L2-regularized logistic regression for binary or multiclass classification; trains a model (on `train.txt`), optimizes L2 regularization strength on `dev.txt`, and evaluates performance on `test.txt`. Reports test accuracy with 95% confidence intervals and prints out the strongest coefficients for each class.

```
from scipy import sparse
from sklearn import linear_model
from collections import Counter
import numpy as np
import operator
import nltk
import math
from scipy.stats import norm
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
!python -m nltk.downloader punkt
```

```
/usr/lib/python3.10/runpy.py:126: RuntimeWarning: 'nltk.downloader' found in sys.modules after import of package 'nltk', but prior to execution of 'n
  warn(RuntimeWarning(msg))
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

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

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

    return X, Y
```

```
class Classifier:

    def __init__(self, feature_method, trainX, trainY, devX, devY, testX, testY, vectorizer):
        self.feature_vocab = {}
        self.feature_method = feature_method
        self.min_feature_count=2
        self.log_reg = None

        self.trainY=trainY
        self.devY=devY
        self.testY=testY

        self.vectorizer = vectorizer
        self.trainX = self.process(trainX, training=True)
        self.devX = self.process(devX, training=False)
        self.testX = self.process(testX, training=False)

    # Featurize entire dataset
    def featurize(self, data):
        featurized_data = []
        for text in data:
            feats = self.feature_method(text)
            featurized_data.append(feats)
        return featurized_data

    def process(self, X_data, training=False):
       if training:
           self.vectorizer.fit(X_data)

       data = self.vectorizer.transform(X_data)
       return data


    # Read dataset and returned featurized representation as sparse matrix + label array
```

```
    #def process(self, X_data, training = False):

        #data = self.featurize(X_data)

        #if training:
            #fid = 0
            #feature_doc_count = Counter()
            #for feats in data:
                #for feat in feats:
                    #feature_doc_count[feat]+= 1

            #for feat in feature_doc_count:
                #if feature_doc_count[feat] >= self.min_feature_count:
                    #self.feature_vocab[feat] = fid
                    #fid += 1

        #F = len(self.feature_vocab)
        #D = len(data)
        #X = sparse.dok_matrix((D, F))
        #for idx, feats in enumerate(data):
            #for feat in feats:
                #if feat in self.feature_vocab:
                    #X[idx, self.feature_vocab[feat]] = feats[feat]

        #return X


    # Train model and evaluate on held-out data
    def train(self):
        (D,F) = self.trainX.shape
        best_dev_accuracy=0
        best_model=None
        for C in [0.1, 1, 10, 100]:
            self.log_reg = linear_model.LogisticRegression(C = C, max_iter=1000)
            self.log_reg.fit(self.trainX, self.trainY)
            training_accuracy = self.log_reg.score(self.trainX, self.trainY)
            development_accuracy = self.log_reg.score(self.devX, self.devY)
            if development_accuracy > best_dev_accuracy:
                best_dev_accuracy=development_accuracy
                best_model=self.log_reg

#             print("C: %s, Train accuracy: %.3f, Dev accuracy: %.3f" % (C, training_accuracy, development_accuracy))

        self.log_reg=best_model


    def test(self):
        return self.log_reg.score(self.testX, self.testY)


    def printWeights(self, n=10):

        feature_names = self.vectorizer.get_feature_names_out()

        reverse_vocab=[None]*len(self.log_reg.coef_[0])
        for k in self.feature_vocab:
            reverse_vocab[self.feature_vocab[k]]=k

        # binary
        if len(self.log_reg.classes_) == 2:
          weights=self.log_reg.coef_[0]

          cat=self.log_reg.classes_[1]
          for feature, weight in list(reversed(sorted(zip(feature_names, weights), key=operator.itemgetter(1))))[:n]:
             print("%sWt%.3fWt%s" % (cat, weight, feature))
          print()

          cat=self.log_reg.classes_[0]
          for feature, weight in list(sorted(zip(feature_names, weights), key=operator.itemgetter(1)))[:n]:
             print("%sWt%.3fWt%s" % (cat, weight, feature))
          print()

        # multiclass
        else:
          for i, cat in enumerate(self.log_reg.classes_):
             weights=self.log_reg.coef_[i]
