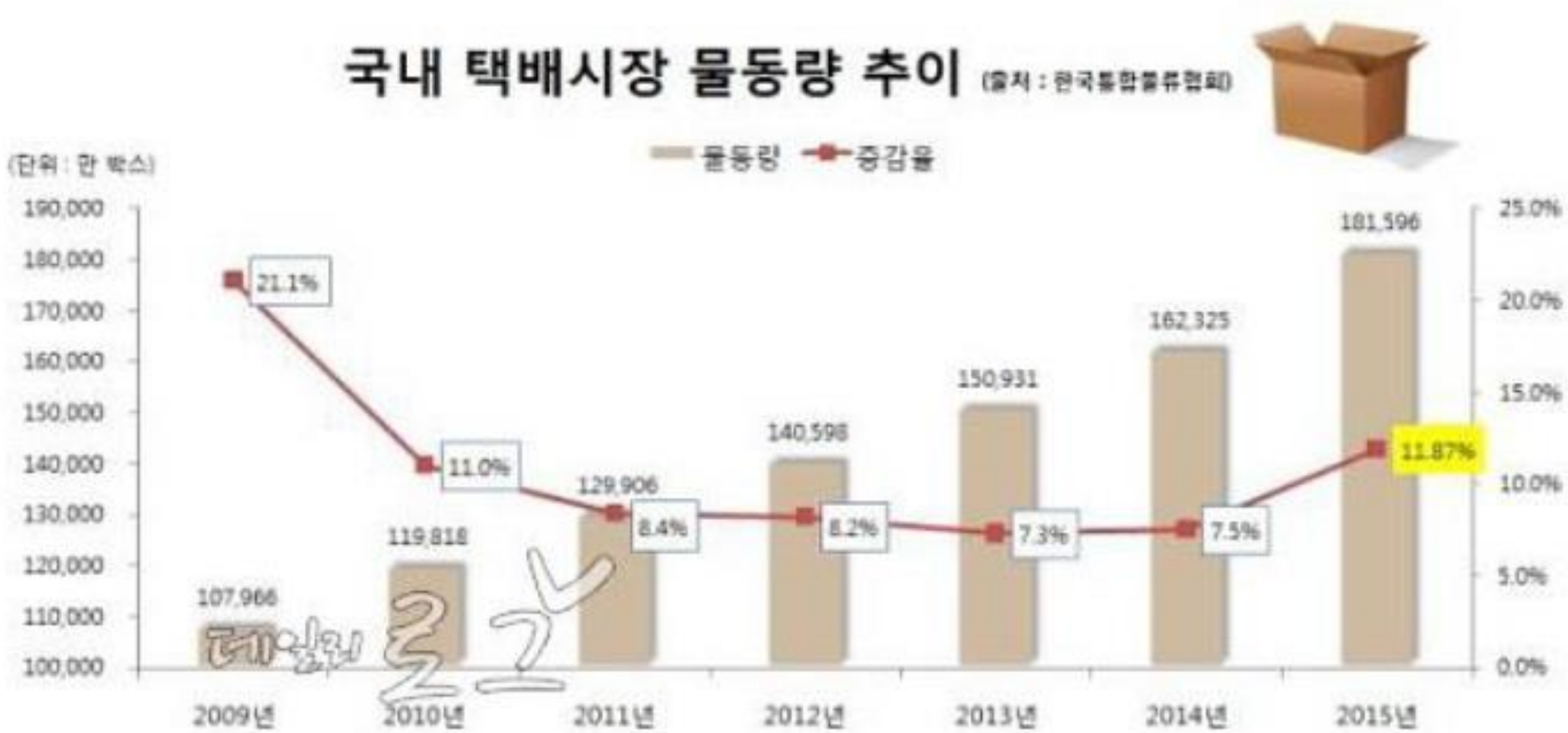
Ⅰ 비즈니스 문제 정의

1. 택배와 민원

"민원"이 빅데이터와 어떤 연관성이 있는지를 알아보고, 관련된 비즈니스 문제를 정의한다.

민원의 사전적 의미는 행정기관에 대해 원하는 바를 요구하는 일이지만, 소비자의 권익이 나날이 높아지면서 그 단어의 의미는 점차 확대됐다. 이제는 공공기간에서 접수 받는 것 외에도 민간 기업의 콜센터에서 접수 받는 것도 민원이라고 볼 수 있다.

민원의 그 종류에 대한 수는 헤아릴 수 없이 복잡하고 다양하다. 그러나 그 중에서 우리는 '택배' 민원에 대한 비즈니스 문제 정의하고 해결할 것이다. 이 분야를 선택한 이유는 간단하다. 이것은 다른 민원 종류보다 최근에 가장 폭발적으로 증가하고 있다. 그 이유는 바로 인터넷과 모바일 기술의 발전으로 언제 어디서나 물건을 구매하거나 되파는 것이 너무나 간편해졌기 때문이다. 게다가 우후죽순 늘어나는 쇼핑몰과 택배 업체 덕분에 소비자는 다양하고 값싼 물건을 만 하루 안에 갖는 것이 너무나도 쉬워졌다. 따라서 다른 민원보다도 상품 구매와 같은 접근이 쉬워지면서 동시에 택배 민원에 대한 접수도 폭발적으로 증가했다.



[그림 1] 국내 택배시장 물동량 추이

2. 문제 정의

그렇다면 해마다 증가하는 택배 민원을 빠르고 정확하게 해결할 수는 없을까?

예전에는 택배 업체가 몇 개 없었던 것과 달리 다수의 업체들이 생겨나 경쟁하게 되어, 해당 택배 업체들은 큰 고민에 빠졌다. 그들의 고민은 간단하다. 고객의 수(소비자와 쇼핑몰 업체 포함)는 한정되어 있는데 어떻게 고객의 만족도를 높여 우리 택배를 이용할 수 있을까에 대한 깊은 고민이다.

그래서 도입된 서비스를 예를 들면 편의점 택배나 운송장번호로 위치를 파악할 수 있는 애플리케이션이 있다. 그러나 이 부분은 고객의 만족도 향상에 크게 기여한 부분은 아니다. 왜냐하면 고객의 소리를 직접 듣고 고칠 수 있는 방법은 아니기 때문이다.

이때 우리는 택배와 관련된 새로운 비즈니스가 도입돼야 한다고 생각했다. 바로 택배 민원에 대한 자동분류기가 필요하다는 것이다. 물론 모든 택배 회사의 웹사이트마다 고객 민원 접수를 받는 게시판이 존재해서 필요하지 않다고 느낄지도 모른다. 그러나 제한된 택배 회사 소속의 콜센터 직원으로는 고객의 모든 민원을 처리하기는 쉽지 않을 것이다. 따라서 기존의 방법으로는 고객 만족도를 높이기가 여간 쉽지 않다.

