Francesco Pontiggia, Jagmeet Sarna 6925480, 6848370 Data Mining 2019: 1/11/2019
Retake 10/01/2020

Report

Assignment 2: Text classification for the Detection of the Opinion Spam

Introduction

The habit of reviewing products and services on online platforms (Tripadvisor, Yelp, Rate Your Music, ...) is increasing very quickly. In parallel, according to (Cone, 2011) users are relying more and more on others' reviews when it comes to make a decision. Due to that, not all the reviews are truthful, but some of them are deliberately manufactured to sound authentic by malicious people who want to gain more customers against competitors. We call these fictitious reviews DECEPTIVE OPINION SPAM. Furthermore, we distinguish between them depending on the general *sentiment* a review expresses: positive reviews express satisfaction about the product or the service, while negative reviews express disappointment. We are aware that this distinction excludes all the possible intermediate evaluations, so it's a strong limit for our analysis (following the normal categorization of web reviews, we are considering only 5-star and 1-star or 2-star reviews).

Previous works have already tried to build automatic classifiers which help to find out deceptive reviews and showed that they perform better than chance and human judge. In particular, Ott et al. (2011) focused on positive deceptive reviews, while Ott et al. (2013) focused on negative deceptive reviews. An important contribution of these researches is the creation of two datasets, respectively of gold-standard deceptive positive reviews and gold standard deceptive negative reviews. Indeed, the biggest obstacle to analysis before these two papers was the absence of sample data (reviews in our case) which can represent in an unbiased manner the deceptive reviews, since it's complicated even to identify (at a certainty level of 100%) deceptive reviews, for obvious reasons. Thanks to that, further analysis is now possible exploiting those data.

We are trying now to improve the performance of the linear classifiers (naïve Bayes and Support Vector Machine) of the two previous works exploring different possibilities via more flexible classifiers and trying to build the best models via hyperparameters tuning. Due to simplicity reasons, we will focus only on negative reviews, leaving the positive ones to future work.

Our experiment is then divided into four parts, and each part takes into account a model. In details, we have:

1. Naïve Bayes
2. Regularized logistic regression
3. Classification tree
4. Random forests

For each model, we consider two sets generated by the reviews:

1. Only the unigrams
2. Unigrams and bigrams (set of all couples of two consecutive words)

In order to allow the reader to reproduce the experiments, all the code can be found in the Appendix parts.

The data

As introduced before, we are exploiting the work of Ott et al. (2013) and using their data. Truthful reviews were collected from the following reviewing websites: Expedia, Hotels.com, Orbitz, Priceline, Tripadvisor and Yelp. Although they can't be considered gold standard data, Mayzlin et al. (2012) and Ott et al. (2012) suggest that deception rate among those data is acceptably small. On the other side, deceptive opinion spam has been generated using Amazon's Mechanical Turk

service. The quality of these reviews has been proved by showing the actual difficulty of human testers in identifying them.

In details, the dataset is composed only of negative reviews (800). Data is equally divided into deceptive opinion spam (400) and truthful reviews (400). Each of these two sets is then divided into 5 folders. We use folders 1:4 of both deceptive and positive reviews (640 reviews in total) as training set, while we use folder 5 (160) as test set.

The reviews regard 20 popular hotels of Chicago, so that each hotel has 20 truthful reviews and 20 deceptive reviews. The truthful reviews were sampled according to a log normal distribution fit to the lengths of the deceptive reviews, since truthful reviews are on average longer than deceptive reviews. For further information about the collecting procedure and about Mechanical Tusk, refer to (Ott et al., 2013), the original paper.

