LogReg

April 28, 2023

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.

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
%cd drive/My Drive/INFO 159
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/INFO 159
```

[2]:
```
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
```

[3]:
```
!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
'nltk.downloader'; this may result in unpredictable behaviour
  warn(RuntimeWarning(msg))
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

[4]:
```
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
```

[5]:
```
class Classifier:

    def __init__(self, feature_method, trainX, trainY, devX, devY, testX,␣
 ↪testY):
        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.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

    # 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):

        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(reverse_vocab,␣
↪weights), key = operator.itemgetter(1))))[:n]:
                print("%s\t%.3f\t%s" % (cat, weight, feature))
            print()

            cat=self.log_reg.classes_[0]
            for feature, weight in list(sorted(zip(reverse_vocab, weights),␣
↪key = operator.itemgetter(1)))[:n]:
                print("%s\t%.3f\t%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(reverse_vocab,␣
↪weights), key = operator.itemgetter(1))))[:n]:
                print("%s\t%.3f\t%s" % (cat, weight, feature))
            print()
```

[6]:
```
import nltk
nltk.download('wordnet')
from nltk.corpus import wordnet as wn
```

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

[7]:
```
from nltk.stem.porter import *
stemmer = PorterStemmer()
```

[8]:
```
def action_verbs(words, bigrams, trigrams, feats, stemmer):
  action_verbs = ['must', 'shall', 'will', 'shall', 'may',
                  'can', 'right', 'option', 'terminate', 'reasonable',
                  'good', 'faith', 'audit', 'examine', 'first',
                  'prior', 'own', 'expense', 'negotiate', 'valuable',
                  'commercially', 'irrevocable', 'perpetual', 'neither',
                  'liable', 'indirect', 'damage', 'waive', 'suit',
                  'action', 'claim', 'loss', 'demand', 'liability',
                  'cost', 'expense', 'written', 'notice', 'immediately'
                  'written', 'notice', 'agreement', 'time',
```

```
                'any', 'amended', 'without', 'penalty', 'fee',
                'not', 'obligation', 'request', 'notice', 'inspect',
                'inspection', 'auditing', 'responsible', 'partially',
                'partial', 'only', 'exclusive', 'exception', 'pay',
                'exceed', 'maximum', 'event', 'no', 'total',
                'worldwide', 'sublicensable', 'royalty', 'free',
                'non-exclusive', 'non', 'license', 'independent',
                'directly', 'indirectly', 'not', 'contain', 'limitation',
                'permitted', 'prohibited', 'consent', 'circumstance',␣
↪'restriction',
                'profit', 'entitled', 'recover', 'limited', 'seek', 'breach',
                'omission', 'penalty', 'access', 'non-transferable',␣
↪'cancellation',
                'transferable', 'limited', 'consent', 'notify', 'compliance',␣
↪''
                ]
synonyms = []
for verb in action_verbs:
  for syn in wn.synsets(verb):
      for l in syn.lemmas():
        synonyms.append(l.name())
action_verbs = list(set(action_verbs + synonyms))
action_verbs = set([stemmer.stem(verb.lower()) for verb in action_verbs])

for word in words:
  word = stemmer.stem(word.lower())
  if word in action_verbs:
    feats[word] = 1


bigrams_list = [
    ('only', 'responsible'), ('partially', 'permitted'),
    ('conditionally', 'responsible'), ('exclusive', 'right'),
    ('with', 'exception'), ('aggregate', 'liability'),
    ('exclusive', 'access'), ('non', 'transferable'),
    ('non', 'sublicensable'), ('non', 'exclusive'),
    ('no', 'restrictions'), ('prior', 'notice'),
    ('written', 'notice'), ('prior', 'consent'),
    ('damages', 'incurred'), ('liability', 'cap'),
    ('not', 'permitted'), ('independent', 'certified'),
    ('cancellation', 'fee'), ('good', 'faith'),
    ('own', 'expense'), ('may', 'terminate'),
    ('no', 'liability'), ('own', 'expense'),
    ('no', 'obligation')
]
bigrams_list = [(stemmer.stem(pairs[0]), stemmer.stem(pairs[1])) for pairs in␣
↪bigrams_list]
```

```
  for bi in bigrams:
    bi = (stemmer.stem(bi[0].lower()), stemmer.stem(bi[1].lower()))
    if bi in bigrams_list:
      feats[bi[0] + "_" + bi[1]] = 1


  trigrams_list = [
      ('unless', 'otherwise', 'agreed'), ('under', 'no', 'circumstances'),
      ('in', 'no', 'event'), ('at', 'no', 'time'),
      ('independent', '3rd', 'party'), ('independent', 'third', 'party'),
      ('any', 'and', 'all'), ('optional', 'prior', 'right')
  ]
  trigrams_list = [(stemmer.stem(triplet[0]), stemmer.stem(triplet[1]), stemmer.
 ↪stem(triplet[2])) for triplet in trigrams_list]

  for tri in trigrams:
    tri = (stemmer.stem(tri[0].lower()), stemmer.stem(tri[1].lower()),  stemmer.
 ↪stem(tri[2].lower()))
    if tri in trigrams_list:
      feats[tri[0] + "_" + tri[1] + "_" + tri[2]] = 1
  return feats
```

