(**'SELECT** *COUNT***(***\****) FROM Review WHERE article = '** + str(article))  
 **return** review\_num.fetchone()  
  
 **def** total(self):  
 total\_num = self.c.execute(**'SELECT** *COUNT***(***\****) FROM Review'**)  
 **return** total\_num.fetchone()  
  
 *# delete the review* **def** remove\_review(self, article):  
 self.c.execute(**'DELETE FROM Review WHERE article = '** + str(article))  
 self.commit()  
  
 *# destructor - close connection* **def** \_\_del\_\_(self):  
 self.conn.close()  
  
 *# commit* **def** commit(self):  
 self.conn.commit()

# Parser.py

**import** codecs  
**import** urllib.request  
  
**from** bs4 **import** BeautifulSoup  
  
**from** ulmart.DataBase\_Ulmart **import** DataBase\_Ulmart  
  
  
**class** Parser:  
  
 @staticmethod  
 **def** parse\_product(start\_link, category\_name, subcategory\_name):  
 **with** urllib.request.urlopen(start\_link) **as** url:  
 html = url.read()  
 soup = BeautifulSoup(html, **"lxml"**)  
 products\_num = soup.find(**"span"**, {**"class"**: **"text-sm text-muted"**}).findAll(**"span"**)[1].text *# get the products number* **if** int(products\_num) <= 30: *# count the pages number (30 items per page)* pages\_num = 1  
 **else**:  
 pages\_num = round(int(products\_num) / 30)  
 nums = [x **for** x **in** range(1, pages\_num + 1)]  
 last\_slash\_num = start\_link.rfind(**"/"**)  
 review\_url = **"https://www.ulmart.ru/catalogAdditional"** + start\_link[last\_slash\_num:] + **"?pageNum="  
 for** num **in** nums: *# look through all pages with products kept on page with start\_link* new\_url = review\_url + str(num)  
 **with** urllib.request.urlopen(new\_url) **as** url: *# load products for a concrete page* html = url.read()  
 soup = BeautifulSoup(html, **"lxml"**)  
 all\_products = soup.findAll(**"div"**, {**"class"**: **"b-product\_\_art"**})  
 **for** title **in** all\_products: *# look through all products on a concrete page* article = title.find(**"span"**).text  
 completed = **False  
 while not** completed: *# get the reviews for each product item with unique article* **try**:  
 completed = parser.reviews(article, category\_name, subcategory\_name)  
 **except** Exception **as** e: *# there was an error during the reviews parsing* print(str(e))  
  