Setup of the experiments

All the experiments have been conducted in R language. In order to do that, it's necessary to install an environment which allows to run R code. We have used Rstudio. Some additional packages have been installed to exploit their functions in the analysis. The following commands should then be typed first in the command line.

```
install.packages("tm")
install.packages("entropy")
install.packages("randomForest")
install.packages("randomForest")
install.packages("rpart")
install.packages("rpart.plot")
install.packages("glmnet")
```

Data can be downloaded from the following website:

https://myleott.com/op-spam.html

Then the following variables have been created in the Rstudio workspace. The four variables represent respectively the corpus of deceptive reviews in the training set, the corpus of the truthful reviews in the training set, the corpus of deceptive reviews in the test set, the corpus of truthful reviews in the test set. Be careful to substitute in the commands the real address of the folder which contains the data (usually, the Download folder).

```
training.corpus.dec <- c("C:/--address of the folder --
op_spam_v1.4/negative_polarity/deceptive_from_MTurk/fold1",
"C:/--address of the folder --
op_spam_v1.4/negative_polarity/deceptive_from_MTurk/fold2",
"C:/--address of the folder --
op_spam_v1.4/negative_polarity/deceptive_from_MTurk/fold3",
"C:/--address of the folder --
op_spam_v1.4/negative_polarity/deceptive_from_MTurk/fold4")

training.corpus.true<- c("C:/ --address of the folder --
op_spam_v1.4/negative_polarity/truthful_from_Web/fold2",
"C:/--address of the folder --
op_spam_v1.4/negative_polarity/truthful_from_Web/fold2",
"C:/--address of the folder --
op_spam_v1.4/negative_polarity/truthful_from_Web/fold3",
"C:/--address of the folder --
op_spam_v1.4/negative_polarity/truthful_from_Web/fold4")

testing.corpus.dec <- "C:/--address of the folder --
op_spam_v1.4/negative_polarity/deceptive_from_MTurk/fold5"
```

```
testing.corpus.true <- "C:/ --address of the folder --
op_spam_v1.4/negative_polarity/truthful_from_Web/fold5"
```

First, with the use of Natural language processing, we clean the data. We use the `tm` package, to

- Convert the entire text to lower case
- Remove numbers
- Remove whitespaces
- Remove punctuation
- Remove stopwords

Stopwords are the more common used words in a language, like articles. They can be considered "noise" for our goal, so it's better to remove them (setting the language as "english", since we are using only reviews in English language).

In order to do this cleaning, we create a new Rscript with a function we call `cleaning.function`, to reuse it more times. The code of this function can be found in *Appendix1*. It unites the two matrices but does not mix deceptive and truthful reviews and returns a list of documents of words.

Description of the experiments

Naïve Bayes

For order purposes, we create a Rscript called `Naïve Bayes` and put all the code inside the function `naïve.bayes.function`. We clean the data using the cleaning function already mentioned, we extract all the unigrams, we remove all the words which appear in no more than 5% of the documents ("sparse terms"), we create a matrix. Here there is a part of it, just to show how data are represented in the dataset we are manipulating. We have the words as columns and the reviews as rows: (i,j) = n if the j-th word appears in the i-th review n times.

	abl	accommod	actual	addit	air	almost	alreadi	also	although	amen	anoth	anyon	anyth	apolog	appear	area	around	arriv	ask
d_hilton_1.txt	0	0	2	0	0	0	0	0	1	1	0	0	0	0	1	1	0	1	0
d_hilton_10.txt	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
d_hilton_11.txt	1	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0
d_hilton_12.txt	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	0
d_hilton_13.txt	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
d_hilton_14.txt	0	1	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
d_hilton_15.txt	0	0	0	0	0	3	1	0	0	0	3	0	2	0	0	1	2	1	0
d_hilton_16.txt	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d_hilton_17.txt	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	2	0
d_hilton_18.txt	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0
d_hilton_19.txt	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	2