[9]:
```
def penalty_fee(words, bigrams, trigrams, feats, stemmer):
  penalty = ["penalty", "fee", "repercussion", "consequence", "breach",␣
 ↪"prohibited", "disallow", "concurrent",
           "entitle", "breach", "non-compliance"]
  audit = ["audit", "prior notice", "written notice", r"\d+ day notice"]
  synonyms = []
  for p in penalty:
    for syn in wn.synsets(p):
        for l in syn.lemmas():
          synonyms.append(l.name())
  updated_pen = list(set(penalty + synonyms + audit))
  updated_pen = set([stemmer.stem(p.lower()) for p in penalty])

  for word in words:
    word = stemmer.stem(word.lower())
    if word in updated_pen:
      feats[word] = 1
  return feats
```

[10]:
```
def binary_bow_featurize(text):
    feats = {}
    words = nltk.word_tokenize(text.lower())
    bigrams = list(nltk.bigrams(text.lower().split()))
    trigrams = list(nltk.trigrams(text.lower().split()))
```

```
    #lemmatizer = nltk.stem.WordNetLemmatizer()
    stemmer = PorterStemmer()

    feats = action_verbs(words, bigrams, trigrams, feats, stemmer)
    feats = penalty_fee(words, bigrams, trigrams, feats, stemmer)

    return feats
```

[11]:
```
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)
```

[12]:
```
def run(trainingFile, devFile, testFile):
    trainX, trainY=load_data(trainingFile)
    devX, devY=load_data(devFile)
    testX, testY=load_data(testFile)

    simple_classifier = Classifier(binary_bow_featurize, trainX, trainY, devX,␣
 ↪devY, testX, testY)
    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]\n" %␣
 ↪(accuracy, lower, upper))

    simple_classifier.printWeights()
```

[13]:
```
trainingFile = "splits/train.txt"
devFile = "splits/dev.txt"
testFile = "splits/test.txt"

run(trainingFile, devFile, testFile)
```

```
Test accuracy for best dev model: 0.624, 95% CIs: [0.529 0.718]

DP      0.507   non-exclus
DP      0.434   exceed
DP      0.378   permit
DP      0.337   sole
DP      0.308   non-transfer
DP      0.238   except
DP      0.228   liabil
DP      0.224   worldwid
```

```
DP      0.221   royalti
DP      0.220   licens

MP      0.588   exclus
MP      0.284   must
MP      0.280   term
MP      0.220   not
MP      0.205   ani
MP      0.204   take
MP      0.196   exclus_right
MP      0.189   pay
MP      0.183   expir
MP      0.173   case

NP      0.324   negoti
NP      0.289   liabl
NP      0.284   first
NP      0.267   may
NP      0.262   special
NP      0.231   have
NP      0.212   commerci
NP      0.191   set
NP      0.190   advis
NP      0.161   amend
```

[13]:

[13]: