Text Classification

By Waad ALMASRI

Introduction

- Text classification is the task of assigning one or more categories to a given piece of text from a larger set of possible categories
- Other terms: topic classification, text categorization, document categorization. NB: topic detection refers to the problem of extracting topics from texts & is different from text classification
- Example: spam/no-spam emails
- Text classification has a wide range of applications across diverse domains: social media, e-commerce, law, marketing, healthcare, etc.
- Text classification is the most used NLP task in the industry and the most researched in academia
- Text classification can be:
  - Binary → 2 classes, with the text belonging to exclusively one of these 2 classes
  - Multiclass → > n classes, n>2, with the text belonging to exclusively one of these n classes
  - Multilabel → n classes, n>1, with the text belonging possibly to 1 and/or more classes simultaneously

Applications

- Content classification and organization:
  - Use cases: content organization, search engines, recommendation systems
  - Data: news websites, blogs, online bookshelves, product reviews, tweets, tagging product descriptions in an e-commerce website, routing customer service requests in a company to the appropriate support team, organizing emails into personal, social and promotions in Gmail
- Customer support:
  - Identify actionable versus noisy tweets i.e. tweets that brands must respond to and those that do not require a response
- E-commerce:
  - Understand the customer's perception pf a product or a service based on the customers' reviews on e-commerce websites like Amazon, e-Bay, etc. → Sentiment analysis
- Language identification: like Google Translate
- ...

Pipeline for building text classification systems

1. Collect/create a labeled dataset
2. Split the dataset into either 2 (training and test) or 3 parts (training, validation and test sets
3. Decide on evaluation metrics (F1 score, precision, recall, area under ROC curve, business KPI, etc.)
4. Transform text into feature vectors
5. Training the pre-processed text and the labels
6. Benchmark the model performance on the test test using the pre-defined metrics
7. Deploy the model and monitor its performance

A simple text classifier without a text classification pipeline

- Problem: classify tweets into positive, negative or neutral
- Solution: lexicon-based sentiment analysis
  - Create a list of positive & negative words in English
  - Compare the usage of + vs - words in the input tweet and make a prediction
  - A more sophisticated solution: Create a dictionary with degrees of positive, negative & neutral sentiment of words or formulate specific heuristics (for example for emojis) and use them to make predictions
- NB:
  - Here, No learning is involved. The classification is based on rules/heuristics and custom-built resources.
  - This is a too simple approach to perform reasonably well for a real-world scenario
  - However, it allows an MVP (Minimum Viable Product) & easily understood

Using existing text classification APIs

If the task in hand is generic in nature, one can use an existing API:

- Google Cloud Natural Language: it provides classification models that can identify close to 700 text categories
- Sentiment analysis: it is provided by Google, Microsoft and Amazon

Text classification

For the rest of the chapter, we will use the “Economics News Article Tone and Relevance.”

It consists of ~8000 news article annotated.

The annotation is binary:

- yes: the article is relevant to the US economy with ~1500 samples
- No: it is irrelevant with ~ 6500 samples

→ Imbalanced dataset → the problem of bias towards the majority class

We will test 3 algorithms: Naive Bayes, logistic regression and support vector machines.

Section 1 - First Step - Solution

```
# convert label to a numerical variable
print(our_data.shape)
our_data = our_data[our_data.relevance != "not sure"] # removing the data where we don't want relevance="not sure".
print(our_data.shape)
our_data['relevance'] = our_data.relevance.map({'yes':1, 'no':0}) # relevant is 1, not-relevant is 0.
our_data = our_data[["text","relevance"]] # Let us take only the two columns we need.
our_data.shape
```

```
(8000, 15)
(7991, 15)

(7991, 2)
```

Section 2 - Solution

```
stopwords = stop_words.ENGLISH_STOP_WORDS
def clean(doc):
    """ this function returns a clean string of text; it removes the break tags, punctuations, stop words and digits.
    Parameters:
    -----------
    doc: String
        The text string

    Returns:
    --------
    doc: String
        The clean text string
    """
    # write your code here
    doc = doc.replace("</br>", " ") # This text contains a lot of <br/> tags.
    doc = "".join([char for char in doc if char not in string.punctuation and not char.isdigit()])
    doc = " ".join([token for token in doc.split() if token not in stopwords])
    return doc
```

Section 3 - Step 1 - solution

```
import sklearn
#from sklearn.cross_validation import train_test_split
from sklearn.model_selection import train_test_split

# Step 1: train-test split
X = our_data.text # the column text contains textual data to extract features from
y = our_data.relevance # this is the column we are learning to predict.
print(X.shape, y.shape)
# split X and y into training and testing sets. By default, it splits 75% training and 25% test
# random_state=1 for reproducibility
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=1)
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

```
(7991,) (7991,)
(5993,) (5993,)
(1998,) (1998,)
```

Section 3 -Steps 2 - Solution

Step 2: Preprocess and Vectorize

```
vect = CountVectorizer(preprocessor=clean) # instantiate a vectorizer
X_train_dtm = vect.fit_transform(X_train)# use it to extract features from training data
# transform testing data (using training data's features)
X_test_dtm = vect.transform(X_test)
print(X_train_dtm.shape, X_test_dtm.shape)
# i.e., the dimension of our feature vector is 49753!
```

```
(5993, 49753) (1998, 49753)
```

Do splitting the dataset into train/Test sets and pre-processing it need to follow any order?

Section 3 - Step 3 - Naive Bayes - Reminder

Naive Bayes: a probabilistic classifier based on the Naive Bayes theorem.

It estimates the conditional probability of each feature/variable of a given text for each class based on this feature's occurrence in that class.

Then, it multiplies the probabilities of all the features of a given text to compute the final probability of classification for each class. *

Finally, it chooses the class with maximum probability

Section 3 - Step 3 - Solution

Step 3: Train the classifier and predict for test data

```
nb = MultinomialNB() # instantiate a Multinomial Naive Bayes model
%time nb.fit(X_train_dtm, y_train) # train the model(timing it with an IPython "magic command")
```

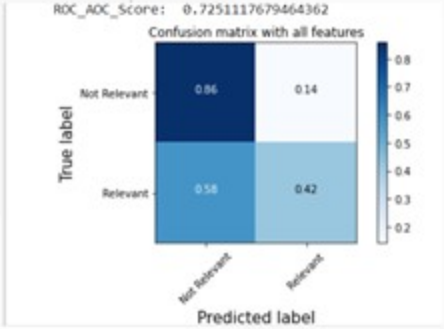
```
Wall time: 15.9 ms

MultinomialNB()
```

```
y_pred_class = nb.predict(X_test_dtm) # make class predictions for X_test_dtm
```

Section 3 - Step 4 - Solution

```
# Print accuracy:
print("Accuracy: ", accuracy_score(y_test, y_pred_class))

# compute the confusion matrix
cnf_matrix = confusion_matrix(y_test, y_pred_class)
# plot cnf_matrix
plt.figure(figsize=(6,4))
plot_confusion_matrix(cnf_matrix, classes=['Not Relevant','Relevant'],normalize=True,
                      title='Confusion matrix with all features')

# calculate AUC
y_pred_prob = nb.predict_proba(X_test_dtm)[:, 1]
print("ROC_AOC_Score: ", roc_auc_score(y_test, y_pred_prob))
```

```
Accuracy:  0.7822822822822822
ROC_AOC_Score:  0.7251117679464362
```



Reasons of poor performance

1. In fact, the number of features extracted is about 50 000 features → large and sparse feature vector → noisy features → poor learning

```
print(X_train_dtm.shape, X_test_dtm.shape)
```

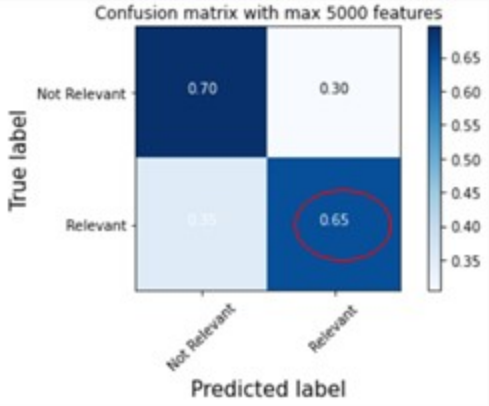
```
(5993, 49753) (1998, 49753)
```

1. Imbalance between classes → learning is biased towards the majority class
2. Inconvenient learning algorithm
3. Insufficient pre-processing and feature extraction mechanism
4. Hyperparameters of models needs further tuning

Improving Classification Performance by decreasing the number of feature vectors

```
vect = CountVectorizer(preprocessor=clean, max_features=5000) # Step-1
X_train_dtm = vect.fit_transform(X_train) # combined step 2 and 3
X_test_dtm = vect.transform(X_test)
# instantiate a Multinomial Naive Bayes model
nb = MultinomialNB()
# train the model(timing it with an IPython "magic command")
%time nb.fit(X_train_dtm, y_train)
# make class predictions for X_test_dtm
y_pred_class = nb.predict(X_test_dtm)
# print the model's accuracy
print("Accuracy: ", metrics.accuracy_score(y_test, y_pred_class))
# print the confusion matrix
cnf_matrix = confusion_matrix(y_test, y_pred_class)
plt.figure(figsize=(6,4))
plot_confusion_matrix(cnf_matrix, classes=['Not Relevant','Relevant'],normalize=True,
                      title='Confusion matrix with max 5000 features')
```

```
CPU times: user 4.91 ms, sys: 0 ns, total: 4.91 ms
Wall time: 4.93 ms
Accuracy:  0.6876876876876877
```



Improving Classification Performance by targeting imbalance data

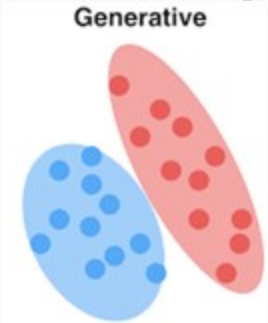
Two known solutions:

- Oversampling: duplicate existing data samples or create new synthetic data samples from the minority class https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html
- Undersampling: delete data samples from the majority class https://imbalanced-learn.org/stable/references/under_sampling.html

Inside the parenthesis: Generative vs Discriminative models

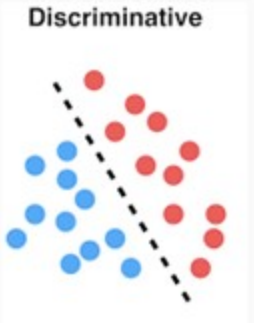
Generative model

- Probabilistic model of each class
- Can be used with unlabeled data
- Estimates from training data p(y) and p(x/y)
- Predicts using Bayes Theorem: p(y/x) = p(y)*p(x/y)/p(x)
- Examples:
  - Naive Bayes
  - Hidden Markov Models (HMMs)
  - Generative Adversarial Networks (GANs)



Discriminative model

- makes predictions on the unseen data based on conditional probability
- Can only be used with labeled data → supervised model
- Estimates from training data p(y/x)
- Examples: Logistic Regression

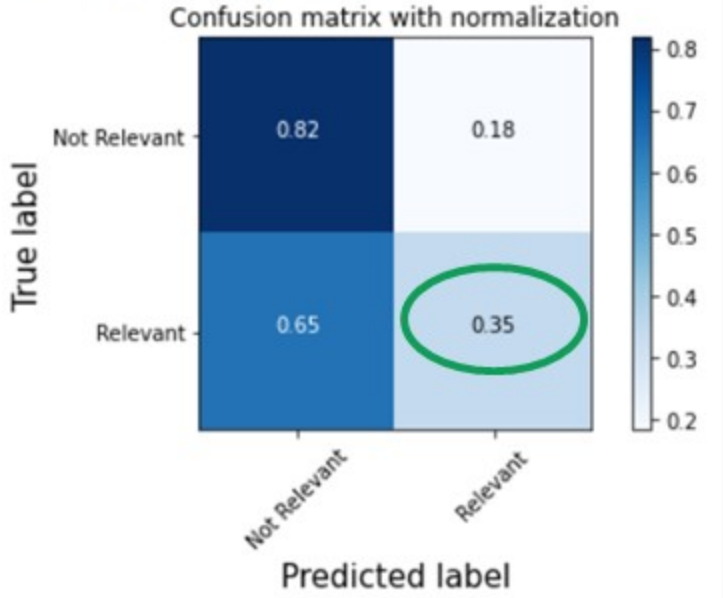


If we remove the argument:

`class_weight="balanced"`

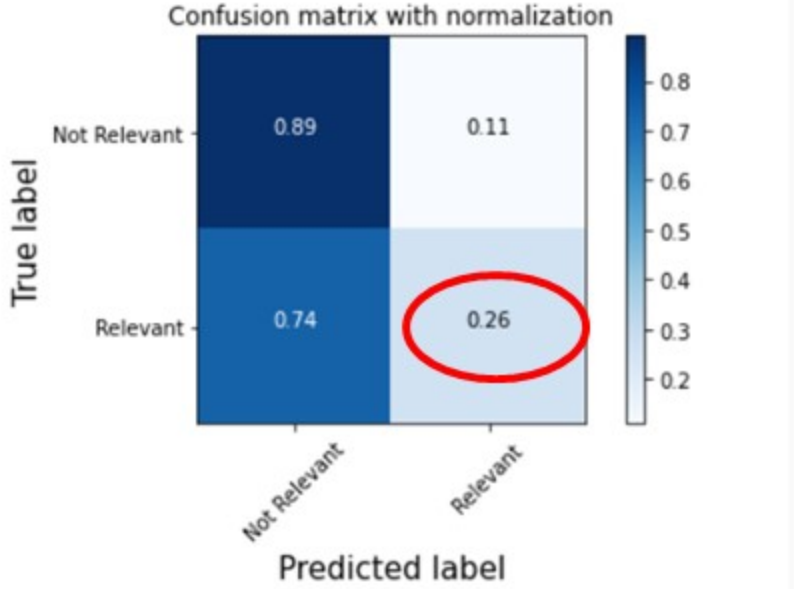
class_weight=”balanced”

Accuracy: 0.7362362362362362
AUC: 0.7251117679464362



class_weight=None

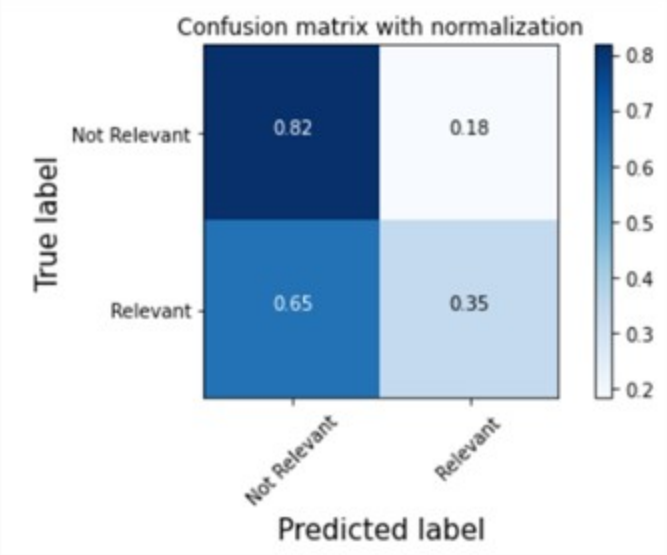
Accuracy: 0.7827827827827828
AUC: 0.7251117679464362



Improving Classification Performance by choosing a new learning algorithm that accounts to imbalance

```
# import the logistic regression model
from sklearn.linear_model import LogisticRegression

# instantiate a logistic regression model
logreg = LogisticRegression(class_weight="balanced")

# fit the model with training data
logreg.fit(X_train_dtm, y_train)

# Make predictions on test data
y_pred_class = logreg.predict(X_test_dtm)

# calculate evaluation measures:
print("Accuracy: ", accuracy_score(y_test, y_pred_class))
print("AUC: ", roc_auc_score(y_test, y_pred_prob))
cnf_matrix = confusion_matrix(y_test, y_pred_class)
plt.figure(figsize=(6,4))
plot_confusion_matrix(cnf_matrix, classes=['Not Relevant','Relevant'],normalize=True,
                      title='Confusion matrix with normalization')
```

```
Accuracy:  0.7362362362362362
AUC:  0.7251117679464362
```



Improving with a different classifier

What is the best learning classifier for my case?

In ML problems, there is no such best algorithm, one should try several approaches and several hyperparameters per approach and find the best one.

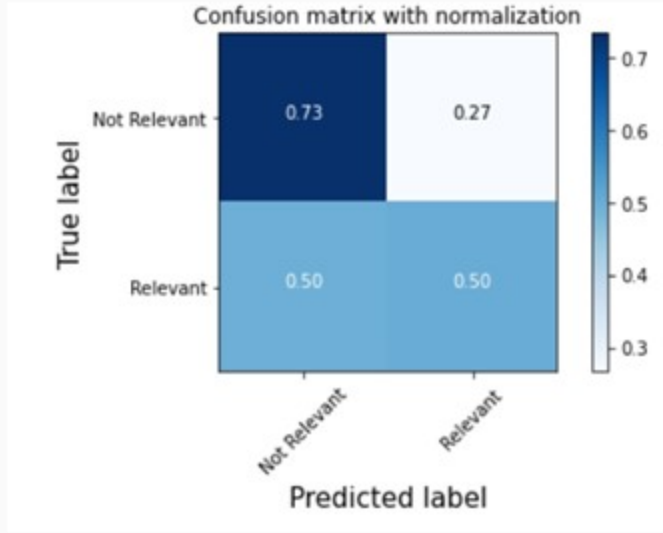
Also, the best algorithm might be a combination of several algorithms.

However, in a real-world text classification project, we usually start with the simplest approach and gradually increase the complexity.

Improving with a different classifier

```
from sklearn.svm import LinearSVC

vect = CountVectorizer(preprocessor=clean, max_features=1000) # Step-1
X_train_dtm = vect.fit_transform(X_train) # combined step 2 and 3
X_test_dtm = vect.transform(X_test)

classifier = LinearSVC(class_weight='balanced') # instantiate an SVM model
classifier.fit(X_train_dtm, y_train) # fit the model with training data

# Make predictions on test data
y_pred_class = classifier.predict(X_test_dtm)

# calculate evaluation measures:
print("Accuracy: ", accuracy_score(y_test, y_pred_class))
print("AUC: ", roc_auc_score(y_test, y_pred_prob))
cnf_matrix = confusion_matrix(y_test, y_pred_class)
plt.figure(figsize=(6,4))
plot_confusion_matrix(cnf_matrix, classes=['Not Relevant','Relevant'],normalize=True,
                      title='Confusion matrix with normalization')
```

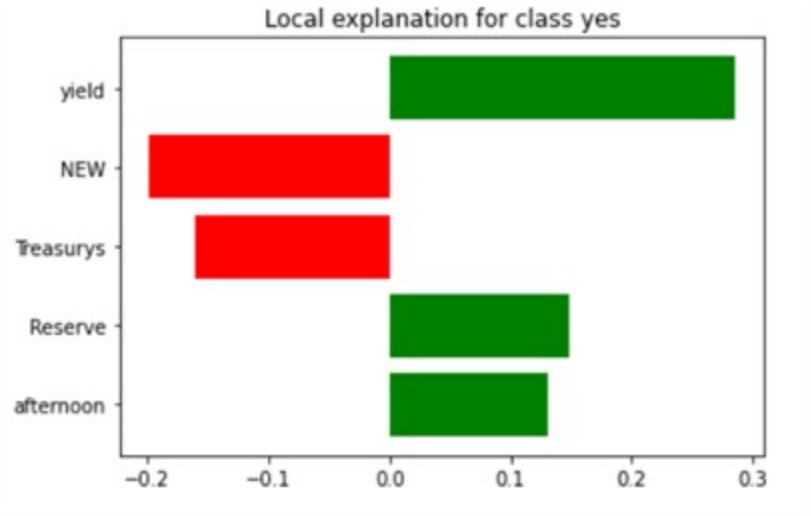
```
Accuracy:  0.6941941941941941
AUC:  0.7251117679464362
```

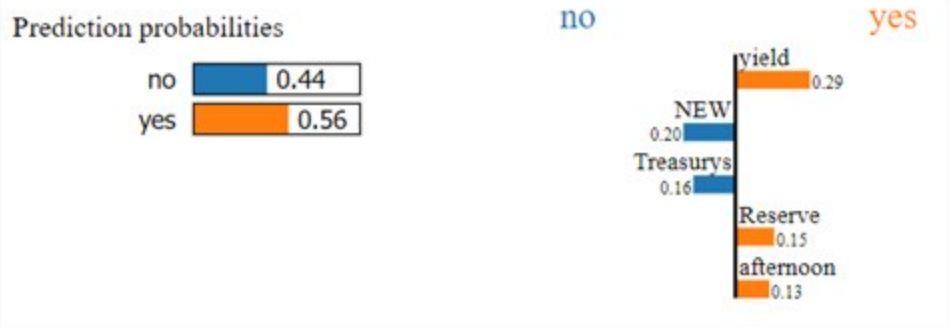


Interpreting Text Classification Models

- How do we explain the decisions our models are making?
- Why did they choose a class over the other?
- We need a method that provides some insights into the model and how it may perform on real-world data (instead of train/test sets), which may result in better, more reliable models in the future
- The ML models usage is increasing in the real-world applications → The model interpretability is no longer optional but necessary
- One famous method: Lime. It interprets a black-box classification model by approximating it with a linear model locally around a given training instance.
- Another method: Shapley

Using Lime to interpret model's classification





Improving using different text representation

Using neural embeddings

Next notebook uses word embeddings as features for text classification.

Data: sentiment-labeled sentences dataset with 1500 positive and 1500 negative sentiment sentences

Source: Amazon, Yelp and IMDB.

Results

Here, a pre-trained embeddings model was used.

In other use cases, we might need to train our own embeddings.

Rule to know when to train new embeddings: compute vocabulary overlap if >80%, then use pre-trained embeddings.

NB: loading embeddings can be computationally expensive and this criterion should be carefully considered

	precision	recall	f1-score	support
0	0.82	0.81	0.82	386
1	0.80	0.82	0.81	364
accuracy			0.81	750
macro avg	0.81	0.81	0.81	750
weighted avg	0.81	0.81	0.81	750

Learning with no or less data and adapting to new domains

No training data

Problem: classification problem of customer complaints into categories: billing, delivery and others.

Solution:

- Find a historical database in the company
- Train a model using this database

Another problem: This db does not exist.

No training data (2)

Creation of an annotated dataset with complaints alongside their categories

- Manual Labeling
- Bootstrapping or weak supervision
  - like Snorkel a software developed by Stanford
- Crowdsourcing
  - Like Amazon Mechanical Turk
  - Captcha test from Google

Small Training Dataset

Supposedly, we have collected some annotated data.

However, there is an issue of this labeled data preventing the training of a performant classifier:

- Insufficient amount of labeled data
- Labeled data turned to be imbalance

Problem: How should we handle such situations?
One solution: ask for additional labeled data → not so feasible
Other solutions: Active Learning & Transfer Learning depending on the data scenario

Active Learning

Scenario 1: we had 1000 data points and could get only 100 of them labeled, which 100 would we choose?
Solution: Active Learning identifies which data points are essential for training.

Using Active Learning to train a classifier is summarized as follows:

1. Train the classifier with available labeled data
2. Predict on new unlabeled data
3. For predictions with low confidence, send them for human annotators to get the correct labels
4. Include these data points to the existing labeled training set and retrain the model

Repeat steps 1 to 4, until reaching a good performance.

Transfer Learning (ULMfit)

Scenario 2: historical labeled data exist for some products. We are asked to work on newer products.
Solution: Transfer Learning or Domain adaptation

Using Transfer Learning in text classification is summarized as follows:

1. Start with a large, pre-trained language model trained on a large dataset of some source domain (the source domain should be a dataset that serves as closely as possible the problematic in hand)
2. Fine-Tuning the model using the target textual unlabeled dataset
3. Train a classifier on labeled target dataset using the fine-tuned language model from step 2

Public NLP datasets

- https://github.com/niderhoff/nlp-datasets
- Sentiment Analysis: Emotion in Text https://www.kaggle.com/c/sa-emotions
- UC Irvine Machine Learning Repository:https://archive.ics.uci.edu/ml/index.php
- Google datasets: https://datasetsearch.research.google.com/