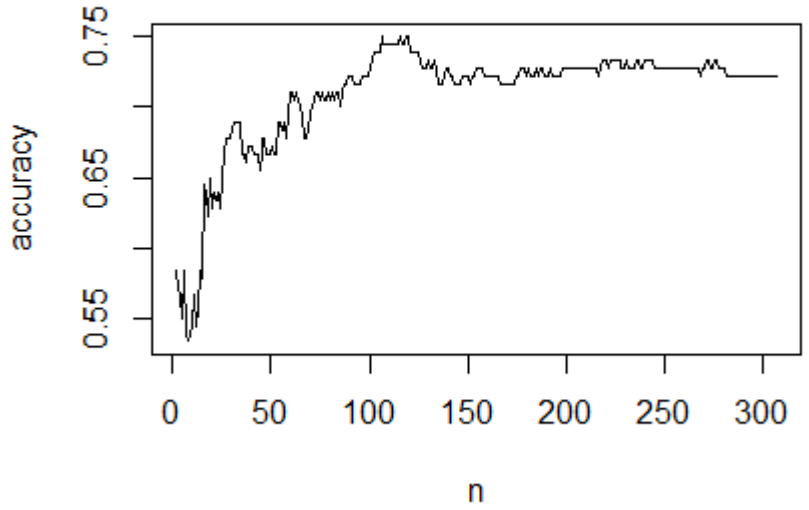
We do the same with bigrams. We have 307 unigrams and 12 bigrams. The bigrams are

```
"called front"     "chicago hotel"    "customer service" "even though"
"front desk"       "got room"         "hard rock"        "hotel chicago"
"never stay"       "room service"     "stay hotel"       "will never"
```

For the test set, we create the same matrix (with the same meaning as well), removing the words which do not appear in the training set.

We train two models:

1. for the first, we compute the mutual information (MI) every feature (unigram) has and order the features according to the MI value. We then build 307 different models all with a different number of features, selecting each time the n best features according to the ordering, n ∈ [2,307]. We select the one with the highest accuracy, corresponding to n = 106. The following graph shows the accuracies depending on n.

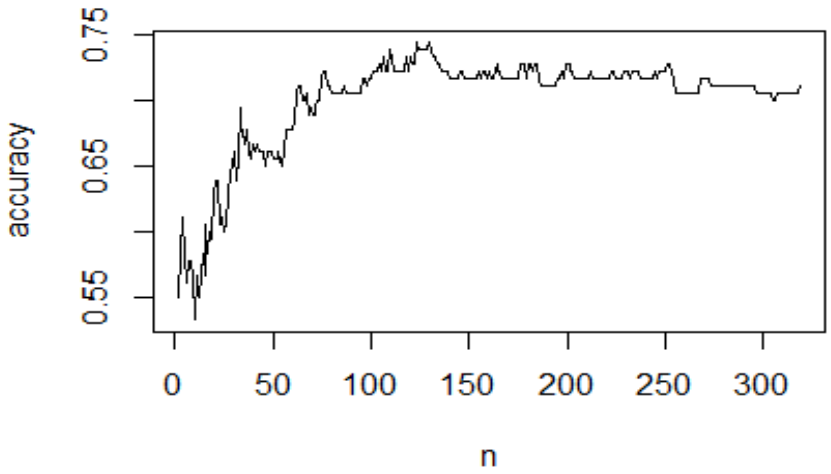


2. For the second, we do the same as before including bigrams. We have then 318 different models, and n ∈ [2,319]. We have that the best n is 123. As an example, here there the best 10 features according to MI, and their values.

```
chicago      location       smell          luxury         hotel chicago
0.09752866   0.03632829     0.03275744     0.03246057     0.03018794

chicago hotel     decided       recently        finally     millennium
   0.02797741     0.02588616    0.02519993     0.02244856     0.02215038
```

The following graph shows again the value of the accuracies depending on n.



Mutual information measures the reduction in uncertainty about variable X achieved by observing the value of variable Y. In our experiment we use it to correlate features with classes (spam – non spam). Filtering out some features according to mutual information can help in avoiding overfitting. Indeed, in both graphs the accuracy increases very fast for n which goes from 0 to 100 (which means we are using too few features, we are underfitting), reaches a maximum between 100 and 150, and then slowly decreases (which means we are overfitting and including some "noise" features).

