Sklearn Logistic Regression with Tf-Idf vectorization

```
from scipy import sparse
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import ConfusionMatrixDisplay, f1_score
from collections import Counter
import numpy as np
import operator
import nltk
import math
from scipy.stats import norm
import jieba
import regex as re
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
%cd "/content/drive/MyDrive/Info 159 Notebooks/Annotation Project - Leiden Weibo Corpus/AP4"
```

```
/content/drive/MyDrive/Info 159 Notebooks/Annotation Project - Leiden Weibo Corpus/AP4
```

```
stopwords_file = "stopwords-zh.txt"
stopwords = set(line.strip() for line in open(stopwords_file))
print(stopwords)
```

```
{'还是', '替', '那么些', '根据', '9', '乃', '若', '呵', '总而言之', '比方', '鄙人', '六', '之一', '出于', '人家', '接着', '此', '然而', '已矣
```

```
def load_data(filename):
    X = []
    Y = []
    with open(filename, encoding="utf-8") as file:
        for line in file:
            cols = line.split("\t")
            idd = cols[0]
            label = cols[2].lstrip().rstrip()
            text = cols[3]

            X.append(text)
            Y.append(label)

    return X, Y

def analyzer(text):
    text = "".join([n for n in re.findall(r'[\u4e00-\u9fff]+', text)])
    # text = "".join([n for n in text if n not in stopwords])
    text = jieba.lcut(text)
    return text

def confidence_intervals(accuracy, n, significance_level):
    critical_value=(1-significance_level)/2
    z_alpha=-1*norm.ppf(critical_value)
    se=math.sqrt((accuracy*(1-accuracy))/n)
    return accuracy-(se*z_alpha), accuracy+(se*z_alpha)


tf_idf = TfidfVectorizer(analyzer = analyzer)
trainingFile = "./splits/train.txt"
devFile = "./splits/dev.txt"
testFile = "./splits/test.txt"

trainX, trainY = load_data(trainingFile)
devX, devY = load_data(devFile)
testX, testY = load_data(testFile)

trainX = tf_idf.fit_transform(trainX)
devX = tf_idf.transform(devX)
testX = tf_idf.transform(testX)
```

```
Building prefix dict from the default dictionary ...
DEBUG:jieba:Building prefix dict from the default dictionary ...
Dumping model to file cache /tmp/jieba.cache
DEBUG:jieba:Dumping model to file cache /tmp/jieba.cache
Loading model cost 0.729 seconds.
DEBUG:jieba:Loading model cost 0.729 seconds.
Prefix dict has been built successfully.
DEBUG:jieba:Prefix dict has been built successfully.
```

```
print(tf_idf.vocabulary_)
```

{'为什么': 138, '每次': 1461, '不': 78, '在': 693, '中国': 130, '都': 2165, '会': 255, '他们': 235, '家': 880, '田鸡': 1670, '爱喝': 1601,

```
Cs = [1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100, 1000, 10000, 100000]
best_score = 0.
best_model = None
for C in Cs:
    log_reg = LogisticRegression(C=C, class_weight='balanced', max_iter=10000, random_state=159)
    model = log_reg.fit(trainX, trainY)
    score = log_reg.score(devX, devY)
    if score > best_score:
        best_model = model
        best_score = score
test_score = best_model.score(testX, testY)
