부 록

A. Word2Vec Embedding 및 Tensorflow Projector 구현 코드

1. main.py(포함 문서 검색 및 관련 단어 추출)

```
from gensim.models import Word2Vec, Doc2Vec
import pickle

def test_word2vec(model, inputword, search_keyword=None):
    if search_keyword is not None:
        if len(inputword) == 1:
            print("=============================================================")
            print("단어 '%s'와(과) 가장 연관성이 높은 %s 그룹 단어는:" % (inputword[0],
search_keyword))
            results = model.most_similar(positive=inputword, topn=len(model.wv.vocab))

            for word, score in results:
                if search_keyword in word:
                    print("word: %s, score: %.3f" % (word, score))

            print("=============================================================\n")
        else:
            vectors = [model.wv[w] for w in inputword]
            result = model.similar_by_vector(sum(vectors), topn=100)

            print("=============================================================")
            print("%s들의 덧셈 연산 결과와 가장 연관성이 높은 %s 그룹 단어는:" %
(str(inputword), search_keyword))
            results = list(filter(lambda r: r[0] not in inputword, result))

            for word, score in results:
                if search_keyword in word:
                    print("word: %s, score: %.3f" % (word, score))

            print("=============================================================\n")
    else:
        if len(inputword) == 1:
            print("=============================================================")
            print("단어 '%s'와(과) 가장 연관성이 높은 단어는:" % (inputword[0]))
            results = model.most_similar(positive=inputword, topn=100)

            for word, score in results:
                print("word:%s, score:%.3f" % (word, score))

            print("=============================================================\n")
```

```
        else:
            vectors = [model.wv[w] for w in inputword]
            result = model.similar_by_vector(sum(vectors), topn=10)

            print("=====================================================")
            print("%s들의 덧셈 연산 결과와 가장 연관성이 높은 단어는:" % str(inputword))
            print(list(filter(lambda r: r[0] not in inputword, result)))
            print("=====================================================\n")


def test_doc2vec(model, input):
    if input.isdigit():
        print("=====================================================")
        print("%s번 문서와 가장 연관성이 높은 문서는:" % input)
        result = model.docvecs.most_similar(str(int(input)-1))

        for idx in range(len(result)):
            doc_idx, sim = result[idx]
            print("문서번호 %d 유사도:%.3f" % (int(doc_idx)+1, sim))
        print("=====================================================\n")

    else:
        with open('datalist.pkl', 'rb') as listfile:
            datalist = pickle.load(listfile)

        print("=====================================================")
        print("단어 '%s'을(를) 포함한 문서 번호는:" % input)
        for idx, doc in enumerate(datalist):
            if input in doc:
                print("Document ID:", idx+1)
        print("=====================================================\n")


if __name__ == "__main__":
    # 인자들은 각자 맞게 수정하면 됨
    word2vec_model = Word2Vec.load("weights/word2vec/model.model")
    doc2vec_model = Doc2Vec.load("weights/doc2vec/model.model")

    # 시나리오 1. word2vec 단일 단어 검색
    test_word2vec(word2vec_model, inputword=['사다리/N/cause'], search_keyword="prevent")
    test_word2vec(word2vec_model, inputword=["추락/N"])

    # 시나리오 2. word2vec 여러 단어 검색 --> 각 단어 벡터의 합 연산
    # test_word2vec(word2vec_model, ["맨홀/N", "청소/N"], search_keyword="summary")
    test_word2vec(word2vec_model, ["사다리/N/summary", "추락/N/summary"])


    # 시나리오 3. doc2vec 문서 id 검색 --> 가장 연관성이 높은 문서의 id 반환
    test_doc2vec(doc2vec_model, '228')

    시나리오 4. 특정 단어 검색 시 해당 단어가 token으로 포함된 문서 번호 출력
    test_doc2vec(doc2vec_model, '작업대/N/summary')
```

2. trainer.py

```
#-*- coding: utf-8 -*-

import re
import pandas as pd
import os
import numpy as np
import pickle
from konlpy.tag import Hannanum
tagger = Hannanum()
# tagger = Okt()

from gensim.models import Word2Vec
from gensim.models.doc2vec import Doc2Vec, TaggedDocument

from visualizer import visualize

def prepare_data(dataroot):
    # ============================================#
    #                   1-1                      #
    # ============================================#
    try:
        df = pd.read_csv(dataroot, encoding='utf-8')
    except:
        try:
            df = pd.read_csv(dataroot, encoding='euc-kr')
        except:
            try:
                df = pd.read_csv(dataroot, encoding='utf-16')
            except:
                try:
                    df = pd.read_csv(dataroot, encoding='cp949')
                except:
                    print("Encoding error!")

    # ============================================#
    #                   1-2                      #
    # ============================================#
    # col 0: document id
    # col 1: summary
    # col 2: cause factor
    # col 3: prevent
    col2group = {0: 'id', 1: 'summary'}
    ds = df.values # 2251, 4

    print(ds.shape)

    # ============================================#
    #                   1-3                      #
    # ============================================#
    # stop_word_list = ['분경', '사망/N', '위하다/P', '하다/P', '피재자/N', '재해임/N', '재
해당/N','재해/N','개소/N', '소재/N', '건설주/N', '건설/N', '현장/N']

    data = []
    doc_idx = 0
    for doc in ds:
```

```
        doc_temp = []
        # 0, 1, 2, 3
        for col_idx in range(len(doc)):
            if col_idx == 0:
                pass
            else:
                doc[col_idx] = str(doc[col_idx]).strip() # remove
                # print(doc[col_idx])

                if len(doc[col_idx]) < 6:
                    print("length warning!", doc[col_idx])
                    pass
                else:
                    temp = [word for word in tagger.pos(doc[col_idx], join=True)]
                    npList = [word for word in temp if word.split("/")[1] == "P" or
word.split("/")[1] == "N"]

                    for idx in range(len(npList)):
                        word = npList[idx]
                        if word.split('/')[1] == "P":
                            npList[idx] = word[:-2] + "다" + word[-2:]

                    npList = [word.replace('몸/N', '몸중심/N') for word in npList]
                    npList = [re.sub('^[가-힣]/N', '', word) for word in npList]
                    npList = [word.replace('백호/N', '백호우/N').replace('청소작업/N', '청소
/N').replace('실족하/N', '실족/N') for word in npList]
                    npList = [word.strip() for word in npList if word.strip()]
                    # npList = [word for word in npList if word not in stop_word_list]
                    # npList = [word + "/" + col2group[col_idx] for word in npList]


                    doc_temp.extend(npList)

        data.append(doc_temp)
        doc_idx += 1
        if doc_idx % 100 == 0:
            print('prepare %d documents completed!' % doc_idx)

    return data


  def train_word2vec(datalist, out_path):
      # hyper parameters
      embedding_dim = 300
      window_size = 4
      num_workers = 4
      num_epochs = 2000
      skip_gram = 1
      negative_sampling = 5

      print("Word2Vec Learning started!")
      embedding_model = Word2Vec(datalist, size=embedding_dim, window=window_size,
                                 workers=num_workers, iter=num_epochs,
                                 sg=skip_gram, negative=negative_sampling,
                                 sample=1e-5)
      print("Word2Vec Learning finished!")

      embedding_model.save(out_path)
      print("Model saved!")
```

```
  def train_doc2vec(datalist, out_path):
      tagged_data = [TaggedDocument(words=doc, tags=[str(i)])
                     for i, doc in enumerate(datalist)]

      # hyper parameters
      embedding_dim = 300
      num_epochs = 1000
      negative_sampling = 5
      alpha = 0.025
      min_alpha = 0.00025
      min_count = 2

      model = Doc2Vec(size=embedding_dim,
                      alpha=alpha,
                      min_alpha=min_alpha,
                      min_count=min_count,  # compute words which are included more than 2
times in corpus
                      dm=1,
                      negative=negative_sampling)
      model.build_vocab(tagged_data)

      print("Doc2Vec Learning started!")
      for epoch in range(num_epochs):
          model.train(tagged_data,
                      total_examples=model.corpus_count,
                      epochs=model.iter)

          model.alpha -= 0.0002
          model.min_alpha = model.alpha

          print("[%d/%d] complete!" % (epoch + 1, num_epochs))
      print("Doc2Vec Learning finished!")

      model.save(out_path)
      print("Model saved!")

  def main(mode):
      tagged_list = prepare_data('dataset/embedding.csv')
      print("Prepare data list completed!")

      with open("datalist.pkl", "wb") as f:
          pickle.dump(tagged_list, f)

      # for doc in tagged_list:
      #     print(doc)

      with open("datalist.pkl", "rb") as f:
          tagged_list = pickle.load(f)

      if mode == "word2vec":
          if not os.path.exists("weights/word2vec"):
              os.makedirs("weights/word2vec")

          train_word2vec(tagged_list, "weights/word2vec/model.model")

      elif mode == "doc2vec":
          if not os.path.exists("weights/doc2vec"):
              os.makedirs("weights/doc2vec")
```

```
        train_doc2vec(tagged_list, "weights/doc2vec/model.model")

if __name__ == "__main__":
    main('word2vec')
    visualize("weights/word2vec/model.model", "tb_word2vec")
    # main('doc2vec')
    # visualize("weights/doc2vec/fallsummary1216/model.model",
"tb_doc2vec/fallsummary1216")
```