 @staticmethod  
 **def** reviews(article, category\_name, subcategory\_name):  
 reviews\_url = **"https://www.ulmart.ru/goods/"** + article + **"/reviews"** *# generate the url for the concrete product* **with** urllib.request.urlopen(reviews\_url) **as** url:  
 html = url.read()  
 soup = BeautifulSoup(html, **"lxml"**)  
 reviews\_num\_already\_have = dataBase.reviews\_num(  
 article) *# check if we have already loaded the reviews in our data base* **if** reviews\_num\_already\_have **is not None**:  
 reviews\_num\_already\_have = reviews\_num\_already\_have[0] *# we have* **else**:  
 reviews\_num\_already\_have = 0 *# we have not* **try**:  
 all\_reviews\_num = soup.find(**"div"**, {  
 **"class"**: **"b-stars-wrap b-stars-wrap\_theme\_normal \_big"**}).find(**"span"**).find(  
 **"span"**).text *# get all reviews number* print(**"\t\t article = "** + article + **"\t all reviews num = "** + all\_reviews\_num)  
 **except** Exception **as** e: *# there was an error during the reviews parsing* **if** soup.find(**"aside"**, {**"class"**: **"b-reviews\_\_side"**}).find(**"p"**).text == **"Нет оценок"**:  
 print(**"\t\t article = "** + article + **"\t 0 отзывов"**)  
 **return True** *# case when there are no reviews at all for the product, so they can not be found* **else**:  
 print(str(e))  
 **return False** *# an error occured* **if** int(all\_reviews\_num) <= 10: *# calculate the number of pages we have for reviews* reviews\_pages\_num = 1  
 **else**:  
 reviews\_pages\_num = round(int(all\_reviews\_num) / 10)  
 **if** reviews\_num\_already\_have == int(all\_reviews\_num): *# we have already downloaded all the reviews for this product* **return True  
 if** reviews\_num\_already\_have != 0:  
 dataBase.remove\_review(article) *# remove the unfinished product reviews to start filing it again* nums = [x **for** x **in** range(1, reviews\_pages\_num + 1)]  
 **for** num **in** nums: *# look through all pages with products kept on page with start\_link* full\_url = reviews\_url + **"/"** + str(num) *# generate the full url for each reviews page* **with** urllib.request.urlopen(full\_url) **as** url: *# load products for a concrete page* html = url.read()  
 soup = BeautifulSoup(html, **"lxml"**)  
 reviews\_on\_page = soup.findAll(**"ul"**, {**"class"**: **"b-list b-list\_theme\_normal b-list\_title-left b-list\_review"**})  
 **for** review **in** reviews\_on\_page: *# get all reviews from concrete product on concrete page* adv = **'Null'** dis = **'Null'** com = **'Null'  
 for** review\_part **in** review.findAll(**"li"**, {**"class"**: **"b-list\_\_item"**}): *# parse the review parts* name = review\_part.find(**'span'**).text  
 **if** name == **"Достоинства"**:  
 adv = review\_part.find(**'div'**).text  
 **elif** name == **"Недостатки"**:  
 dis = review\_part.find(**'div'**).text  
 **elif** name == **"Общие впечатления"**:  
 com = review\_part.find(**'div'**).text  
 dataBase.add\_review(category\_name, subcategory\_name, article, adv, dis,  
 com) *# add the review to the data base* **return True** @staticmethod  
 **def** parse\_sub\_category(start\_link, category\_name):  
 **with** urllib.request.urlopen(start\_link) **as** url:  
 html = url.read()  
 soup = BeautifulSoup(html, **"lxml"**)  
 **for** category **in** soup.findAll(**"div"**, {**"class"**: **"col-main-4"**}): *# all subcategories for category from start\_link* ul = category.find(**"ul"**)  
 **for** li **in** ul.findAll(**"li"**, {**"class"**: **"b-list\_\_item b-list\_\_item\_bigger "**}):  
 name = li.text  
 name = name.replace(**"\n"**, **""**)  
 link = url\_market + li.find(**'a'**, href=**True**)[**'href'**]  
 print(**"\t"** + link + **" "** + name)  
 parser.parse\_product(link, category\_name, name) *# get all products for each subcategory* @staticmethod  
 **def** parse\_categories():  
 **with** codecs.open(**"category\_links.txt"**, **"r"**, **"utf-8"**) **as** links: *# all category links (they were downloaded before)* **for** link **in** links:  
 parsed\_link = link.partition(**' '**)[0]  
 name = link.partition(**' '**)[2]  
 name = name.replace(**"\n"**, **""**)  
 print(parsed\_link + **" "** + name)  
 parser.parse\_sub\_category(parsed\_link, name) *# get subcategories for each category*dataBase = DataBase\_Ulmart()  
url\_market = **"https://www.ulmart.ru"**start\_url = **"https://www.ulmart.ru/catalog/computers\_notebooks"**parser = Parser()  
parser.parse\_categories() *# get categories*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ | | | | | | | | | | |
| Изм. |  | Номера листов (страниц) | | | | Всего  листов  (страниц) в  документе | №  документа | Входящий №  сопроводительн  ого документа и  дата | Подпись | Дата |
| измененн  ых | | замененны  х | новых | аннулиро  ванных |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |

# ПРИЛОЖЕНИЕ 1

# СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. ГОСТ 19.101-77 Виды программ и программных документов. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
2. ГОСТ 19.102-77 Стадии разработки. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
3. ГОСТ 19.103-77 Обозначения программ и программных документов. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
4. ГОСТ 19.104-78 Основные надписи. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
5. ГОСТ 19.105-78 Общие требования к программным документам. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
6. ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
7. ГОСТ 19.401-78 Текст программы. Требования к содержанию и оформлению. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
8. ГОСТ 19.603-78 Общие правила внесения изменений. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
9. ГОСТ 19.604-78 Правила внесения изменений в программные документы, выполненные печатным способом. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.