따라서 고객 만족도를 높이기 위해 **택배 민원 자동분류기**를 만들려고 한다. 그 목적은 간단하지만 효과는 강력하다. 말 그대로 민원에 대한 자동분류기를 도입하는 것이지만 고객이 민원을 접수하는 즉시 그 결과를 뽑아낼 수 있기 때문이다. 그러므로 고객은 택배관련 문제가 발생 시 그 해결책을 빠르게 알 수 있으며, 택배업체는 민원 제기한 고객의 만족도를 높여 타 업체보다 월등한 경쟁력을 지닐 수 있다는 점이다.

또한 민원 접수는 대게 콜센터 직원들이 맡고 있어 해당 민원 분류에 대한 주관적인 견해가 들어갈 수 있다는 단점이 있다. 그러나 자동분류기를 도입하게 되면 민원 분류에 대한 객관적이며 효율적인 방법을 제시할 것이다. 또한 이점은 콜센터 직원에게도 도움을 줄 것이 분명하다.

이러한 민원 자동분류기, 어떻게 만들 수 있을까?

Ⅱ 비즈니스 문제 해결 방안

1. 문제 해결 방안

지능 정보를 활용한 고객의 택배 민원처리를 어떻게 할 수 있을까?

지능정보를 활용한 택배 민원 자동분류기

본 문서는 택배 민원 자동분류기를 도입하여 고객과 택배 업체의 만족도를 높일 방법을 제시하겠다. 우선 택배와 관련된 민원의 종류를 생각해보자. 택배 관련 민원은 대표적으로 **배송지연, 택배분실, 서비스 불만족, 택배 파손으로 크게 다섯 가지 유형을 들** 수 있으며 다른 민원보다 충분히 예측 가능하다는 점이 있다.

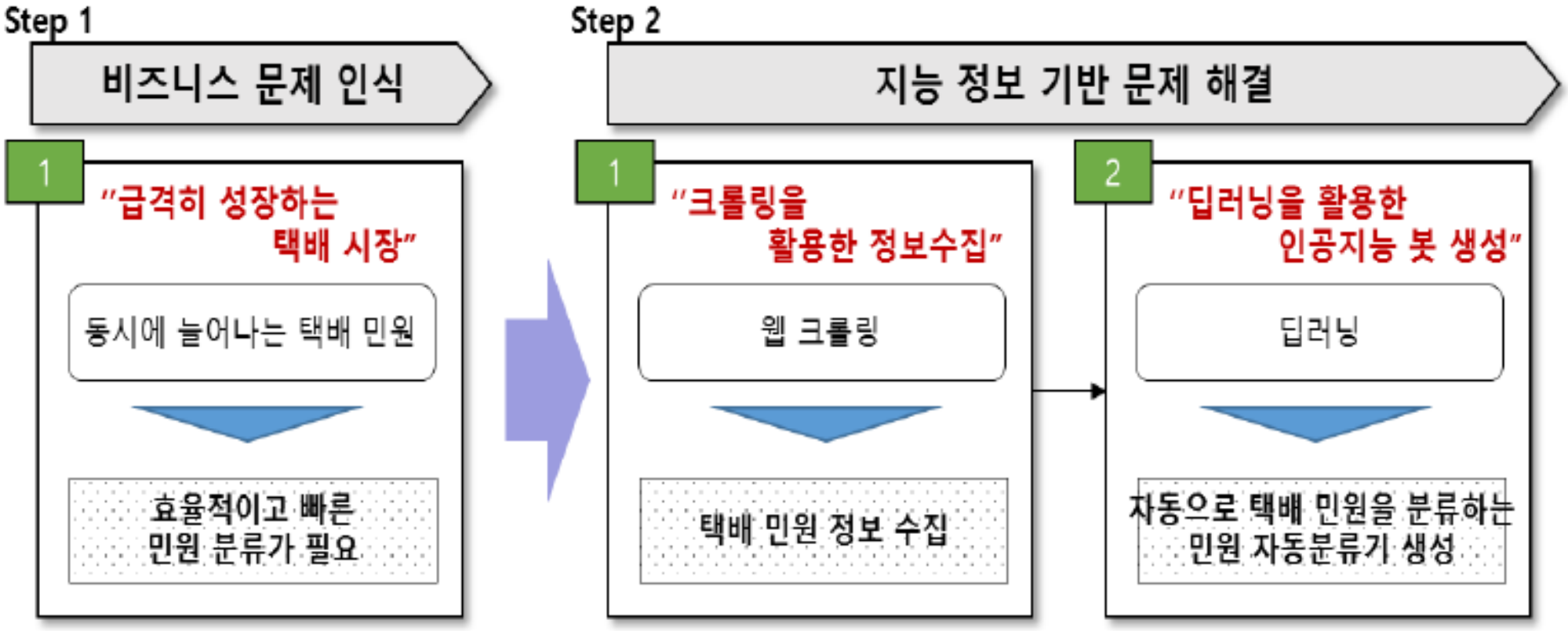
따라서 민원 분류를 지능정보 기술로 빠르고 효율적으로 하게 된다면 고객과 택배업체 모두에게 큰 만족감을 가져다 줄 것이 분명하다.

그러므로 본 실습에서는 '택배 민원'을 바탕으로 민원 자동분류기를 개발하여 고객의 만족도를 높일 수 있는 비즈니스 솔루션을 제시하겠다.

2. 문제 해결 과정

비즈니스 문제인 '택배 민원 자동처리'를 해결하고 자지능정보(딥러닝)를 활용하여 어떻게 문제를 해결할 수 있는지에 대한 과정을 설명한다. 따라서 택배민원이 접수된 내용을 토대로 민원 발생 유형 자동분류기를 개발하여 문제 해결에 앞장서겠다.

[문제 해결 과정]



[분석 과정]

[분석 환경]

필요 환경

- 운영체제: Ubuntu 14.04 LTS 이상 (아래 코드는 Windows에서도 테스트 완료)
- 텐서플로우 버전: r0.11 이상 (r.0.12 에서도 테스트 완료)
- 파이썬 버전: 3.5.x

Ⅲ 비즈니스 문제 해결

1. 필요 데이터 확보 및 탐색

우선 택배에 대한 민원 데이터가 필요하다. 실제 택배 민원 데이터 수집은 웹 크롤링을 통해 이루어지지만 본 교육 컨텐츠의 목적 상 그 과정은 생략한다. 따라서 본 컨텐츠에서는 크롤링을 통해 수집된 데이터(약 2700여개의 민원으로 이루어진 텍스트 파일)가 생성되었다는 가정 하에 다음 과정을 진행하겠다.