3. visualizer.py

```
from gensim.models import Word2Vec
import numpy as np
import os
import tensorflow as tf
from tensorflow.contrib.tensorboard.plugins import projector

def visualize(model_path, logdir):
    os.makedirs(logdir, exist_ok=True)

    model = Word2Vec.load(model_path)

    meta_file = "w2x_metadata.tsv"
    placeholder = np.zeros((len(model.wv.index2word), 300))

    with open(os.path.join(logdir, meta_file), 'wb') as file_metadata:
        # file_metadata.write("Word".encode('utf-8') + b'\n')
        for i, word in enumerate(model.wv.index2word):
            placeholder[i] = model[word]

            # temporary solution for https://github.com/tensorflow/tensorflow/issues/9094
            if word == '':
                print("Emply Line, should replecaed by any thing else, or will cause a
bug of tensorboard")
                file_metadata.write("{0}".format('<Empty Line>').encode('utf-8') + b'\n')

            else:
                # group = metadata[]
                line = word
                file_metadata.write(line.encode('utf-8') + b'\n')

        print("Write summary model metadata complete!!!")
        # define the model without training
        sess = tf.InteractiveSession()

        embedding = tf.Variable(placeholder, trainable=False, name='w2x_metadata')
        tf.global_variables_initializer().run()

        saver = tf.train.Saver()
        writer = tf.summary.FileWriter(logdir, sess.graph)

        # adding into projector
        config = projector.ProjectorConfig()
        embed = config.embeddings.add()
        embed.tensor_name = 'w2x_metadata'
        embed.metadata_path = meta_file

        # Specify the width and height of a single thumbnail.
        projector.visualize_embeddings(writer, config)
        saver.save(sess, os.path.join(logdir, 'w2x_metadata.ckpt'))
        print('Run `tensorboard --logdir={0}` to run visualize result on
tensorboard'.format(logdir))
```

B. CNN기반 문서 분류 모델 코드

1. config.py

```
  import argparse
  parser = argparse.ArgumentParser(description='CNN text classifier')

  # Data setting configurations
  parser.add_argument('--train_root', type=str, default=r'dataset/0505/total.csv',
help='path to training CSV file')
  parser.add_argument('--test_root', type=str, default=r'dataset/0505/fallen.csv',
help='path to testing CSV file')
  parser.add_argument('--shuffle', action='store_true', default=False, help='shuffle the
data every epoch')
  parser.add_argument('--batch_size', type=int, default=5, help='batch size for training
[default: 64]')
  parser.add_argument('--split_ratio', type=float, default=0.2, help='train set/ eval set
split ratio [default:0.1]')

  # Training configurations
  parser.add_argument('--lr', type=float, default=0.0001, help='initial learning rate
[default: 0.001]')
  parser.add_argument('--num_epochs', type=int, default=256, help='number of epochs for
train [default: 256]')
  parser.add_argument('--save_dir', type=str, default=None, help='where to save the
snapshot')
  parser.add_argument('--training_model', type=str, default=None, help='to continue
training')

  # Step sizes configurations
  parser.add_argument('--log_interval',  type=int, default=1,   help='how many steps to
wait before logging training status [default: 1]')
  parser.add_argument('--eval_interval', type=int, default=5, help='how many steps to wait
before evaluating [default: 100]')

  # Model configurations
  parser.add_argument('--dropout', type=float, default=0.5, help='the probability for
dropout [default: 0.5]')
  parser.add_argument('--max-norm', type=float, default=3.0, help='l2 constraint of
parameters [default: 3.0]')
  parser.add_argument('--embedding_dim', type=int, default=512, help='number of embedding
dimension [default: 128]')
  parser.add_argument('--channel_out', type=int, default=128, help='number of each kind of
kernel')
  parser.add_argument('--kernel_sizes', type=str, default='3,4,5,6,7', help='comma-
separated kernel size to use for convolution')

  # Misc.
  parser.add_argument('--snapshot', type=str, default=None, help='filename of model
snapshot [default: None]')
  parser.add_argument('--predict', type=str, default=None, help='predict the sentence
given')
  parser.add_argument('--mode', type=str, default='train', choices=['train', 'test'],
help='train or test')
  args = parser.parse_args()
  def get_config():
```

```
return args
```

