Introduction

For the ENS Cachan MVA Master Course "Advanced Learning for Text and Graph Data" by professor Vazirgiannis of Ecole Polytechnique, a Kaggle competition was hosted on https://goo.gl/MSj7k5 (https://goo.gl/MSj7k5). The task was to to develop an email recipient recommendation system which, given the content and the date of a message, recommends a list of 10 recipients ranked by decreasing order of relevance. My username in the competition was AymanYACHAOUI and I achieved 0.37868 accuracy on the private leaderboard, ranking 22/58 teams and 4th as a one-membered team.

The data consists of Enron corporation emails sent by 125 senders and is organised as follows:

training_info.csv : for each email, is recorded its id (training_info$mids), its date (training_info$date), its content (training_info$body) and the list of emails it was sent to (training_info$recipients).
training_set.csv : for each of the 125 senders (training_set$sender), are listed the corresponding training_info email ids (training_set$mids).
test_info.csv : for each email, is recorded its id (test_info$mids), its date (test_info$date), its content (test_info$body). The recipients are to be predicted
test_set.csv : for each of the 125 senders (test_set$sender), are listed the corresponding test_info email ids (test_set$mids).

The corpus from which the data comes from belongs to the public domain and was the subject of a paper by Carvalho and Cohen [https://goo.gl/y5QRM7 (https://goo.gl/y5QRM7)]. What follows in an implementation of their base methods in Python 2.7. The reason I have chosen to implement the findings of the paper are simplicity of the approach and the unnecessity of cross-validation and parameter tuning since the authors proceeded to these and produce test curves for.

Data preprocessing

Let's load the previous datasets into memory

In [1]:
```
import pandas as pd
import numpy as np
training = pd.read_csv('training_set.csv', sep=',', header=0)
training_info = pd.read_csv('training_info.csv', sep=',', header=0)
testing = pd.read_csv('test_set.csv', sep=',', header=0)
test_info = pd.read_csv('test_info.csv', sep=',', header=0)
```

The first step undertaken by the authors was using a non-repeated email database. Let's check for double emails on the Kaggle data. I use the SequenceMatcher of the difflib package to check for ratio of same chars on two strings. We will check for the first 20 chars and view the most similar email bodies

In [ ]:
```
from difflib import SequenceMatcher
def similar(a, b):
    return SequenceMatcher(None, a, b).ratio()
```

In [ ]:
```
max_similarity = 0
flag=False
for i in xrange(training_info.shape[0]):
    for j in xrange(training_info.shape[0]):
        if(j>i):
            similarity = similar(training_info['body'].iloc[i][:20],training_info['body'].iloc[j][:20])
            flag = True if similarity > max_similarity else False
            if(flag):
                max_similarity = similarity
                L=[i,j]
print 'Most-similar emails by '+str(max_similarity)+': '
print training_info['body'].iloc[L[0]]
print training_info['body'].iloc[L[1]]
if(max_similarity < 0.6):
    print 'No duplicate emails found'
```

The second inconsistency the authors dealt with was multiple email accounts for the same person. Here we focus on the last name part of the 125 email senders and distinguish 125 different results. Therefore we can assume that the Kaggle data quality meets the authors'.

The authors proceed by considering two distinct sets of messages: messages sent by the user (sent collection) and messages received by the user (received collection).
For the former, we will proceed to this splitting on train and test data.
For the latter, it is important to note as did Nathan Rouxel on the competition forum (https://goo.gl/vjhjZq (https://goo.gl/vjhjZq)) that several invalid email addresses are hidden in the recipients fields. Competition admin Antoine recommended keeping only .*@.* shaped addresses.

In [7]:
```python
#The sent collection
sent_col_train = {}
for row in training.itertuples():
    user = row[1]
    mids = row[2].split(' ')
    sent_col_train[user] = [int(i) for i in mids]

sent_col_test = {}
for row in testing.itertuples():
    user = row[1]
    mids = row[2].split(' ')
    sent_col_test[user] = [int(i) for i in mids]

#The received collection and the recepients by email
rec_col = {}
rec_mid = {}
for row in training_info.itertuples():
    mid = row[1]
    recepients = row[4].split(' ')
    for recepient in recepients:
        if '@' in recepient:
            if recepient in rec_col.keys():
                rec_col[recepient].append(mid)
            else:
                rec_col[recepient] = [int(mid)]
            if mid in rec_mid.keys():
                rec_mid[mid].append(recepient)
            else:
                rec_mid[mid] = [recepient]
```

They also define the address book AB as the recipients that were addressed in the messages for each user. We further add occurence of the contact and sort by it. We begin by extracting all recepients by email ids using the recepients by email dictionnary (rec_mid) by iterating over all user,email_id contained in the sent collection of the user (sent_col_train). We obtain a list of lists of contacts for each user that we flatten before creating an entry dictionnary for the user key and the frequency ranked contacts.

In [39]:
```python
import operator
from collections import Counter
AB = {}
for user, mids in sent_col_train.iteritems():
    user_contacts = []
    for mid in mids:
        user_contacts.append(rec_mid[mid])
    user_contacts = [contact for sub_user_contacts in user_contacts for contact in sub_user_contacts]
    contact_vs_count = dict(Counter(user_contacts))
    contact_vs_count = sorted(contact_vs_count.items(), key=operator.itemgetter(1), reverse = True)
    AB[user] = contact_vs_count
```

Data Modeling

To model the textual information inside the email messages, the authors put forward two methods:
Tf-Idf Centroid : For every email in the sent collection, the normalized Tf-Idf vector is derived. These were summed up for every recepient in the address book exchanged by two users in an oriented way; creating for each recepient a Tf-Idf centroid. For prediction, cosine similarity between centroid vectors and the Tf-Idf vector of the testing emails were used to select the 10 most likely receivers.
K-NN 30 : For an email, the 30 closest emails are retrieved. Similarity distance is once again defined as the cosine distance between pairs of normalized Tf-Idf vectors. The weight of each recepient is then computed depending on the sum of similarity scores and ranked acordingly. The 10 most relevant are then filtered.
In the TO+CC+BCC -which is the case of our data challenge- prediction task, the difference between TfIdf-Centroid and Knn-30 methods is not statistically significant, whereas in the CC+BCC prediction task Knn-30 seems to outperform TfIdf-Centroid significantly. I have therefore chosen to implement the KNN30 method.

Generating the Tf-Idf bag of words

During my trials, I noticed slightly better performance after stemming - I used WordNet Lemmatizer - but regular expression validation of english words didn't bring enhancements. What follows is the procedure.

```
In [49]: from nltk.stem import WordNetLemmatizer
         wnl = WordNetLemmatizer()
         from nltk.corpus import stopwords
         stop = set(stopwords.words('english'))
         train_features = {}
         for i in range(len(training_info)):
             bow = [word for word in training_info.iloc[i,2].lower().split(' ') if (word not in stop )]
             bow = [wnl.lemmatize(t) for t in bow]
             train_features[training_info.iloc[i,0]] = " ".join(bow)
             #print temp_train.iloc[i,1]

         test_features = {}
         for i in range(len(test_info)):
             bow = [word for word in test_info.iloc[i,2].lower().split(' ') if (word not in stop )]
             bow = [wnl.lemmatize(t) for t in bow]
             test_features[test_info.iloc[i,0]] = " ".join(bow)
```

KNN 30 Learning

In [70]:
```
senders_adrs = AB.keys()
Mostsimilar50 = {}
Mostsimilar50_P = {}


for sender_id in xrange(125):
    user_sent_col_train =  sent_col_train[senders_adrs[sender_id]]
    user_sent_col_test = sent_col_test[senders_adrs[sender_id]]

    #Get the user sent emails features
    user_features_train = []
    user_features_test = []
    for key in user_sent_col_train:
        user_features_train.append(train_features[key])
    for key in user_sent_col_test:
        user_features_test.append(test_features[key])
    #Transform the user sent email features to tf_idf vects
    count_vectorizer = CountVectorizer(ngram_range=(1, 2))
    count_vectorizer.fit(user_features_train)

    user_matrix_train = count_vectorizer.transform(user_features_train)
    user_matrix_test = count_vectorizer.transform(user_features_test)


    tfidf = TfidfTransformer(norm="l2")
    user_matrix_train = tfidf.fit_transform(user_matrix_train).toarray()
    user_matrix_test = tfidf.transform(user_matrix_test).toarray()
    #Compute Distances and pick 50 most similar
    #This yielded better results than the paper's 30
    Neighbors = {}
    for pred in range(len(user_matrix_test)):
        pred_features = user_matrix_test[pred,:]
        pred_features_postiv = np.argsort(-pred_features)
        pred_features_postiv = pred_features_postiv[0:sum(pred_features!=0)]
        distance = pred_features[pred_features_postiv].dot(user_matrix_train[:,pred_features_postiv].tran
spose())
        distance = np.argsort(-distance)
        idx = distance[0:50]
        tmp = {}
        tmp['mids'] = [user_sent_col_train[j] for j in idx]
        tmp['scores'] = [distance[j] for j in idx]
        Neighbors[user_sent_col_test[pred]] = tmp
    #Compute the weight of each recipient ri in |AB(u)| according to the sum of similarity scores
    #of the messages (from the top 50 messages)
    p = re.compile('.*@.*')
    for mid in user_sent_col_test:
        Mostsimilar50_P[mid] = []
        candidates = [mail[0] for mail in AB[senders_adrs[sender_id]]]
        for candidate in range(len(candidates)):
            score = 0
            if not p.match(candidates[candidate]):
                print candidates[candidate]
                score = -1
                Mostsimilar50_P[mid].append((candidates[candidate],score))
                continue
            for neighbor in range(len(Neighbors[mid]['mids'])):
                if candidates[candidate] in rec_mid[Neighbors[mid]['mids'][neighbor]]:
                    score += Neighbors[mid]['scores'][neighbor]
            Mostsimilar50_P[mid].append((candidates[candidate],score))

    #Sort recepients and pick top-10
    for mid in user_sent_col_test:
        candidates = [mail[0] for mail in Mostsimilar50_P[mid]]
        scores = [score[1] for score in Mostsimilar50_P[mid]]
        idx = np.argsort(-np.array(scores))
        Mostsimilar50[mid] = [candidates[i] for i in idx[0:10]]
```

Generate Predictions and save as csv file

In [72]:
```
with open('predictions.csv', 'wb') as writer:
    writer.write('mid,recipients' + '\n')
    for mid, recipients in Mostsimilar50.iteritems():
        writer.write(str(mid) + ',' + ' '.join(recipients) + '\n')
```