print(test_score)
```

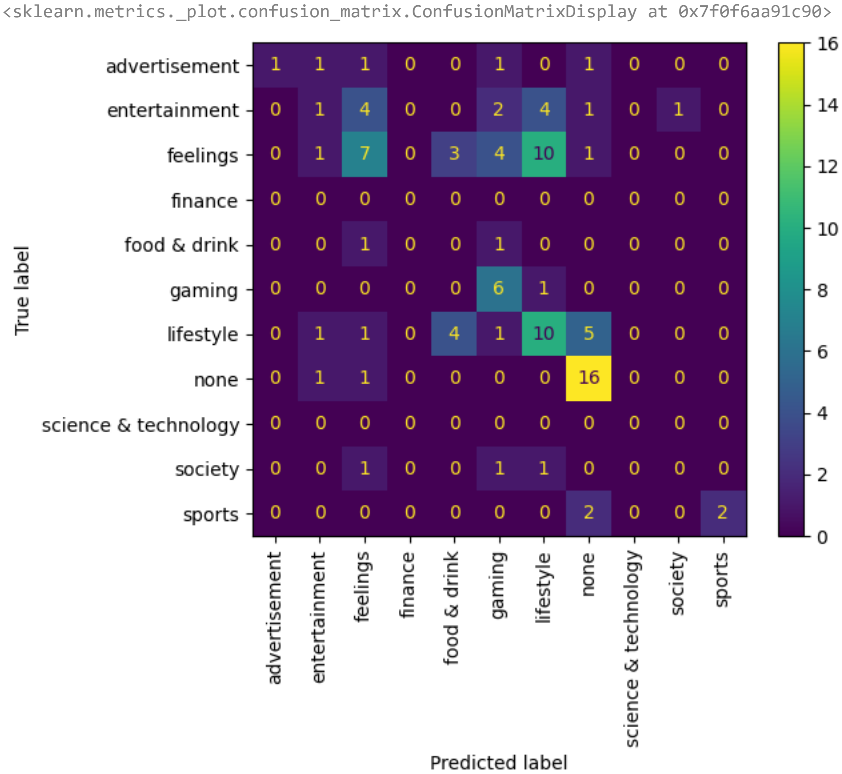
```
0.43
```

```
print(best_model.get_params())
lower, upper=confidence_intervals(test_score, len(testY), .95)
print(lower, test_score, upper)
```

```
{'C': 1, 'class_weight': 'balanced', 'dual': False, 'fit_intercept': True, 'intercept_scaling': 1, 'l1_ratio': None, 'max_iter': 10000,
0.3329669356893162 0.43 0.5270330643106838
```

```
ConfusionMatrixDisplay.from_estimator(best_model, testX, testY, labels=best_model.classes_, xticks_rotation='vertical')
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f0f6aa91c90>
```




```
reverse_vocab=[None]*len(best_model.coef_[0])
for k in tf_idf.vocabulary_:
    reverse_vocab[tf_idf.vocabulary_[k]]=k
for i, cat in enumerate(best_model.classes_):
```

```
    weights=best_model.coef_[i]

    for feature, weight in list(reversed(sorted(zip(reverse_vocab, weights), key = operator.itemgetter(1))))[:10]:
        print("%s\t%.3f\t%s" % (cat, weight, feature))
    print()
```

```
gaming  0.560   超过
gaming  0.551   刚刚

lifestyle       0.416   想
lifestyle       0.410   去
lifestyle       0.384   一个
lifestyle       0.353   在
lifestyle       0.345   不
lifestyle       0.320   呀
lifestyle       0.313   呢
lifestyle       0.309   说
lifestyle       0.306   喝多
lifestyle       0.294   了

none    1.248   图片
none    1.248   分享
none    0.489   阳朔
none    0.489   号线
none    0.468   喂
none    0.437   酷
none    0.388   准备
none    0.352   圖片
none    0.346   鹰
none    0.346   射手

science & technology    1.592   请
science & technology    1.380   软件
science & technology    1.380   整理
science & technology    1.380   微盘
science & technology    1.380   嘿嘿嘿
science & technology    1.281   电脑
science & technology    1.279   传到
science & technology    1.278   用
science & technology    0.952   上
science & technology    0.862   路上

society 1.764   了
society 1.334   太
society 0.953   珈
society 0.953   玮
society 0.923   浪漫
society 0.923   心心心
society 0.923   全是
society 0.887   喜糖
society 0.855   又
society 0.834   可怕

sports  2.256   斯科尔斯
sports  1.381   体育
sports  1.381   五星
sports  0.889   鲨鱼
sports  0.889   领先
sports  0.833   第轮
sports  0.806   啊
sports  0.725   号
sports  0.717   上海
sports  0.666   面料
```

```
scores = f1_score(testY, best_model.predict(testX), labels=best_model.classes_, average=None, zero_division=0)
for i, cl in enumerate(best_model.classes_):
    print(f"{cl}: {scores[i]}")
print("overall weighted: %f" % f1_score(testY, best_model.predict(testX), labels=best_model.classes_, average='weighted', zero_division=0))
```

```
advertisement: 0.33333333333333337
entertainment: 0.1111111111111111
feelings: 0.33333333333333337
finance: 0.0
food & drink: 0.0
gaming: 0.5217391304347825
lifestyle: 0.41666666666666663
none: 0.7272727272727274
science & technology: 0.0
society: 0.0
sports: 0.6666666666666666
overall weighted: 0.403542
```

Majority Class

```
from sklearn.dummy import DummyClassifier
```

```