2. main.py

```
#-*- coding: utf-8 -*-

from __future__ import print_function
import random
import torch
import torch.backends.cudnn as cudnn

import os, sys
sys.path.append(os.path.dirname(os.path.abspath(os.path.dirname(__file__))))

from config import get_config
from train import Trainer
from dataloader import get_loader

# Device configuration
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

def main(config):
    if config.save_dir is None:
        config.save_dir = 'samples'
    os.system('mkdir {0}'.format(config.save_dir))

    config.manual_seed = random.randint(1, 10000)
    print("Random Seed: ", config.manual_seed)

    random.seed(config.manual_seed)
    torch.manual_seed(config.manual_seed)
    torch.cuda.manual_seed_all(config.manual_seed)

    cudnn.benchmark = True

    print("[*] Preparing dataloader...")
    train_set, train_loader, eval_set, eval_loader, test_set, test_loader  =
get_loader(train_root=config.train_root,

test_root=config.test_root,

split_ratio=config.split_ratio,

batch_size=config.batch_size)
    print("Length of training set:", len(train_set))
    print("Length of evaluating set:", len(eval_set))
    print("Length of testing set:", len(test_set))
    print("[*] Preparing dataloader completed!")

    trainer = Trainer(config, train_set, train_loader, eval_set, eval_loader, test_set,
test_loader)

    if config.mode == 'train':
        trainer.train()
    else:
        trainer.test()

if __name__ == "__main__":
    config = get_config()
    main(config)
```

3. dataloader.py

```
from torch.utils.data import DataLoader, Dataset

from konlpy.tag import Hannanum
tagger = Hannanum()

import numpy as np
import pandas as pd
from sklearn.utils import shuffle

class GlobalDataset(Dataset):
    def __init__(self, train_root, test_root, MAX_LENGTH, split_ratio):
        entire_set = pd.read_csv(train_root, header=None, encoding='UTF8')
        entire_set = entire_set.drop(columns=0).dropna()
        entire_set = shuffle(entire_set)

        # train-eval split
        split_row = int(len(entire_set) * (1.-split_ratio))
        self.train_set = entire_set.values[:split_row, :]
        self.eval_set = entire_set.values[split_row:, :]

        test_set = pd.read_csv(test_root, header=None,
encoding='UTF8').drop(columns=[0]).dropna()
        self.test_set = test_set.values

        self.group2idx = {'ELEC': 0, 'FALL': 1, 'COLA': 2, 'CRAS': 3, 'SPLA' : 4}
        self.MAX_LENGTH = MAX_LENGTH

        self.train_x = []
        self.train_y = []
        self.eval_x = []
        self.eval_y = []
        self.test_x = []

        self.build_word2idx()

    def build_word2idx(self):
        # global word-to-index vocabulary for training, evaluating, testing
        self.word2idx = {}

        # for training set, build word vocabulary
        for item in self.train_set:
            sentence = item[0]
            group = item[1]

            # remove useless space
            sentence = sentence.lstrip().rstrip()
            if len(sentence) > 5:
                tokenized = tagger.nouns(sentence)
                for token in tokenized:
                    if token not in self.word2idx:
                        self.word2idx[token] = len(self.word2idx)

                tokenized = [self.word2idx[w] for w in tokenized]
                padding = [0 for i in range(self.MAX_LENGTH - len(tokenized))]
                tokenized.extend(padding)
                self.train_x.append(tokenized)
                self.train_y.append(self.group2idx[group])
```

```
        # for evaluating set, add word vocabulary
        for item in self.eval_set:
            sentence = item[0]
            group = item[1]

            # remove useless space
            sentence = sentence.lstrip().rstrip()
            if len(sentence) > 5:
                tokenized = tagger.nouns(sentence)
                for token in tokenized:
                    if token not in self.word2idx:
                        self.word2idx[token] = len(self.word2idx)

                tokenized = [self.word2idx[w] for w in tokenized]
                padding = [0 for i in range(self.MAX_LENGTH - len(tokenized))]
                tokenized.extend(padding)
                self.eval_x.append(tokenized)
                self.eval_y.append(self.group2idx[group])

        # for testing set, add word vocabulary
        for item in self.test_set:
            sentence = item[0]

            # remove useless space
            sentence = sentence.lstrip().rstrip()
            if len(sentence) > 5:
                tokenized = tagger.nouns(sentence)
                for token in tokenized:
                    if token not in self.word2idx:
                        self.word2idx[token] = len(self.word2idx)

                tokenized = [self.word2idx[w] for w in tokenized]
                padding = [0 for i in range(self.MAX_LENGTH - len(tokenized))]
                tokenized.extend(padding)
                self.test_x.append(tokenized)

        self.train_x = np.asarray(self.train_x)
        self.train_y = np.asarray(self.train_y)
        self.eval_x = np.asarray(self.eval_x)
        self.eval_y = np.asarray(self.eval_y)
        self.test_x = np.asarray(self.test_x)

    def __len__(self):
        return len(self.train_x)

    def __getitem__(self, idx):
        return self.train_x[idx], self.train_y[idx]

class EvalDataset(Dataset):
    def __init__(self, global_dataset):
        self.global_dataset = global_dataset
        self.eval_x = self.global_dataset.eval_x
        self.eval_y = self.global_dataset.eval_y

    def __len__(self):
        return len(self.eval_x)

    def __getitem__(self, idx):
```

```
        return self.eval_x[idx], self.eval_y[idx]

class TestDataset(Dataset):
    def __init__(self, global_dataset):
        self.global_dataset = global_dataset
        self.test_x = self.global_dataset.test_x

    def __len__(self):
        return len(self.test_x)

    def __getitem__(self, idx):
        return self.test_x[idx]

def get_loader(train_root, test_root, split_ratio, batch_size):
    train_set = GlobalDataset(train_root, test_root, 500, split_ratio)
    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    eval_set = EvalDataset(train_set)
    eval_loader = DataLoader(eval_set, batch_size=batch_size, shuffle=False)
    test_set = TestDataset(train_set)
    test_loader = DataLoader(test_set, batch_size=1, shuffle=False)

    return train_set, train_loader, eval_set, eval_loader, test_set, test_loader
```

4. model.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F

from config import get_config

# Device configuration
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class CNNTextClassifier(nn.Module):
    def __init__(self, vocab_size, num_classes):
        super(CNNTextClassifier, self).__init__()
        config = get_config()
        config.kernel_sizes = [int(k) for k in config.kernel_sizes.split(',')]
        self.config = config

        embedding_dim = vocab_size
        channel_in = 1
        channel_out = config.channel_out
        kernel_sizes = config.kernel_sizes

        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.conv1 = nn.ModuleList([nn.Conv2d(channel_in, channel_out,
                                              (k, embedding_dim))
                                    for k in kernel_sizes])
        self.dropout = nn.Dropout(config.dropout)
        self.fc1 = nn.Linear(len(kernel_sizes) * channel_out, num_classes)

    def forward(self, x):
        h = self.embedding(x)
        h = h.unsqueeze(1) # (batch, 1, vocab_size, embedding_dim)
        h = [F.relu(conv(h)).squeeze(3) for conv in self.conv1] # [(batch, channel_out,
vocab_size)] * len(kernel_sizes)
        h = [F.max_pool1d(i, i.size(2)).squeeze(2) for i in h]  # [(batch, channel_out)]
* len(kernel_sizes)
        h = torch.cat(h, 1) # (batch, channel_out * len(kernel_sizes)
        h = self.dropout(h)
        logit = self.fc1(h) # (batch, num_classes)
        return logit
```

5. train.py

