OrdinalRegression+Analysis

April 28, 2023

[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
```

1 Model Training

Ordinal regression is a classification method for categories on an ordinal scale – e.g. [1, 2, 3, 4, 5] or [G, PG, PG-13, R]. This notebook implements ordinal regression using the method of Frank and Hal 2001, which transforms a k-multiclass classifier into k-1 binary classifiers (each of which predicts whether a data point is above a threshold in the ordinal scale – e.g., whether a movie is "higher" than PG). This method can be used with any binary classification method that outputs probabilities; here L2-regularizaed binary logistic regression is used.

This notebook 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.

[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_ordinal_data(filename, ordering):
    X = []
    Y = []
    orig_Y=[]
    for ordinal in ordering:
        Y.append([])

    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)

            index=ordering.index(label)
            for i in range(len(ordering)):
                if index > i:
                    Y[i].append(1)
                else:
                    Y[i].append(0)
            orig_Y.append(label)

    return X, Y, orig_Y
```

[5]:
```
class OrdinalClassifier:

    def __init__(self, ordinal_values, feature_method, trainX, trainY, devX,␣
 ↪devY, testX, testY, orig_trainY, orig_devY, orig_testY):
        self.ordinal_values=ordinal_values
        self.feature_vocab = {}
        self.feature_method = feature_method
        self.min_feature_count=2
        self.log_regs = [None]* (len(self.ordinal_values)-1)

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

        self.orig_trainY=orig_trainY
        self.orig_devY=orig_devY
        self.orig_testY=orig_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


    def train(self):
        (D,F) = self.trainX.shape


        for idx, ordinal_value in enumerate(self.ordinal_values[:-1]):
            best_dev_accuracy=0
            best_model=None
            for C in [0.1, 1, 10, 100]:
                log_reg = linear_model.LogisticRegression(C = C, max_iter=1000)
                log_reg.fit(self.trainX, self.trainY[idx])
                development_accuracy = log_reg.score(self.devX, self.devY[idx])
                if development_accuracy > best_dev_accuracy:
                    best_dev_accuracy=development_accuracy
                    best_model=log_reg


            self.log_regs[idx]=best_model

    def test(self):
        cor=tot=0
        counts=Counter()
        preds=[None]*(len(self.ordinal_values)-1)
        for idx, ordinal_value in enumerate(self.ordinal_values[:-1]):
            preds[idx]=self.log_regs[idx].predict_proba(self.testX)[:,1]

        preds=np.array(preds)

        for data_point in range(len(preds[0])):


            ordinal_preds=np.zeros(len(self.ordinal_values))
            for ordinal in range(len(self.ordinal_values)-1):
                if ordinal == 0:
                    ordinal_preds[ordinal]=1-preds[ordinal][data_point]
                else:
                   ␣
↪ordinal_preds[ordinal]=preds[ordinal-1][data_point]-preds[ordinal][data_point]

            ordinal_preds[len(self.
↪ordinal_values)-1]=preds[len(preds)-1][data_point]

            prediction=np.argmax(ordinal_preds)
            counts[prediction]+=1
            if prediction == self.ordinal_values.index(self.
↪orig_testY[data_point]):
                cor+=1
            tot+=1

        return cor/tot

    # Adding this method to output the actual predictions which the test␣
↪function does not
    def predict(self):
        cor=tot=0
        counts=Counter()
        preds=[None]*(len(self.ordinal_values)-1)
        for idx, ordinal_value in enumerate(self.ordinal_values[:-1]):
            preds[idx]=self.log_regs[idx].predict_proba(self.testX)[:,1]

        preds=np.array(preds)
        all_predictions = []
        org_labels = []

        for data_point in range(len(preds[0])):


            ordinal_preds=np.zeros(len(self.ordinal_values))
            for ordinal in range(len(self.ordinal_values)-1):
                if ordinal == 0:
                    ordinal_preds[ordinal]=1-preds[ordinal][data_point]
                else:
                    ␣
↪ordinal_preds[ordinal]=preds[ordinal-1][data_point]-preds[ordinal][data_point]

            ordinal_preds[len(self.
↪ordinal_values)-1]=preds[len(preds)-1][data_point]

            prediction=np.argmax(ordinal_preds)
            org_label = self.ordinal_values.index(self.orig_testY[data_point])
            org_labels.append(org_label)
            all_predictions.append(prediction)

        return all_predictions, org_labels
```