trainingFile = "./splits/train.txt"
devFile = "./splits/dev.txt"
testFile = "./splits/test.txt"

trainX, trainY = load_data(trainingFile)
devX, devY = load_data(devFile)
testX, testY = load_data(testFile)

clf = DummyClassifier(strategy='most_frequent', random_state=159)
clf.fit(trainX, trainY)
majority_class = clf.score(trainX, trainY)
dev_score = clf.score(devX, devY)
test_score = clf.score(testX, testY)
print(f"majority class: {majority_class}, dev accuracy: {dev_score}, test accuracy: {test_score}")
```

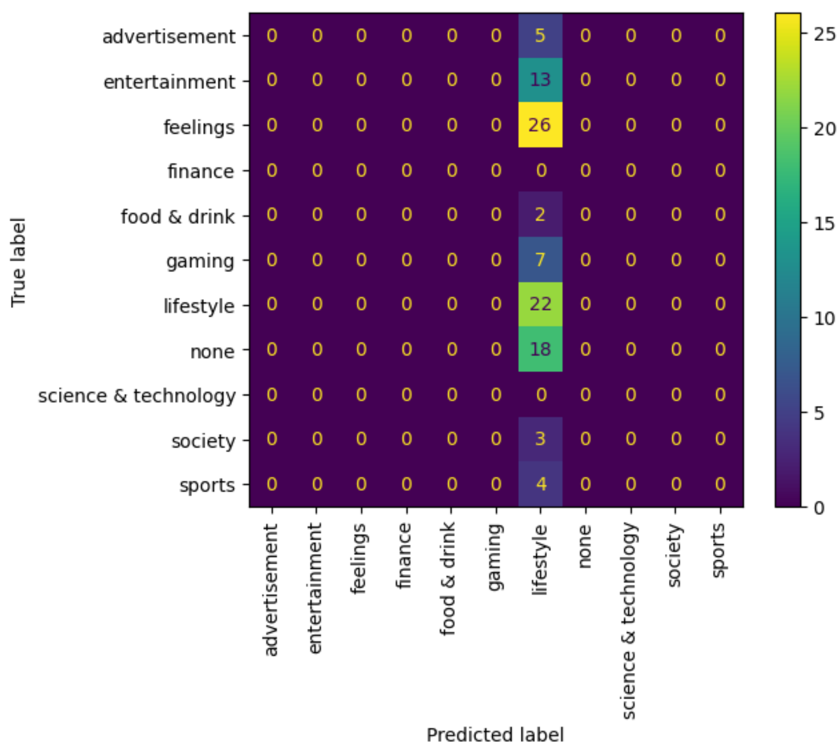
```
majority class: 0.24666666666666667, dev accuracy: 0.23, test accuracy: 0.22
```

```
lower, upper=confidence_intervals(test_score, len(testY), .95)
print(lower, test_score, upper)
```

```
0.13880921643246003 0.22 0.30119078356754
```

```
ConfusionMatrixDisplay.from_estimator(clf, testX, testY, labels=best_model.classes_, xticks_rotation='vertical')
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f7e0b0885e0>
```



TextCNN

```
!pip install fasttext
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting fasttext
  Downloading fasttext-0.9.2.tar.gz (68 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 68.8/68.8 kB 2.7 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Collecting pybind11>=2.2
  Using cached pybind11-2.10.4-py3-none-any.whl (222 kB)
Requirement already satisfied: setuptools>=0.7.0 in /usr/local/lib/python3.10/dist-packages (from fasttext) (67.7.2)
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from fasttext) (1.22.4)
Building wheels for collected packages: fasttext
  Building wheel for fasttext (setup.py) ... done
  Created wheel for fasttext: filename=fasttext-0.9.2-cp310-cp310-linux_x86_64.whl size=4393275 sha256=17127eb85bf8e75bfa7c63591eaddf654
  Stored in directory: /root/.cache/pip/wheels/a5/13/75/f811c84a8ab36eedbaef977a6a58a98990e8e0f1967f98f394
Successfully built fasttext
Installing collected packages: pybind11, fasttext
Successfully installed fasttext-0.9.2 pybind11-2.10.4
```

```
import os
os.environ['CUDA_LAUNCH_BLOCKING'] = "1"

from collections import Counter
import jieba
import regex as re
import fasttext.util
import torch
import torch.nn as nn
import torch.nn.functional as F
from sklearn.preprocessing import LabelEncoder
import numpy as np

import random
from scipy.stats import norm
import math
```

```
%cd "/content/drive/MyDrive/Annotation Project - Leiden Weibo Corpus/AP4"
```

```
[Errno 2] No such file or directory: '/content/drive/MyDrive/Annotation Project - Leiden Weibo Corpus/AP4'
/content/drive/MyDrive/Info 159 Notebooks/Annotation Project - Leiden Weibo Corpus/AP4
```

```
# uncomment the following line if its the first time downloading the fasttext model
# fasttext.util.download_model('zh', if_exists='ignore')
ft = fasttext.load_model('cc.zh.300.bin')
```

```
Warning : `load_model` does not return WordVectorModel or SupervisedModel any more, but a `FastText` object which is very similar.
```

```
def tokenizer(text):
    text = "".join([n for n in re.findall(r'[\u4e00-\u9fff]+', text)])
    text = jieba.lcut(text)
    return text

def load_data(filename):
    X = []
    Y = []
    with open(filename, encoding="utf-8") as file:
        for line in file:
            cols = line.split("\t")
            idd = cols[0]
            label = cols[2].lstrip().rstrip()
            text = cols[3]

            X.append(text)
            Y.append(label)

    return X, Y
def confidence_intervals(accuracy, n, significance_level):
    critical_value=(1-significance_level)/2
    z_alpha=-1*norm.ppf(critical_value)
    se=math.sqrt((accuracy*(1-accuracy))/n)
    return accuracy-(se*z_alpha), accuracy+(se*z_alpha)
```

```
trainingFile = "./splits/train.txt"
devFile = "./splits/dev.txt"
testFile = "./splits/test.txt"

trainX, trainY = load_data(trainingFile)
devX, devY = load_data(devFile)
testX, testY = load_data(testFile)

trainX = list(map(tokenizer, trainX))
devX = list(map(tokenizer, devX))
```

```
testX = list(map(tokenizer, testX))

vocab = Counter([word for sentence in trainX for word in sentence])
vocab_size = len(vocab)
print(vocab_size)
print(vocab)
print(trainX)
```

```
2348
Counter({'的': 263, '了': 182, '我': 131, '你': 93, '是': 74, '在': 61, '啊': 43, '都': 37, '有': 35, '就': 32, '也': 27, '不': 26, '吧':
[['为什么', '每次', '不', '在', '中国', '都', '会', '在', '他们', '家', '田鸡', '爱喝', '乙醇', '的', '甲醇'], ['小妞', '曼陀罗', '你', '厉害
```

```
def tokens_to_embeddings(tokens, vocab=vocab, ft=ft, embedding_dim=300, max_tokens=200):
    embeddings = []
    for i in range(max_tokens):
        if i < len(tokens):
            if tokens[i] in vocab.keys():
                embeddings.append(ft.get_word_vector(tokens[i]))
            else:
                embeddings.append(np.random.rand(embedding_dim))
        else:
            embeddings.append(np.zeros(embedding_dim))
    return np.array(embeddings)
```

```
trainX = np.array(list(map(tokens_to_embeddings, trainX)))
devX = np.array(list(map(tokens_to_embeddings, devX)))
testX = np.array(list(map(tokens_to_embeddings, testX)))
```

```
le = LabelEncoder()
trainY = le.fit_transform(trainY)
devY = le.transform(devY)
testY = le.transform(testY)
```

```
print(le.classes_)
print(len(le.classes_))
```

```
['advertisement' 'entertainment' 'feelings' 'finance' 'food & drink'
 'gaming' 'lifestyle' 'none' 'science & technology' 'society' 'sports']
11
```

```
print(trainY)
```

```
[ 6  4  7  6  5  6  4  1  1  6  6  6  6 10  2  5  2  7  5  5  5  1  6  7
  6  1  6  6  5  6  0  1  6  6  6  6  4  1  4  7  0  7  7  6  1  4  6  1
  4  1 10  7  7  7  6  6  6  1  6  6  6  1  6  5  6  7  5  6  5  6  4  0
  4  7  6  9  6  6  6  1  1  8  9 10  7  1  5 10  5  0  5  9  9  4  2  2
  6  7  7  7 10  7  2  2  2  2  5  2  2  2  2  2  0  2  0  2  2  2  2  5
  6  2  2  7  6  5  7  7  2  7  5  4  5  4  6  4  2  1  7  7  1 10 10  7
  2  6  2  2  5  2  4  2  0  2  2  5  1  2  0 10  1  2  1  5  2  0  2  6
  1  7  6  2  6  6  6  6  7  7  6  6  2  6  0  7  7  2  0  0  2  2  1  2
  7  0  2  3  7  1  6  1  9  2  2  0  2  6  6  4  6  7  2 10  6  5  6  2
  7  6  3  5  5  2 10  7  6  0  6  6  6  6  7  7  2  7  2  2  6  6  6  7
  6  6  5  2  4  6  6  6  6 10  0  4  1  2  6  6  9  2  2  2  4  1  0  2
  2  2  6  2  9  2  5  7  6  2  2  6  6  5  8  7  5  7  2  7  9  7  2  0
  4  4  1  7  1  2  6  2  2  4  4  5]
```

```
print(trainX.shape)
```

```