```

```
            for feature, weight in list(reversed(sorted(zip(feature_names, weights), key=operator.itemgetter(1))))[:n]:
                print("%sWt%.3fWt%s" % (cat, weight, feature))
            print()
```

```
nltk.download('opinion_lexicon')

from nltk.corpus import opinion_lexicon


def binary_bow_featurize(text, vectorizer):
    feats = {}
    words = nltk.word_tokenize(text)

    tfidf_matrix = vectorizer.transform([text])
    word2tfidf = dict(zip(vectorizer.get_feature_names_out(), tfidf_matrix.toarray()[0]))

    #To assess capitalization of text (whether title contains at least one word that's in ALLCAP)
    ALLCAP = 0
    exclam = 0
    qmark = 0
    sarc_1 = 0
    sarc = set(['...', '??? ', ':)', ':(', 'lmao'])

    # Get the positive and negative words from the opinion lexicon
    positive_words = set(opinion_lexicon.positive())
    negative_words = set(opinion_lexicon.negative())

    # Add some custom negative words
    negative_words |= set(['not', 'no', 'never', 'hate', 'sad', 'Not worth', 'not good'])
    negative = 0
    # Add some custom positive words
    positive_words |= set(['awesome', 'fantastic', 'amazing', 'love', 'happy', 'fun', 'the best', 'finally made it'])
    positive = 0

    for word in words:
        word_lower = word.lower()
        tfidf_weight = word2tfidf.get(word_lower, 0)

        if word in negative_words:
          negative += tfidf_weight
        if word in positive_words:
          positive += tfidf_weight

        if word in sarc:
          sarc_1 = 1
        if word == '?':
          qmark = 1
        if word == '!':
          exclam = 1

        if len(word) != 1 and word == word.upper():
          ALLCAP = 1
        word=word.lower()
        feats[word]=1
    length = len(words)

    #Clarity
    if length > 10 or qmark == 1:
      feats['Clarity'] = 1

    #Sarcasm
    if exclam == 1 or sarc_1 == 1 or ALLCAP == 1:
      feats['Sarcasm'] = 1

    #Tone
    threshold = 0.5  # You can experiment with different threshold values
    if positive / (positive + negative) >= threshold:
        feats['positive'] = 1
    elif negative / (positive + negative) >= threshold:
        feats['negative'] = 1
    else:
        feats['neutral'] = 1
```

```
    #Q vs. Statement
    if qmark == 1:
      feats['QS'] = 1

    #Capitalization
    if ALLCAP == 1:
      feats['allcap'] = 1


    #Sentence Length
    if length <= 10:
        feats['length:0-10'] = 1
    elif length <= 20:
        feats['length:11-20'] = 1
    else:
        feats['length:20+'] = 1



    return feats
```

```
[nltk_data] Downloading package opinion_lexicon to /root/nltk_data...
[nltk_data]   Package opinion_lexicon is already up-to-date!
```

```
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)


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

    vectorizer = TfidfVectorizer(tokenizer=nltk.word_tokenize)
    vectorizer.fit(trainX)

    simple_classifier = Classifier(lambda text: binary_bow_featurize(text, vectorizer), trainX, trainY, devX, devY, testX, testY, vectorizer)
    simple_classifier.train()
    accuracy=simple_classifier.test()

    lower, upper=confidence_intervals(accuracy, len(testY), .95)
    print("Test accuracy for best dev model: %.3f, 95%% CIs: [%.3f %.3f]Wn" % (accuracy, lower, upper))

    simple_classifier.printWeights()



trainingFile = "train.txt"
devFile = "dev.txt"
testFile = "test.txt"

run(trainingFile, devFile, testFile)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/feature_extraction/text.py:528: UserWarning: The parameter 'token_pattern' will not be used since 'to
  warnings.warn(
Test accuracy for best dev model: 0.564, 95% CIs: [0.468 0.661]

Average 0.135   a
Average 0.072   ?
Average 0.068   's
Average 0.067   on
Average 0.064   )
Average 0.062   make
Average 0.061   not
Average 0.060   dpdr
Average 0.051   (
Average 0.049   start

Not popular     0.083   training
Not popular     0.083   ,
Not popular     0.059   stargate
```

```
Not popular     0.058   -
Not popular     0.058   homeless
Not popular     0.055   be
Not popular     0.055   down
Not popular     0.053   code
Not popular     0.052   wacom
Not popular     0.050   vs

Popular 0.171   with
Popular 0.086   .
Popular 0.080   !
Popular 0.064   climbing
Popular 0.063   ]
Popular 0.063   [
Popular 0.062   best
Popular 0.062   of
Popular 0.061   game
Popular 0.057   upload

label   0.067   text
label   0.067   original
label   0.027   the
label   -0.000  thanks
label   -0.000  sure
label   -0.000  shaving
label   -0.000  right
label   -0.000  place
label   -0.000  pics
label   -0.000  myself/my
```