The results are reported with the following contingency tables

1. Only unigrams

		Review is actually	
		Deceptive	Truthful
Model predicts the review	Deceptive	66	11
	Truthful	14	69

Accuracy	Precision	Recall	F1 score
0,7500	0,8571	0,8250	0,8407

2. Both unigrams and digrams

		Review is actually	
		Deceptive	Truthful
Model predicts the review	Deceptive	64	10
	Truthful	16	70

Accuracy	Precision	Recall	F1 score
0,7444	0,8649	0,8000	0,8312

Including bigrams does not improve the performances of the classifier (and it makes them worse indeed).

To spot the most important features pointing toward a positive or negative review, we use the conditional probabilities of each features found by the models, i.e. the probabilities of observing a specific feature (a unigram or a bigram) given a specific class. The two classes of our analysis are deceptive (class "0"), or truthful (class "1") review. The results are the following (we report only the best 5 features per each category, and their conditional probability)

1. With only unigrams

```
                                    0
room            0.113356890
hotel           0.104734982
chicago         0.052720848
service         0.028268551
like            0.023462898
------------------------------------
                                    1
room            0.1062979540
```

```
hotel        0.0983056266
service      0.0292519182
night        0.0255754476
staff        0.0254156010
```

2. With both unigrams and bigrams

```
                                0
room          0.100463485
hotel         0.092822247
chicago       0.046724289
service       0.025053238
like          0.020794188
------------------------------
                                1
room          0.0942326768
hotel         0.0871475131
service       0.0259316990
night         0.0226725237
staff         0.0225308205
```

We observe that the features with the highest probability are almost the same for the two classes: this could be caused by the fact that these words are the most used in both the reviews, so their probabilities are high anyway, and not depending on the single class.

To get more significant results, we use then the difference between the log conditional probability of class 1 and the log conditional probability of class 0, since this values are a measure of how much the presence of a word can point toward one of the two classes (increasing the difference between its value of conditional probability and the value of the other class) in the document and make us predict then the class with the greatest value. We get the following values (the difference is the third column, a positive value points toward a truthful review, a negative value points toward a deceptive review, we report only the first and the last five)

1. For only unigrams

```
open          0.001272085 0.0068734015        0.732655650
star          0.001837456 0.0083120205        0.655489695
elevator      0.001272085 0.0057544757        0.655489695
comfortable   0.001554770 0.0059143223        0.580238743
conference    0.001696113 0.0063938619        0.576308449
-----------------------------------------------------------
decided       0.008621908 0.0022378517       -0.585772095
recently      0.007632509 0.0017583120       -0.637571371
millennium    0.005371025 0.0006393862       -0.924293901
smell         0.007491166 0.0007992327       -0.971876161
luxury        0.006925795 0.0004795396       -1.159645121
```

2. For both unigrams and bigrams

```
open          0.001127396 0.0060932408        0.732771958
star          0.001628460 0.0073685702        0.655606003
elevator      0.001127396 0.0051013178        0.655606003
comfortable   0.001377928 0.0052430211        0.580355051
conference    0.001503194 0.0056681309        0.576424757
-----------------------------------------------------------------------------

hotel chicago 0.006513842 0.0009919229       -0.817359292
chicago hotel 0.006138043 0.0008502196       -0.858498818
millennium    0.004760115 0.0005668131       -0.924177593
smell         0.006639108 0.0007085164       -0.971759853
luxury        0.006138043 0.0004251098       -1.159528813
```

The code of this experiment can be found in *Appendix2*.