[6]:
```
# Import nltk, wordnet, and stemmer
import nltk
nltk.download('wordnet')
from nltk.corpus import wordnet as wn
from nltk.stem.porter import *
stemmer = PorterStemmer()
```

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

[7]:
```
# Match unigram, bigram, and trigram features after lowering and stemming
# For unigram features, wordnet was used to find synonyms to ensure those are␣
↪also matched when creating the bow

def action_verbs(words, bigrams, trigrams, feats, stemmer):

  # Matching unigram features and synonyms based on wordnet synoynms
  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


  # Matching bigram features
  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


  # Matching trigram features
  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
```

[8]:
```
# Additional unigram features

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
```

[9]:
```
# Create bow features based specific unigram, bigrams, and trigrams features we␣
 ↪have identifed as important keywords or phrases

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
```

[10]:
```
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)
```

[11]:
```
def run(trainingFile, devFile, testFile, ordinal_values):


    trainX, trainY, orig_trainY=load_ordinal_data(trainingFile, ordinal_values)
    devX, devY, orig_devY=load_ordinal_data(devFile, ordinal_values)
    testX, testY, orig_testY=load_ordinal_data(testFile, ordinal_values)

    simple_classifier = OrdinalClassifier(ordinal_values, binary_bow_featurize,␣
 ↪trainX, trainY, devX, devY, testX, testY, orig_trainY, orig_devY, orig_testY)
    simple_classifier.train()
    accuracy=simple_classifier.test()

    lower, upper=confidence_intervals(accuracy, len(testY[0]), .95)
    print("Test accuracy for best dev model: %.3f, 95%% CIs: [%.3f %.3f]\n" %␣
 ↪(accuracy, lower, upper))
```

[12]:
```
trainingFile = "splits/train.txt"
devFile = "splits/dev.txt"
testFile = "splits/test.txt"

# ordinal values must be in order *as strings* from smallest to largest, e.g.:
# ordinal_values=["G", "PG", "PG-13", "R"]

ordinal_values=["NP", "DP", "MP"]

run(trainingFile, devFile, testFile, ordinal_values)
```

```
Test accuracy for best dev model: 0.644, 95% CIs: [0.550 0.737]
```

2 Analysis

2.1 Analysis 1: Model Selection

In terms of model selection, we chose three classifiers to compare: BERT, logistic regression, and ordinal regression. When we used BERT with an embedding size of 768, we received “CUDA out of memory” error which can be attributed to having a big batch size and low GPU memory of the remote machine. Through trial and error, we have reached the conclusion that there was a tradeoff between computation time for memory usage especially when we had a bigger BOW dataset so we decided to rule this classifier out.

Next, we have tried using logistic regression which has resulted in an accuracy score of 62.4% and 95% confidence interval of [0.529, 0.718]. Then we opted in for the ordinal regression classifier as our labels [MP DP NP] are multicategorical as well as in order of how binding a given legal clause is. The accuracy score for ordinal regression was 64.4% with a 95% confidence interval of [0.550, 0.737]. Overall, we chose the ordinal regression as it incurred a higher accuracy score and had a narrower confidence interval making the model more reliable. Granted, the classifier’s ability in predicting the true labels didn’t differ significantly which we assume is due to the unclear boundary between our ordinal labels as ordinal regression assumes natural ordering between labels. We believe there are numerous cases where we fail to distinguish MP and DP (which will be discussed more in the confusion matrix) which renders our model insufficient to choose over our simple regression model. Minor point to note is that having a small dataset size also makes it practically harder to distinguish between the 3 labels.