(300, 200, 300)
```

```
class TextCNN(nn.Module):
    def __init__(self, ):
        super(TextCNN, self).__init__()

        filter_sizes = [3,5,7,9,11]
        ch_in = 1
        num_filters = 400
        num_classes = 11
        dropout = 0.5
        embedding_dimension = 300

        # self.embedding = nn.Embedding(vocab_size, embedding_dimension)
        self.convs = nn.ModuleList(
```

```
            [nn.Conv2d(ch_in, num_filters, (size, embedding_dimension)) for size in filter_sizes])
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(len(filter_sizes) * num_filters, num_classes)

    def forward(self, x):
        # x = self.embedding(x)
        x = x.unsqueeze(1)
        x = [F.relu(conv(x)).squeeze(3) for conv in self.convs]
        x = [F.max_pool1d(item, item.size(2)).squeeze(2) for item in x]
        x = torch.cat(x, 1)
        x = self.dropout(x)
        logits = self.fc(x)
        return logits


def get_batches(all_x, all_y, batch_size=50):

    batches_x=[]
    batches_y=[]

    for i in range(0, len(all_x), batch_size):

        current_batch=[]

        batch_x=all_x[i:i+batch_size]

        batch_y=all_y[i:i+batch_size]

        batches_x.append(torch.FloatTensor(batch_x).cuda())
        batches_y.append(torch.LongTensor(batch_y).cuda())

    return batches_x, batches_y

def evaluate(model, x, y):
    model.eval()
    corr = 0.
    total = 0.
    with torch.no_grad():
        for x, y in zip(x, y):
            y_preds=model.forward(x)
            for idx, y_pred in enumerate(y_preds):
                prediction=torch.argmax(y_pred)
                if prediction == y[idx]:
                    corr += 1.
                total+=1
    return corr/total, total


import os
import sys
import torch
import torch.nn.functional as F


def train(train_x, train_y, dev_x, dev_y, model):
    model.cuda()
    batch_x, batch_y = get_batches(train_x, train_y)
    dev_batch_x, dev_batch_y = get_batches(dev_x, dev_y)

    optimizer = torch.optim.Adam(model.parameters(), lr=1e-5)

    cross_entropy=nn.CrossEntropyLoss()


    num_epochs=1000
    best_dev_acc = 0.
    patience=2000

    best_epoch=0

    for epoch in range(num_epochs):
        model.train()

        # Train
        for x, y in zip(batch_x, batch_y):
            x = x.cuda()
            y = y.cuda()
            logits = model(x)
```

```
            loss = cross_entropy(logits, y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # Evaluate
        train_accuracy, _= evaluate(model, batch_x, batch_y)
        dev_accuracy, _= evaluate(model, dev_batch_x, dev_batch_y)
        if epoch % 10 == 0:
            print("Epoch %s, train_accuracy: %.3f, dev accuracy: %.3f" % (epoch, train_accuracy, dev_accuracy))
            if dev_accuracy > best_dev_acc:
                torch.save(model.state_dict(), 'textcnn.pt')
                best_dev_acc = dev_accuracy
                best_epoch=epoch
        if epoch - best_epoch > patience:
            print("No improvement in dev accuracy over %s epochs; stopping training" % patience)
            break
    model.load_state_dict(torch.load('textcnn.pt'))
    print("\nBest Performing Model achieves dev accuracy of : %.3f" % (best_dev_acc))
    return model


model = TextCNN()
model = train(trainX, trainY, devX, devY, model)
```

```
Epoch 0, train_accuracy: 0.083, dev accuracy: 0.090
Epoch 10, train_accuracy: 0.457, dev accuracy: 0.230