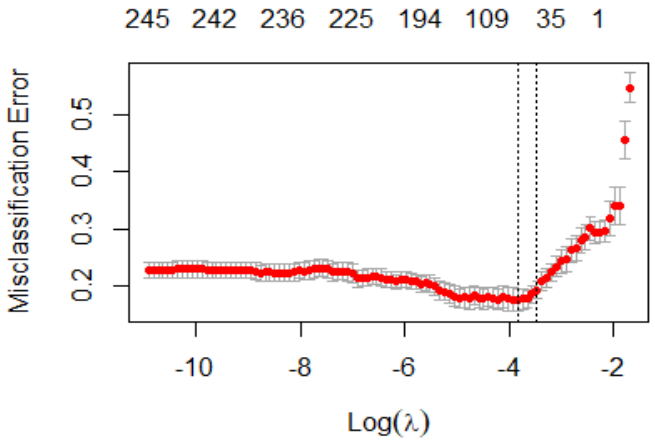
Regularized Logistic Regression

To prepare the data for this experiment we repeat the same procedure done in the previous experiment, for both unigrams and bigrams. We build two models, one with unigrams and the other with both unigrams and bigrams. We put all the code into a function, `logistic.regression`, that we call from the command line with the appropriate parameters.

For each model, the cv.glmnet package performs cross validation on the lambda hyperparameter of the regularization, but not on alpha. The default value of alpha is 1. No other cross validation to tune the hyperparameters is done.

The results are as follows.

1. Only with unigrams
   The best lambda value is 0.02158487. This is the plot of different log lambda values against misclassification error.

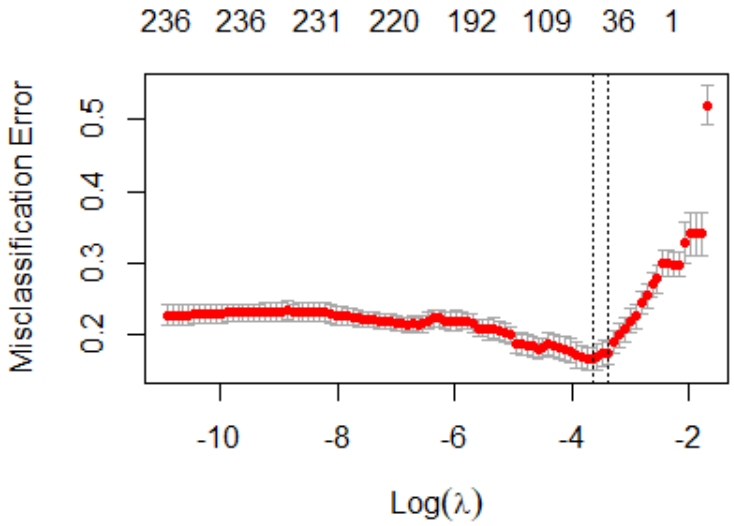


   The following confusion matrix reports the performances.

		Review is actually	
		Deceptive	Truthful
Model predicts	Deceptive	58	8
The review	Truthful	22	72

Accuracy	Precision	Recall	F1 score
0,7222	0,8788	0,7250	0,7945

2. With both unigrams and bigrams
   The best lambda value is 0.02599906. The following graphs are as previously.



		Review is actually	
		Deceptive	Truthful
Model predicts	Deceptive	58	8
The review	Truthful	22	72

Accuracy	Precision	Recall	F1 score
0,7222	0,8788	0,7250	0,7945

The lambda hyperparameter is a measure of how much we want to penalize high weights given to the features to predict the correct class. High weights indeed cause overfitting. The two plots show a common pattern: a high value of lambda produces a very simple model with almost all 0 weights which is unable to work correctly (underfitting). By decreasing lambda, we improve the performance until a maximum. If we do not stop increasing, we get a model with very high weight which is exposed to overfitting, causing the slow worsening of the performance.

The logistic regression classifiers have lower accuracies with respect to the naïve bayes classifiers, so we are induced to state that for this problem the generative linear model is a better choice than the discriminative linear model. However, the precision of the latter is a bit smaller than the precision of the former. A McNemar test could help to find out if the differences are significant. We perform it below here. To do it, we build new vectors out of the old ones of predictions: now, each element of these vectors is 1 if the corresponding predictions was correct, 0 otherwise. We then build a confusion matrix between two vectors and use the R function `mcnemar.test`. The code for the McNemar test (and for all the McNemar tests of this report) can be found in *Appendix6)*

```
McNemar's Chi-squared test with continuity correction

data:  naive.vs.logreg.conf.matrix.unigrams
McNemar's chi-squared = 0.69565, df = 1, p-value = 0.4042
```

Considering only unigrams, we refuse the null ipothesis (the differences are not significant) with an error probability of 40,42%: we can't state at all that the differences are significant.

```
McNemar's Chi-squared test with continuity correction

data:  naive.vs.logreg.conf.matrix
McNemar's chi-squared = 4, df = 1, p-value = 0.0455
```

Considering both unigrams and bigrams, we refuse the null ipothesis (the differences are not significant) with an error probability of 4,55%: there is not strong evidence that the differences are significant (i.e., naïve bayes is a better classifier).

Including bigrams does not improve the performance for this model (which is actually exactly the same). A McNemar test helps us.

```
McNemar's Chi-squared test with continuity correction

data:  logreg.conf.matrix
McNemar's chi-squared = 4, df = 1, p-value = 0.0455
```

We refuse the null ipothesis (the differences are not significant) with an error probability of 4,55%: there is not strong evidence that the differences are significant.

To spot which are the most important variables pointing toward a deceptive (or truthful) review, we look at the coefficient of the built models: the features with the highest weights are the most important pointing toward a truthful review (since they increase the probability of 1, which means "truthful"), while the features with the lowest weights are the most important pointing toward a fake review. These are (we report the first 5 for each category)

1. For the model with only unigrams,
   ```
   elevator          1.019806926
   star              1.066931343
   open              0.766984027
   cant              0.752569003
   rate              0.749664617
   -------------------------
   smelled           -0.755827891
   chicago           -0.807095153
   recently          -0.827175402
   millennium        -1.089214678
   luxury            -1.117586479
   ```

2. For the model with both unigrams and bigrams,
   ```
   elevator          0.825817460
   star              0.741545470
   cant              0.526669861
   open              0.583202710
   rate              0.521997687
   -------------------------
   recently         -0.594929674
   hotel chicago    -0.673143416
   chicago          -0.620774721
   millennium       -0.778313611
   luxury           -0.991769957
   ```

The code of this experiment can be found in *Appendix3*.

Classification Trees

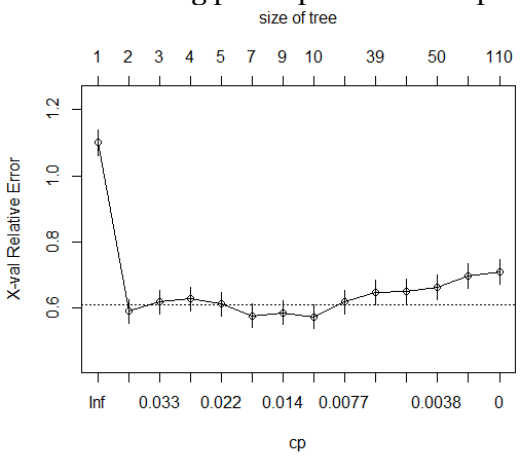
We prepare the training set and the test set as before, and build two trees, one only with unigrams and the other with both unigrams and bigrams, putting all the code in the `classification.tree` function. The procedure to build the tree is the same for both and is reported here.