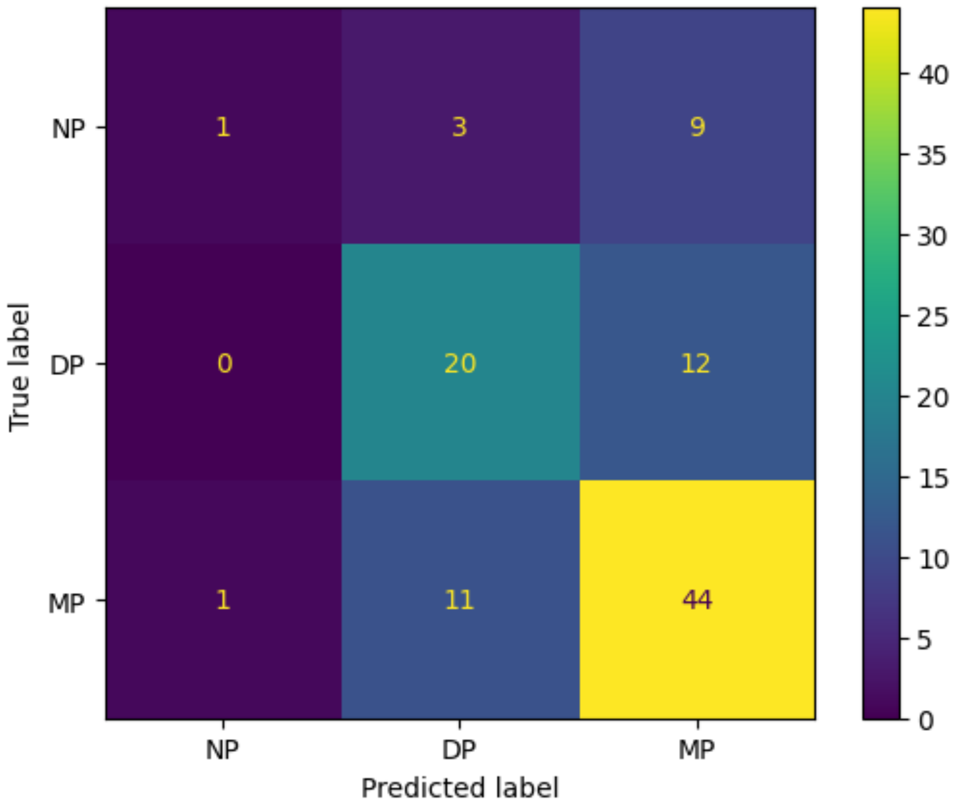
2.2 Analysis 2: Confusion Matrix

[13]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
trainX, trainY, orig_trainY=load_ordinal_data(trainingFile, ordinal_values)
devX, devY, orig_devY=load_ordinal_data(devFile, ordinal_values)
testX, testY, orig_testY=load_ordinal_data(testFile, ordinal_values)

simple_classifier = OrdinalClassifier(ordinal_values, binary_bow_featurize,␣
 ↪trainX, trainY, devX, devY, testX, testY, orig_trainY, orig_devY, orig_testY)
simple_classifier.train()
```

```
preds, org_labels=simple_classifier.predict()
```

```
[14]: from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
cm = confusion_matrix(org_labels, preds, labels=[0, 1, 2])
disp = ConfusionMatrixDisplay(confusion_matrix=cm,␣
 ↪display_labels=ordinal_values)
disp.plot();
```



Based on the confusion matrix above, we could analyze which labels the model often mistakens which other labels. Looking at the MP row first, it can be seen that the model does get most of the data points with a true label of MP correctly with 44/56 being correctly classified. However, 11/56 with true label as MP were classified as DP and 1/56 was classified as NP. This suggests that model may have some difficulty separating between the DP and MP classes.

However, for the DP category, it can be seen that the model often mistakens this class for the MP class as well with 12/32 being mistaken as such. It is also worth pointing out that none of the data points with true label DP were mistaken to be NP. This suggests that perhaps more guidelines should be present to more accurately differentiate between the DP and NP categories.

For the data points that have true label as NP, it can be seen most of these were actually classified as MP which is interesting considering these two categories are not adjacent in terms of the ordering of the ordinal values. This may be due to the fact that some data points have words or phrases that indicate both NP and MP which may make the model struggle classifying these correctly. This may also be due to the fact there may just be too few data points in the NP category so the model does not have enough data to recognize any patterns that might be in the NP category. This also might suggest that this category may need to be more defined in the guidelines as the model was only able to correctly classify 1/13.

2.3 Analysis 3: Feature Analysis

[1]:
```
# Refer to apendix at the end for the feature importance image
```

The feature weights above were gathered from running the logistic regression notebook with the same features we have defined in this ordinal regression notebook. Words from the training were stemmed which explains why the features above are stemmed as well. Looking at the MP category first, it can be seen that the three most important features for distinguishing this category are 'exclusive', 'must', and 'term'. Both 'exclusive' and 'must' align well with our guidelines since these are key words and phrases that we have defined in the guidelines to indicate a data point is more likely to be MP. However, the word 'term' was not really present in our guidelines which may indicate that the model has identified a pattern in predicting this class that we may not have accounted for in the guidelines itself.

For the DP category, the three most important features were 'non-exclusive', 'exceed' and 'permit'. For the 'non-exclusive' feature, this aligns well with our guidelines since this term is likely tied to the mention of a license which we have defined in the guidelines that it should be DP. Also the 'exceed' feature also aligns well with our guidelines since it likely indicates the presence of data point mentioning a liability cap which we have defined as the DP category in the guidelines.

For the NP category, the three most important features were 'negotiate', 'liability', and 'first'. The 'negotiate' features aligns well with our guidelines since it most likely indicates a potential room to change an agreement which is not very binding and we have defined this as the NP category. Also the 'liability' feature also aligns with our guidelines since it most likely indicates the lack of liability a party might have which we have defined as NP. On the other hand, the feature 'first' does not really align with our guidelines and it may be due to the model potentially fitting to noise in the NP category specifically since there are so few NP data points in training compared to the other categories.

2.4 Analysis 4: Class Balance

[15]:
```
import pandas as pd
train = pd.read_csv("splits/train.txt", sep='\t', header=None)
train[2].value_counts()
```

[15]:
```
MP    146
DP    110
NP     48
Name: 2, dtype: int64
```

[16]:
```
test = pd.read_csv("splits/test.txt", sep='\t', header=None)
test[2].value_counts()
```

```
[16]: MP    56
      DP    32
      NP    13
      Name: 2, dtype: int64
```

```
[17]: from sklearn.metrics import f1_score
      f1_score(org_labels, preds, average=None)
```

```
[17]: array([0.13333333, 0.60606061, 0.72727273])
```

In terms of class balance, it can be see from the values above that both the training and test datasets are fairly unbalanced. The MP class represent about 50% of the labels in both training and test set. This could have a significant impact in the model we built because the model would most likely pay more attention to labels of the MP category during training because it would improve performance the most. On the other hand, this also means the model may pay less attention to the NP category especially during training since it only represents only around 15% of the labels. Therefore, our model is biased towards predicting MP labels which could result in a poor accuracy overall. Our majority target class to minority target class is around a noticeable 3:1 in the training dataset and 4:1.5 in the testing dataset. We have calculated f1 scores for all our target classes and received 0.13 for NP, 0.61 for DP and 0.73 for MP. Consistent with our observation above, we see that NP has much lower f1 score compared to the other two labels which indicates that the model fails to accurately predict our minority class.

To adjust for the class imbalance problem, we could combine oversampling and changing class weights to our minority target class NP although we expect the effect of oversampling would be greater than adjusting class weights as our class imbalance is reasonably big. We could replicate our existing examples classified as NP as well as undersampling our results for MP thereby giving more representation for NP cases. Changing class weights could also help

2.5 Analysis 5: Implications

Overall based on our analysis, it seems as though the model particularly has the most trouble distinguishing between the DP and MP categories. This may indicate that our guidelines may need to be more well defined. If we were to change our guidelines, we may perhaps look at going through the training set again to define more clear rules to distinguish the two categories. Perhaps there may be more ambiguity than we had originally thought when creating the guidelines and annotating the data. Maybe making more clear rules in the guidelines to separate classes may lead to better overall model performance.

Putting all our results and analysis together, we believe law firms and legal departments could use our model to automate the review process of legal contracts to better determine which contracts to focus on. This could also be used for risk management as the more legally binding a contract is, the more risk associated with making decisions. As a whole, we believe our model could act as a general guideline for creating a metric to determine the legal consequences/impact of a contract.

```
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
```