```
import torch
import torch.nn as nn
import torch.optim as optim
from model import CNNTextClassifier
import time, os
import numpy as np

# Device configuration
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class Trainer(object):
    def __init__(self, config, train_set, train_loader, eval_set, eval_loader, test_set,
test_loader):
        self.config = config
        self.train_set= train_set
        self.train_loader = train_loader
        self.test_set = test_set
        self.test_loader = test_loader
        self.eval_set = eval_set
        self.eval_loader = eval_loader

        self.mode = config.mode

        self.idx2group = { 0 : 'ELEC', 1 : 'FALL', 2 : 'COLA', 3 : 'CRAS', 4 : 'SPLA'}

        self.lr = config.lr
        self.num_epochs = config.num_epochs

        self.log_interval = config.log_interval
        self.eval_interval = config.eval_interval
        self.save_dir = config.save_dir

        if len(self.config.kernel_sizes) == 1:
            self.channels = "single"
        else:
            self.channels = "multi"

        self.build_net()

    def build_net(self):
        model = CNNTextClassifier(len(self.train_set.word2idx)+1, 5) # class의 개수

        if self.config.training_model is not None:
            model.load_state_dict(torch.load(self.config.training_model))

        if self.mode == 'test':
            model.load_state_dict(torch.load(os.path.join(self.save_dir, 'best_acc.pth'),
lambda storage, loc: storage))

        self.model = model.to(device)
        print("[*] Prepare model completed!")

    def train(self):
        criterion = nn.CrossEntropyLoss()
```

```
        parameters = filter(lambda p: p.requires_grad, self.model.parameters())
        optimizer = optim.Adam(parameters, lr=self.lr)

        steps = 0
        best_acc = 0.

        print("\nLearning started!")
        start_time = time.time()
        for epoch in range(self.num_epochs):
            for step, (feature, target) in enumerate(self.train_loader):
                self.model.train()
                step_batch = feature.size(0)
                feature = feature.to(device).long()
                target = target.to(device).long()

                logits = self.model(feature)
                loss = criterion(logits, target)

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                steps += 1

                if steps % self.log_interval == 0:
                    predicted = torch.max(logits, 1)[1].view(target.size())
                    corrects = (predicted.data == target.data).sum()
                    accuracy = 100.0 * (float(corrects) / step_batch)

                    end_time = time.time()
                    print("[%d/%d] [%d/%d] time:%.3f loss:%.3f accuracy:%.3f"
                          % (epoch + 1, self.num_epochs, step + 1, len(self.train_loader),
                             end_time - start_time, loss.item(), accuracy))

                if steps % self.eval_interval == 0:
                    acc = self.eval()
                    if acc > best_acc:
                        best_acc = acc
                        torch.save(self.model.state_dict(), os.path.join(self.save_dir,
'best_acc.pth')) # save model
                        print("Save model completed!")

        print("Learning finished!")
        torch.save(self.model.state_dict(), os.path.join(self.save_dir, 'final.pth')) #
save model
        print("Save model completed!")

    def eval(self):
        self.model.eval()

        avg_acc = 0.
        avg_loss = 0.

        criterion = nn.CrossEntropyLoss()

        for feature, target in self.eval_loader:
            step_batch = feature.size(0)

            feature = feature.to(device).long()
```

```
            target = target.to(device).long()

            logits = self.model(feature)
            loss = criterion(logits, target)

            predicted = torch.max(logits, 1)[1].view(target.size())
            corrects = (predicted.data == target.data).sum()
            accuracy = 100.0 * (float(corrects) / step_batch)

            avg_loss += loss.item() / len(self.train_loader)
            avg_acc += accuracy / len(self.train_loader)

        print("Evaluation- loss: %.3f accuracy: %.3f"% (avg_loss, avg_acc))
        return avg_acc

    def test(self):
        self.model.eval()
        for idx, feature in enumerate(self.test_loader):
            feature = feature.to(device).long()

            logits = self.model(feature)
            predicted = torch.max(logits, 1)[1]
            predicted = self.idx2group[predicted.item()]

            print("Document ID: {}\tPredicted class: {}".format(idx+1, predicted))
```