1.1 데이터 불러오기

실행 코드

```
# 라이브러리를 불러옵니다.
from pylab import*
import csv
import numpy as np
import tensorflow as tf
import collections
import argparse
import time
import os
from six.moves import cPickle
from string import digits
import re
import matplotlib.pyplot as plt
```

```
# 데이터 처리를 위한 기본 라이브러리인 numpy를 불러온다.
# 기타 텍스트 데이터 전 처리를 위한 라이브러리들 역시 불러온다.
# 데이터의 수를 세기 위한 라이브러리인 collections를 불러온다.
# 전체 경과 시간을 측정하기 위한 라이브러리인 time을 불러온다.
# 데이터 경로를 합치기 위한 라이브러리인 os를 불러온다.
# byte 데이터로 저장하고 불러오기 위한 라이브러리인 cPickle을 불러온다.
```

실행 코드

```
# 데이터를 불러옵니다.
save_dir = "/home/eduuser/NN/Data/complaint"

senta = []
with open(save_dir + '/' + 'complaint.csv', 'r') as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        senta.append(row)

name = senta[0]
print(name)
cls_name = senta[0][3:]
print(len(senta))
print(senta)
```

출력 결과

```
2728 # 데이터 길이

['대번호', '소번호', '문장', '훼손/파손', '분실', '배송지연', '오배송', '기타', '관련없음']

[['558', '2', '  송장을 확인하자 본인에게 발송된 명절 선물(생물 꽃게)로, 냉장 보관을 해야하는 상품을 연락도 없이 택배 보관함도 아닌 소화전함에 방치해두어 부패함.', '', '', '', '', 'y', ''], ['558', '3', ' 피신청인측 고객센터에 접수하였으나 의정부 대리점으로 미루고, 의정부지점의 담당기사와 통화하였으나 송장에 적힌 본인의 휴대폰 다른 번호가 기재가 되어 자신의 잘못이 아님으로 미룸.', '', '', '', '', 'y', ''], ['558', '4', '  택배대리점(03108365136) 대표에게 전화 받았으나 같은 말을 반복하여 전화 끈음.', '', '', '', '', '', 'y'], ['558', '5', '  신청인은 제대로 배달품 인계하지 않은 책임으로 배상요구함.', '', '', '', '', '', 'y'], ['559', '1', '신청인은 2015.9.17. 피신청인[(주)천일정기화물]을 통해 충남 논산에서 울산으로 리틀윙스트리플 유아자전거를 배송 의뢰하고 7,000원을 결제함. \n', '', '', '', '', '', 'y'], ['559', '2', '리틀윙스트리플 유아자전거는 2년 전에 300,000원에 구입했고, 이 자전거를 울산에 거주하는 소비자에게 중고제품으로 120,000원을 받고 판매함.', '', '', '', '', '', 'y'], ….., ['559', '3', '배송의뢰 시 플라스틱으로 되어 있는 햇빛가리개는 안장에 고정시켜 박스로 감싸 포장했는데 햇빛가리개는 파손되고 안장 부분과 차체는 흰 상태로 배송되어 피신청인에게 이의제기하고 배상을 요구하였으나 책임을 회피함.', 'y', '', '', '', '', ''], ['559', '4', '이에 신청인은 자전거 위에 무거운 물건을 쌓아 올려 햇빛가리개가 파손되고 차체가 흰 것이라고 주장하며, 공정위 고시 소비자분쟁해결기준에 의거 중고 제품가액(120,000원)의 배상을 요구함.', '', '', '', '', '', 'y'], ['560', '1', '신청인은 2015. 5. 8. 피신청인 2(몰테일)의 홈페이지에서 이 사건 공기청정기(IQair)를 주문하고 1,083,974원을 결제한 사실이 있는데 피신청인 2가 피신청인 1(대신화물)을 통해 배송 하던 중 배송업자의 과실로 파손되어 손해배상을 요구한 사건임.', 'y', '', '', '', '', '']]
```

훈련용으로 사용될 데이터는 총 2728 개의 문장으로 구성되어 있으며, 해당 민원의 대번호/소번호/내용(문장 단위)/민원발생 유형(총 6 가지)으로 나눠졌음을 확인했다.

실행 코드

```
senta = senta[1:]
senta = np.array(senta)

senta_x = senta[:,2]
senta_y = senta[:,3:]

print(senta_x)
print(senta_y)
```

출력 결과

```
[' 2013. 11. 4. 신청인은 지인에게 선물하기 위해 햅쌀 20kg을 피신청인을 통해 선불로 택배 발송함.'
 ' 2013. 11. 8. 상하차 도중 포장이 훼손되어 배송불가하다는 연락을 받아, 피신청인의 택배기사가 훼손된 것과 동일한 물품을 구입하여 재발송하기로하였으나 11. 20. 경까지 이행하지 않음'
 '신청인은 청원군 햅쌀 20kg 배송 이행과 함께 두 달 이상 지연된 것에 대한 배상을 요구함.' ...,
 '배송의뢰 시 플라스틱으로 되어 있는 햇빛가리개는 안장에 고정시켜 박스로 감싸 포장했는데 햇빛가리개는 파손되고 안장 부분과 차체는 흰 상태로 배송되어 피신청인에게 이의제기하고 배상을 요구하였으나 책임을 회피함.'
 '이에 신청인은 자전거 위에 무거운 물건을 쌓아 올려 햇빛가리개가 파손되고 차체가 흰 것이라고 주장하며, 공정위 고시 소비자분쟁해결기준에 의거 중고 제품가액(120,000원)의 배상을 요구함.'
```

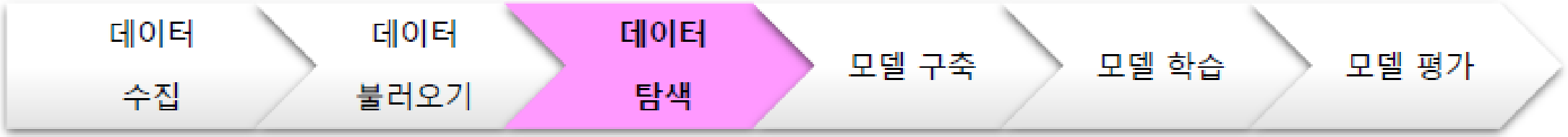
```
 '신청인은 2015. 5. 8. 피신청인 2(몰테일)의 홈페이지에서 이 사건 공기청정기(IQair)를 주문하고 1,083,974원을 결제한 사실이 있는데 피신청인 2가 피신청인 1(대신화물)을 통해 배송 하던 중 배송업자의 과실로 파손되어 손해배상을 요구한 사건임.']

[['' '' '' '' '' 'y']
 ['y' '' '' '' '' '']
 ['' '' '' '' '' 'y']
 ...,
 ['y' '' '' '' '' '']
 ['' '' '' '' '' 'y']
 ['y' '' '' '' '' '']]
```

모델 구축을 위해 학습에 사용 될 Input(민원 내용)과 Target(민원의 종류)로 나누어 각각을 senta_x와 senta_y로 지정한다. Csv 파일을 열어보면 민원 내용에 대해서 유형별로 나뉜 특정 민원에 대해 해당되면 'y', 그렇지 않으면 '' 값으로 구성되어 있다.

Input(민원 내용)을 살펴보면 분석에 필요 없는 정보들(숫자, 기호 등)이 많이 존재함을 알 수 있다. 따라서 다음 장에서는 이러한 불필요한 정보들을 데이터 탐색을 통해 제거하고 기타 전처리 과정을 실시한다.

1.2 데이터 탐색



[불필요한 정보 다루기]

실행 코드

```
# 불필요한 기호 및 숫자 데이터를 제거합니다.
remove_digits = str.maketrans('', '', digits)
x_senta = []

for i in arange(len(senta_x)):
    tmp = senta_x[i]
    tmp = tmp.translate(remove_digits)
    tmp = re.sub(r'[^\w]', ' ', tmp)
    x_senta.append(tmp.split())

x_senta = np.array(x_senta)

print(x_senta)
```

출력 결과

```
['배송의뢰', '시', '플라스틱으로', '되어', '있는', '햇빛가리개는', '안장에', '고정시켜', '박스로', '감싸', '포장했는데', '햇빛가리개는', '파손되고', '안장', '부분과', '차체는', '흰', '상태로', '배송되어', '피신청인에게', '이의제기하고', '배상을', '요구하였으나', '책임을', '회피함']
 ['이에', '신청인은', '자전거', '위에', '무거운', '물건을', '쌓아', '올려', '햇빛가리개가', '파손되고', '차체가', '흰', '것이라고', '주장하며', '공정위', '고시', '소비자분쟁해결기준에', '의거', '중고', '제품가액', '원', '의', '배상을', '요구함']
 ['신청인은', '피신청인', '몰테일', '의', '홈페이지에서', '이', '사건', '공기청정기', 'IQair', '를', '주문하고', '원을', '결제한', '사실이', '있는데', '피신청인', '가', '피신청인', '대신화물', '을', '통해', '배송', '하던', '중', '배송업자의', '과실로', '파손되어', '손해배상을', '요구한', '사건임']]
```

앞 장에서 살펴 본 것 같이 input(민원 내용)에는 민원발생 유형을 판단하는데 있어 불필요한 정보들(숫자, 기호 등)이 다수 존재한다. 이러한 불필요한 정보들을 제거하지 않으면 분석을 하는데 Error'를 일으킬 수 있기에 제거하겠다.

그 결과로 위의 출력 결과는 숫자와 기호를 제거되었다. 또한 그 결과값은 문장을 분석 단위인 '단어'로 나눴다.

실행 코드

```
# 민원 라벨 데이터를 생성합니다.
shape = senta_y.shape
y_senta = np.zeros(shape)
index_y = senta_y == 'y'
y_senta[index_y] = 1

# 생선된 라벨 데이터를 검증합니다.
check = np.apply_along_axis(sum,1,y_senta)
del_ind = where(check != 1)[0]
print(del_ind)
print(len(del_ind))

x_senta = np.delete(x_senta, del_ind, axis=0)
y_senta = np.delete(y_senta, del_ind, axis=0)

cls_dist = y_senta.sum(axis=0)
print(cls_dist)
```

출력 결과

```
30 # 검출된 문장 수
[ 178  179  180 ..., 2321 2329 2421]
```

이제 target(민원유형) 데이터에 대해 살펴보자. Target 데이터는 해당 민원유형에 대해 'y', 그렇지 않으면 ''로 구성되어있다. 따라서 분석의 효율성을 높이기 위해 'y'를 1, ''는 0으로 변환했다.

Target을 1 또는 0으로 변환 후 데이터의 정합성 여부를 확인해야 한다. 각 target은 반드시 하나의 1 값을 가지고 있어야 된다. 따라서 정합성 확인 규칙은 'target의 합=1'이 된다. 그 결과 정합성 규칙을 위배하는 문장이 총 30개가 나왔다. 이는 전체 문장 수(2728)에 비해 극히 일부분 이기 때문에 과감하게 제거하도록 한다.

 # 데이터의 정합성?

데이터를 여러 사람이 입력/삭제 등의 작업을 수행하게 되면, 데이터가 일치하지 않을 수 있습니다. 이 때 데이터의 정합성에 문제가

이제 데이터 탐색을 통해 정제된 데이터를 학습용과 테스트용으로 나누도록 하자. 아래 코드를 통해 총 문장의 70%를 학습 데이터 그리고 나머지 30%를 테스트 데이터로 지정한다.

실행 코드

```
# 학습 및 테스트 데이터를 생성합니다.
N_senta = len(x_senta)
N_train = int(ceil(N_senta * 0.80))
idx = np.random.choice(N_senta, N_train, replace=False)
x_senta_train = x_senta[idx]
y_senta_train = y_senta[idx]

test_idx = np.delete(arange(N_senta),idx)
x_senta_test = x_senta[test_idx]
y_senta_test = y_senta[test_idx]
```

random.choice 메서드를 통해 무작위로 데이터를 7:3으로 나눈다.

[단어 수 및 문장 길이 결정]

단어 수와 문장 길이에 대한 데이터 탐색에 앞서 '단어' 단위로 이루어진 분석 데이터에 numbering하는 과정이 필요하다. Numbering 규칙은 다양하게 존재하지만 본 분석에서는 '단어'의 counting 순서로(가장 많이 등장한 단어를 0으로 지정) 0부터 numbering을 실시하도록 하겠다.

 # 단어 수와 문장 길이에 대해 고려하는 이유?

간단히 말해 불필요한 정보를 제거하기 위해서이다 단어 수나 문장의 길이에 대한 고려없이 있는 그대로 모델링에 사용한다는 것은 불필요한 정보를 모델에 입력하는 것과 동일한 효과를 가지게 된다..