Epoch 20, train_accuracy: 0.477, dev accuracy: 0.230
Epoch 30, train_accuracy: 0.493, dev accuracy: 0.240
Epoch 40, train_accuracy: 0.497, dev accuracy: 0.240
Epoch 50, train_accuracy: 0.513, dev accuracy: 0.240
Epoch 60, train_accuracy: 0.527, dev accuracy: 0.240
Epoch 70, train_accuracy: 0.533, dev accuracy: 0.240
Epoch 80, train_accuracy: 0.557, dev accuracy: 0.240
Epoch 90, train_accuracy: 0.577, dev accuracy: 0.240
Epoch 100, train_accuracy: 0.673, dev accuracy: 0.310
Epoch 110, train_accuracy: 0.703, dev accuracy: 0.310
Epoch 120, train_accuracy: 0.720, dev accuracy: 0.310
Epoch 130, train_accuracy: 0.730, dev accuracy: 0.320
Epoch 140, train_accuracy: 0.753, dev accuracy: 0.320
Epoch 150, train_accuracy: 0.777, dev accuracy: 0.330
Epoch 160, train_accuracy: 0.790, dev accuracy: 0.330
Epoch 170, train_accuracy: 0.813, dev accuracy: 0.330
Epoch 180, train_accuracy: 0.830, dev accuracy: 0.330
Epoch 190, train_accuracy: 0.843, dev accuracy: 0.330
Epoch 200, train_accuracy: 0.870, dev accuracy: 0.340
Epoch 210, train_accuracy: 0.883, dev accuracy: 0.340
Epoch 220, train_accuracy: 0.893, dev accuracy: 0.340
Epoch 230, train_accuracy: 0.900, dev accuracy: 0.340
Epoch 240, train_accuracy: 0.903, dev accuracy: 0.350
Epoch 250, train_accuracy: 0.917, dev accuracy: 0.350
Epoch 260, train_accuracy: 0.923, dev accuracy: 0.360
Epoch 270, train_accuracy: 0.923, dev accuracy: 0.360
Epoch 280, train_accuracy: 0.927, dev accuracy: 0.360
Epoch 290, train_accuracy: 0.937, dev accuracy: 0.350
Epoch 300, train_accuracy: 0.940, dev accuracy: 0.350
Epoch 310, train_accuracy: 0.943, dev accuracy: 0.350
Epoch 320, train_accuracy: 0.957, dev accuracy: 0.360
Epoch 330, train_accuracy: 0.957, dev accuracy: 0.360
Epoch 340, train_accuracy: 0.957, dev accuracy: 0.360
Epoch 350, train_accuracy: 0.967, dev accuracy: 0.360
Epoch 360, train_accuracy: 0.970, dev accuracy: 0.360
Epoch 370, train_accuracy: 0.970, dev accuracy: 0.370
Epoch 380, train_accuracy: 0.970, dev accuracy: 0.380
Epoch 390, train_accuracy: 0.973, dev accuracy: 0.380
Epoch 400, train_accuracy: 0.973, dev accuracy: 0.370
Epoch 410, train_accuracy: 0.977, dev accuracy: 0.380
Epoch 420, train_accuracy: 0.977, dev accuracy: 0.380
Epoch 430, train_accuracy: 0.980, dev accuracy: 0.380
Epoch 440, train_accuracy: 0.990, dev accuracy: 0.380
Epoch 450, train_accuracy: 0.990, dev accuracy: 0.380
Epoch 460, train_accuracy: 0.990, dev accuracy: 0.370
Epoch 470, train_accuracy: 0.990, dev accuracy: 0.380
Epoch 480, train_accuracy: 0.990, dev accuracy: 0.380
Epoch 490, train_accuracy: 0.990, dev accuracy: 0.380
Epoch 500, train_accuracy: 0.990, dev accuracy: 0.390
Epoch 510, train_accuracy: 0.990, dev accuracy: 0.380
Epoch 520, train_accuracy: 0.990, dev accuracy: 0.380
Epoch 530, train_accuracy: 0.990, dev accuracy: 0.390
Epoch 540, train_accuracy: 0.990, dev accuracy: 0.400
Epoch 550, train_accuracy: 0.990, dev accuracy: 0.390
Epoch 560, train_accuracy: 0.990, dev accuracy: 0.380
```

```
Epoch 570, train accuracy: 0.990, dev accuracy: 0.380
```

```
test_batch_x, test_batch_y = get_batches(testX, testY)
accuracy, test_n=evaluate(model, test_batch_x, test_batch_y)

lower, upper=confidence_intervals(accuracy, test_n, .95)
print("Test accuracy for best dev model: %.3f, 95%% CIs: [%.3f %.3f]\n" % (accuracy, lower, upper))
```

```
Test accuracy for best dev model: 0.380, 95% CIs: [0.285 0.475]
```

0s completed at 10:26 PM