실행 코드

```
# 언어 데이터 전처리 과정
# 각 단어의 등장 횟수를 셉니다.
x_vocab = []

for i in arange(len(x_senta_train)):
    x_vocab.extend(x_senta_train[i])
print(x_vocab)

counter = collections.Counter(x_vocab)
count pairs = sorted(counter.items(), key=lambda x: (-x[1], x[0])) # <= Sort

# 각 단어에 고유 번호를 부여하고 해당 단어와 번호로 이루어진 사전을 만들고 단어 수 제한을 위한 데이터 탐색을 진행합니다.
chars, counts = zip(*count pairs)
vocab = dict(zip(chars, range(len(chars))))

print(len(chars))
print(count_pairs[0:10])
print(vocab)
```

출력 결과

```
9029 # 고유 단어 수

{'파손된것': 8386, '오기재했다는': 6490, '안된다는': 2199, '배송의뢰를': 4889, '전설이다': 7295, '신청인도': 5996, '반박함': 4553, '제기했으나': 7452, '구매함': 875, '높아': 3705, '과실을': 595, '올려': 6550, '상담원': 2083, '서면으로': 2104, '소비자는': 5707, '올림': 6553, '팀장에게': 8361, '네비게이션': 3682, '요구에도': 6606, '물품이라': 1895, '제품에': 344, '구겨': 3319, …, '네': 3681, '하이엔드오디오': 8626, '폐기하였는데': 8479, '항공운임을': 8706, '배송온다고': 4879, '쯤에도': 7814, '납부한': 3628, '회신을': 1589, '뜨고': 4089, '중고사이트를': 7640, '가능하면': 2873, '드라이버가': 4038, '먼저': 1855}
```

위는 총 9029개의 고유 단어에 대해 counting 후 numbering을 실시 한 결과이다. 위 결과를 탐색해보면 민원에서 자주 등장하는 단어들도 있지만 그렇지 않은 단어들도 존재한다. 데이터 분석 측면에서 민원에 자주 등장하지 않는(예: 전체 2728개의 문장에서 1번 또는 2번 등장) 단어들은 불필요한 정보이다.

따라서 본 분석에서는 총 9029개의 고유 단어 중 전체 문장에서 5번 미만으로 등장한 단어에 대해 Zero Padding을 실시한다.

Zero Padding이란 특정 데이터가 실제 모델 학습 시 영향을 미치지 않도록 Dummy값으로 대체하는 것을 뜻한다.

실행 코드

```
count_check = np.array(counts, dtype=int)
vocab_restric = sum(count_check >= 5)
print(vocab_restric)
```

출력 결과

```
860 # 유의미한 단어 수
```

전체 문장에서 5번 이상 등장하는 단어의 수는 총 860개이다. 따라서 860개를 제외한 나머지 '단어'들은 불필요한 정보로 간주 되어 각각에 대해 Zero Padding을 실시한다.

다음으로는 각 문장의 길이에 대해 데이터 탐색을 실시해 보자.

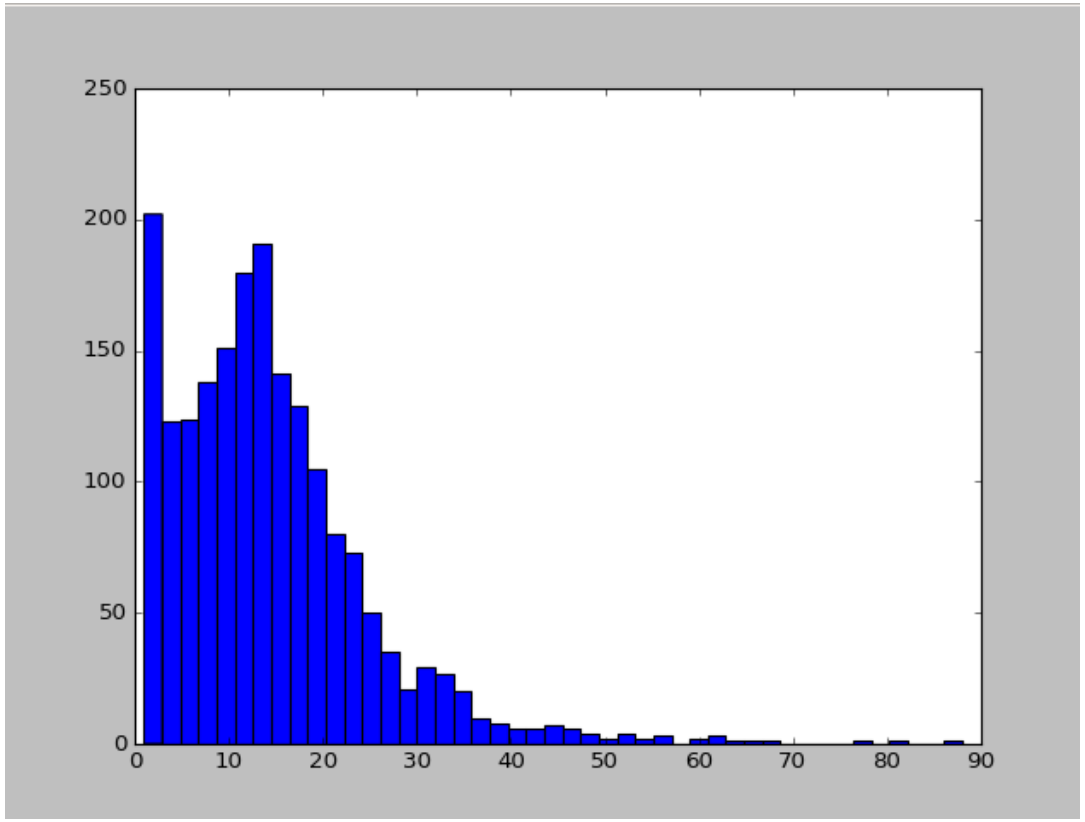
실행 코드

```
# 문장 길이 제한을 위한 데이터 탐색을 진행합니다.
length = []

for i in arange(len(x_senta_train)):
    length_each = len(x_senta_train[i])
    length.append(length_each)

plt.hist(length, bins='auto')
```

출력 결과



위 그림은 각 문장의 길이에 대한 분포를 나타낸 히스토그램이다. 히스토그램을 살펴보면 대부분의 문장은 0~40 개 사이의 단어들로 이루어 졌음을 알 수 있다. 실제 민원을 구성하는 문장들은 그 길이가 제 각각이다. 하지만 데이터 분석을 하기 위해서는 이러한 길이를 하나로 통일시킬 필요가 있다. 따라서 위 히스토그램을 바탕으로 본 분석에서는 문장의 길이를 30으로 지정하도록 한다.

실행 코드

```
vocab_size = len(vocab)
seq_length = 30

# 문장 제한 길이에 맞춰 불필요한 정보를 처리합니다.
corpus_train = []
for i in arange(len(x_senta_train)):
    tmp = list(map(vocab.get, x_senta_train[i]))
    tmp = tmp[:seq_length]
    tmp_len = len(tmp)
    num_add = seq_length - tmp_len
    if num_add > 0:
        add = [vocab_size-1] * num_add
        tmp.extend(add)
        tmp = np.array(tmp)
    else:
        tmp = np.array(tmp)
    corpus_train.append(tmp)

corpus_test = []
for i in arange(len(x_senta_test)):
    tmp = list(map(vocab.get, x_senta_test[i]))
    tmp = tmp[:seq_length]
    tmp_len = len(tmp)
```

```
    num_add = seq_length - tmp_len
    if num_add > 0:
        add = [vocab_size-1] * num_add
        tmp.extend(add)
        tmp = np.array(tmp)
    else:
        tmp = np.array(tmp)
    ind_None = [i for i,x in enumerate(tmp) if x == None]
    tmp[ind_None] = vocab_size - 1
    corpus_test.append(tmp)
```

문장의 길이가 30 미만인 문장에 대해서도 Zero Padding을 실시한다.

[배치 데이터 생성]

데이터 탐색을 통해 정제 된 데이터를 기반으로 모델 학습의 관점에서 효율성을 높이기 위해 배치 데이터를 생성한다. 사실 모델 학습 시 총 문장 2728개를 모두 사용하는 것이 정석이다. 하지만 모두 사용하게 될 경우 학습 속도와 용량에 큰 무리가 가기 때문에 전체 데이터를 적당한 값으로 여러 개로 나누어 모델을 학습시킨다.

본 분석에서는 기준이 되는 배치 사이즈를 100으로 정하였다.

실행 코드

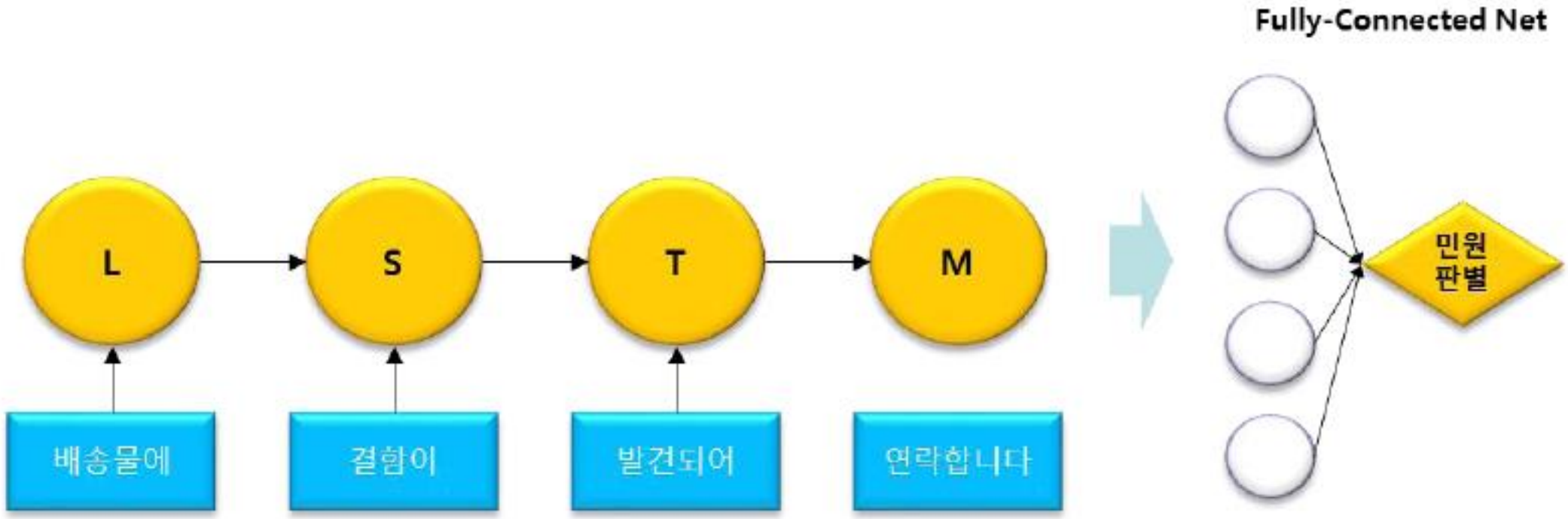
```
# 배치데이터를 생성합니다.
N_train  = len(corpus_train)
N_test = len(corpus_test)
batch_size = 100
num_batches_train = round(N_train/batch_size-0.5)
num_batches_test = round(N_test/batch_size-0.5)
num_cut_train = batch_size * num_batches_train
num_cut_test = batch_size * num_batches_test

train = np.array(corpus_train)
xbatches_train = np.split(train[:num_cut_train], num_batches_train, 0)
ybatches_train = np.split(y_senta_train[:num_cut_train], num_batches_train, 0)

test = np.array(corpus_test)
xbatches_test = np.split(test[:num_cut_test], num_batches_test, 0)
ybatches_test = np.split(y_senta_test[:num_cut_test], num_batches_test, 0)
```

```
# 학습 데이터 및 테스트 데이터 각각에 대해 배치 데이터 생성한다.
```

2. 딥러닝을 활용한 민원 자동분류기 생성



본 실습의 목적은 위 그림과 같이 해당 데이터를 기반으로 **'단어 혹은 문장' 다음에 위치할 '단어 및 문장'을 예측**하는 것이 핵심이다. 따라서 서로 연결되어 있는 언어들의 관계를 가장 잘 표현할 수 있는 분석기법에 **가장 적합한 'LSTM'으로 모형**을 활용할 것이다. 또한 LSTM을 통해 추출된 결과를 통해 클래스(민원)을 판별하는 것이 필요하다. 따라서 본 실습에서는 LSTM을 기반으로 Fully-Connected Net을 추가하여 분석을 진행하겠다.

[모델 파라미터 설정]

실행 코드

```
# LSTM 파라미터 값을 지정합니다.
rnn_size   = 100
grad_clip  = 5.
num_neuron = 50
num_classes = 6
emb_size = 35
```

```
# LSTM의 cell 개수 및 Fully-Connected Net의 neuron 개수 그리고 클래스 수 및 Word-embedding Size 지정한다.
```

[LSTM 구축]

실행 코드

```
# LSTM 모델을 정의합니다.
cell = tf.nn.rnn_cell.BasicLSTMCell(rnn_size, state_is_tuple=True)
input_data = tf.placeholder(tf.int32, [None, seq_length])
targets    = tf.placeholder(tf.float32, [None, num_classes])
istate     = cell.zero_state(batch_size, tf.float32)

# 가중치 및 편향을 정의하고 Word-Embedding 작업을 수행합니다.
with tf.variable_scope('RnnLm'):
    W = tf.get_variable("W",[rnn_size,num_neuron])
    b = tf.get_variable("b",[num_neuron])

    softmax_w = tf.get_variable("softmax_w", [num_neuron, num_classes])
    softmax_b = tf.get_variable("softmax_b", [num_classes])
    with tf.device("/cpu:0"):
        embedding = tf.get_variable("embedding", [vocab_restric, emb_size])
        emb_zeros = tf.zeros([vocab_size-vocab_restric,emb_size])
        embedding_add = tf.concat(0,[embedding,emb_zeros])
        inputs = tf.split(1, seq_length, tf.nn.embedding_lookup(embedding_add, input_data))
        inputs = [tf.squeeze(_input, [1]) for _input in inputs]

outputs, last_state = tf.nn.seq2seq.rnn_decoder(inputs, istate, cell
                , loop_function=None, scope = 'RnnLm')
last_outputs = outputs[-1]
```

```
# LSTM은 single-layer로 구축한다.
# emb_zeros를 통해 불필요한 단어들에 대한 Zero padding 실시한다.
```

[Fully Connected Network 구축]

실행 코드

```
# ReLU 함수를 정의합니다.
h_1 = tf.nn.relu(tf.matmul(last_outputs,W)+b)

# Softmax 함수를 정의합니다.
y = tf.nn.softmax(tf.matmul(h_1,softmax_w)+softmax_b)
```

```
# 2개의 hidden layer와 활성화 함수 ReLu 그리고 Softmax 사용한다.
```

[Optimizer 파라미터 설정]

실행 코드

```
# 학습을 위한 매개 변수를 정의합니다.
lr          = tf.Variable(0.0, trainable=False)
tvars       = tf.trainable_variables()
cross_entropy = tf.reduce_mean(-tf.reduce_sum(targets * tf.log(y), reduction_indices=[1]))
grads, _ = tf.clip_by_global_norm(tf.gradients(cross_entropy, tvars), grad_clip)
_optm       = tf.train.AdamOptimizer(lr)
optm        = _optm.apply_gradients(zip(grads, tvars))
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(targets,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

print ("Network Ready")
```

```
# lr(Learning Ratio)를 설정하여 학습 속도 조절한다.
```

[모델 학습]

실행 코드

```
# 모델을 학습합니다.
num_epochs    = 60
learning_rate = 0.002
decay_rate    = 0.97

sess = tf.Session()
sess.run(tf.initialize_all_variables())
init_time = time.time()

for epoch in range(num_epochs):
    sess.run(tf.assign(lr, learning_rate * (decay_rate ** epoch)))
    state     = sess.run(istate)
    batchidx  = 0
    for iteration in range(num_batches_train):
        start_time   = time.time()
        xbatch       = xbatches_train[batchidx]
        ybatch       = ybatches_train[batchidx]
        batchidx     = batchidx + 1
        train_cost, train_accuracy, _ = sess.run([cross_entropy, accuracy, optm]
            , feed_dict={input_data: xbatch, targets: ybatch, istate: state})
        total_iter = epoch*num_batches_train + iteration
        end_time     = time.time();
        duration     = end_time - start_time
        if total_iter % 100 == 0:
            print ("[%d/%d] train cost: %.4f / train accuracy: %.4f / Each batch learning took %.4f sec"
                  % (total_iter, num_epochs*num_batches_train, train_cost, train_accuracy, duration))
```

num_epochs를 통해 학습 횟수 조절, 총 학습 횟수는 num_epochs * batches_train(훈련데이터 배치 수)이다.

출력 결과

```
[0/720] train cost: 2.0356 / train accuracy: 0.1800 / Each batch learning took 0.2707 sec
[100/720] train cost: 1.0226 / train accuracy: 0.7000 / Each batch learning took 0.0176 sec
[200/720] train cost: 0.8983 / train accuracy: 0.6800 / Each batch learning took 0.0178 sec
[300/720] train cost: 0.7261 / train accuracy: 0.7600 / Each batch learning took 0.0232 sec
[400/720] train cost: 0.5153 / train accuracy: 0.8200 / Each batch learning took 0.0178 sec
[500/720] train cost: 0.5958 / train accuracy: 0.8700 / Each batch learning took 0.0176 sec
[600/720] train cost: 0.3317 / train accuracy: 0.9000 / Each batch learning took 0.0176 sec
[700/720] train cost: 0.2552 / train accuracy: 0.9100 / Each batch learning took 0.0177 sec
```

학습된 모델을 기반으로 테스트 데이터 셋에 대해 모델 평가를 실시한다.

실행 코드

```
# 모델을 테스트 하고 정확도를 추출합니다.
test_result = []
batchidx  = 0

for i in arange(num_batches_test):
    x_test = xbatches_test[batchidx]
    y_test = ybatches_test[batchidx]
    batchidx     = batchidx + 1
    tmp = sess.run(accuracy, feed_dict={input_data: x_test, targets: y_test})
    test_result.append(tmp)

Accuracy = mean(test_result) * 100
print ("Test Accuracy: %.4f" % (Accuracy))
```

출력 결과

```
Test Accuracy: 67.7500 %
```

최종적으로 모델의 Accuracy는 약 67%로 평가되었다. 이는 모델 학습 시 도출된 Train_Accuracy(약 91%)와 다소 차이가 있음을 알 수 있다. 물론 67%의 결과는 클래스가 6개라는 것을 고려하면 일반적인 문장 분류(2개 클래스의 경우 약 88%)와 비교하여 상당히 높은 수치이다. 그러나 여기서 주목하여야 할 점은 Train_Accuracy와 24% 이상 차이가 발생한다는 것이다. 이런 경우 다음 2가지를 체크할 필요가 있다.

```
# 체크 사항
  1) 테스트 데이터 셋 정합성 확인
  2) 오버피팅 발생 여부 확인
```

먼저 '2)' 의 경우 모델 파라미터 수를 줄이거나 드롭아웃(Drop Out)기법을 활용하여 모델을 재구축함으로써 확인 가능하다. 또한 재구축된 모델에 대한 성능 검토는 **Cross Validation을 통해 진행**해야한다. 하지만 그전에 본 실습에 쓰인 모델은 언어 모델링이기 때문에 반드시 먼저 확인해야 될 사항이 있다. 바로 **테스트 데이터 셋의 정합성 확인**이다.

'테스트 데이터셋 정합성 확인' 의 경우는 **언어 모델링에서 주로 체크**해야 되는 사항으로 테스트 데이터 셋에 대한 정밀한 데이터 탐색이 필요하다. 일반적인 언어 모델의 경우 훈련 데이터 셋의 크기가 작으면 여러가지 문제가 발생할 수 있다. 구체적으로 말하자면 학습에 사용되지 않은 '단어'가 테스트 셋에 존재하면 모델은 이를 '불필요한 정보'로 인지하게 된다.

그러나 테스트 셋에서는 오히려 해당 단어가 '유용한 정보'일 가능성이 있으므로 학습에 사용되지 않은 단어가 테스트 셋에 얼마나 포함되어 있는지에 대한 비율을 확인 할 필요가 있다. 아래는 이와 관련된 코드다.

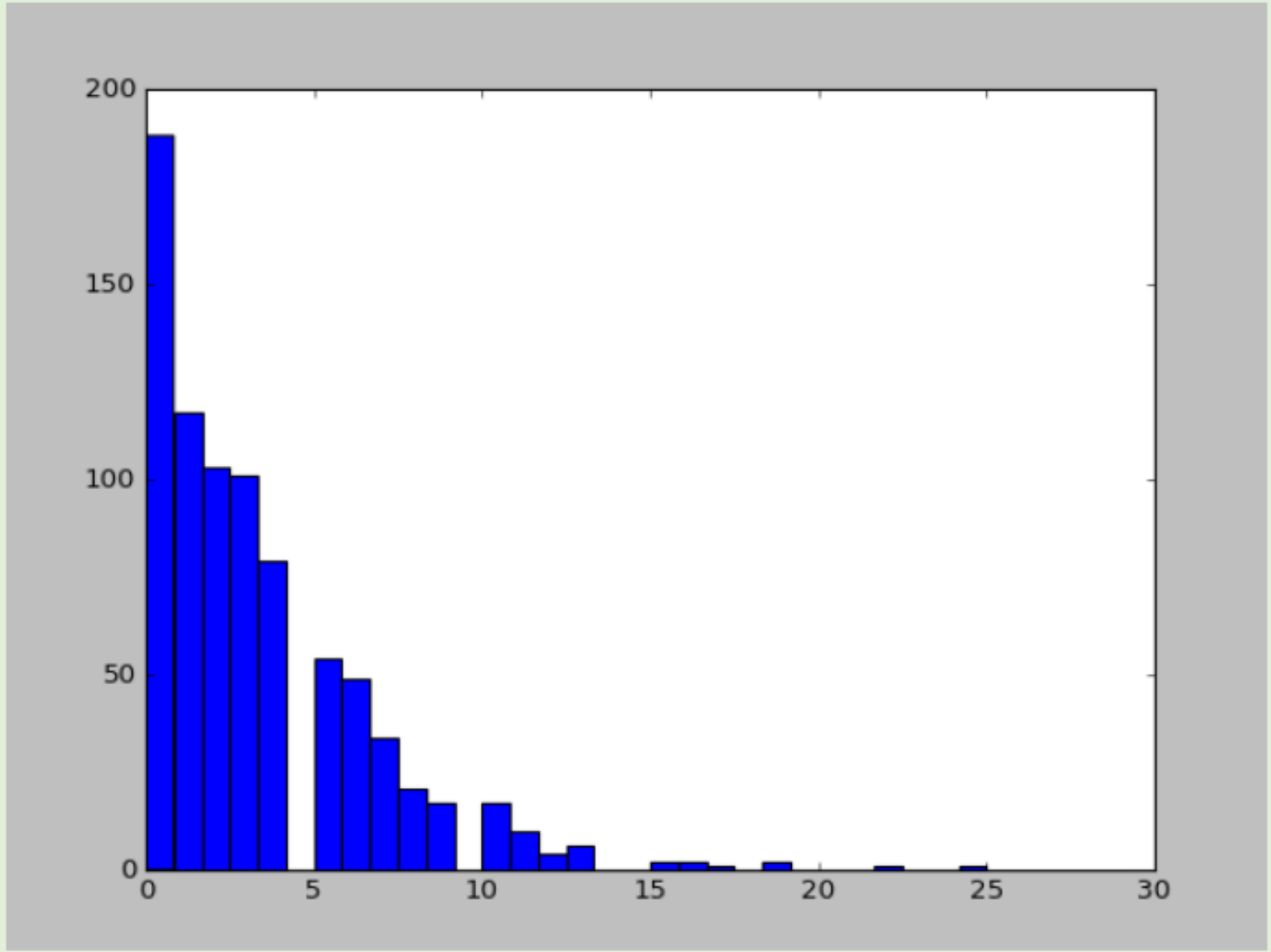
실행 코드

```
# 모델에 대한 피드백 과정을 수행합니다.
feedback = []
for i in arange(len(x_senta_test)):
    tmp = list(map(vocab.get, x_senta_test[i]))
    tmp = tmp[:seq_length]
    ind_None = [i for i,x in enumerate(tmp) if x == None]
    feedback.append(len(ind_None))
clf()
plt.hist(feedback, bins='auto')

none_check = np.array(feedback, dtype=int)
none_pro = sum(none_check >= 5)/len(none_check)
print(none_pro*100)
```

출력 결과

```
29.1718170581 # 해당 비율
```



위의 히스토그램은 테스트 셋을 이루는 각 문장에 대해 학습에 사용되지 않은 단어의 수를 나타냈다. 많은 문장들이 그 수가 5개 미만이지만 5개 이상인 문장들도 상당수 존재함을 알 수 있다.

5개 이상의 문장에 대해서 정확히 비율을 구하면 대략 30%가 되므로 Test Accuracy에 충분한 영향을 미칠 수 있다고 판단되었다. 따라서 Test Accuracy 향상을 위해서는 충분한 학습 데이터 셋을 확보하는 것이 무엇보다 중요하다고 판단된다. 이를 확인하기 위해 본 실습에서는 전체 데이터 셋을 절반으로 줄여 그 정확도를 확인해 보았다. 그 결과 '**사용되지 않은 단어가 테스트 셋에 얼마나 포함된 확률**'은 **35%까지 상승하였고 Test Accuracy 또한 60%로 약 8%로 가까이 떨어짐을 확인**할 수 있다. 아래는 해당 코드이다.

실행 코드

```
# 전체 데이터의 수를 조정합니다.
def Total(percent):
    N_senta = len(x_senta)
    N_total = int(ceil(N_senta * percent))
    total_idx = np.random.choice(N_senta, N_total, replace=False)
    x_result = x_senta[total_idx]
    y_result = y_senta[total_idx]
    return x_result, y_result

x_senta, y_senta = Total(0.5)
```

```
# 위의 함수를 활용하여 전체 데이터 셋을 절반으로 줄인 후 다시 전체 코드를 실행한다.
```