The `Rpart` package builds a full tree on the training set, using as cost complexity parameter (alpha) the value 0 (`cp = 0`). This parameter is used in the formula to compute the cost of a tree: C(T) = R(T) + alpha * |T|, where R(T) is the resubstitution error of the tree and |T| is the complexity of tree. Setting alpha to 0 means that we develop the tree until all the leaf nodes contain samples of a single class, since in this way we have the entire training sample correctly classified, and the cost is 0. This is not a good condition, since it causes the problem of overfitting over the training data with a very big tree. On the other hand, an high value of alpha leads to a too small tree (underfitting) which performs badly as well. These theoretical considerations are confirmed by our cp plots. Generally, we consider the following pruning procedure: starting from alpha = 0 and increasing it, we gradually reach different values where a part of the tree is pruned since the cost of the tree without it is smaller than the cost of the tree with it. Finally, we have only the root node. The Rpart package, while building the tree with alpha = 0, computes a pruning sequence reporting the values of alpha at which the smallest cost minimizing tree changes (i.e., is pruned in some parts). To evaluate the found values of alpha Rpart performs cross validation and reports it as well. We have then just to pick the value with the small cross validation prediction error (`xerror`) and prune the tree with that value. Here there are the results.

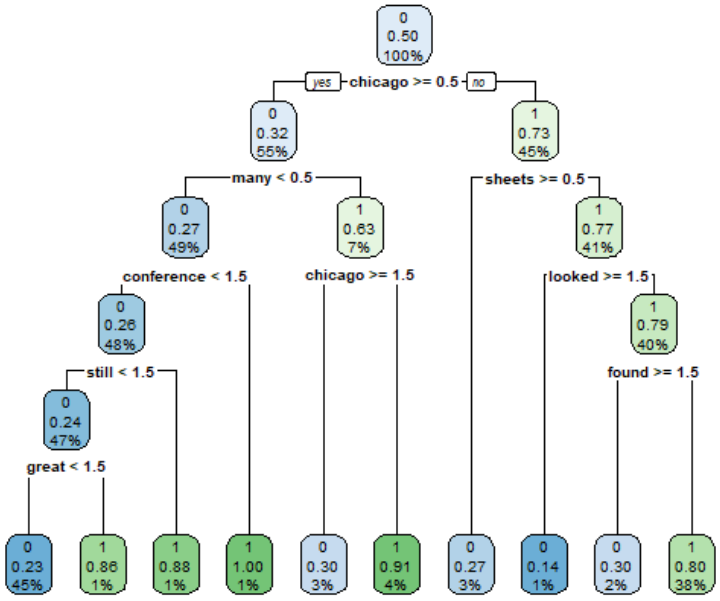
1. Only with unigrams
   Cp values

	CP	nsplit	rel error	xerror	xstd
1	0.4093750	0	1.000000	1.10000	0.039330
2	0.0343750	1	0.590625	0.59063	0.036064
3	0.0312500	2	0.556250	0.61875	0.036543
4	0.0250000	3	0.525000	0.62813	0.036694
5	0.0187500	4	0.500000	0.61250	0.036440
6	0.0156250	6	0.462500	0.57812	0.035839
7	0.0125000	8	0.431250	0.58750	0.036009
8	0.0093750	9	0.418750	0.57500	0.035781
9	0.0062500	15	0.362500	0.61875	0.036543
10	0.0049107	38	0.215625	0.64688	0.036982
11	0.0046875	46	0.175000	0.65000	0.037028
12	0.0031250	49	0.159375	0.66250	0.037209
13	0.0015625	91	0.028125	0.69688	0.037669
14	0.0000000	109	0.000000	0.70938	0.037822

   The best value is `0.009375`.
   The following plot represents the cp value against the cross - validation prediction error.



This is the pruned tree: in the nodes, 0 means deceptive and "1" means truthful. There are indicated also the percentages of reviews per each node out of the total training set of reviews. The values between 0 and 1 indicate the probability of a truthful review at that node: a prob. value below 0.5 makes up predict 0 (i.e., deceptive), above 0.5 makes us predict 1 (i.e., truthful). Values near 0.5 are a sign of great uncertainty.



This is the confusion matrix.

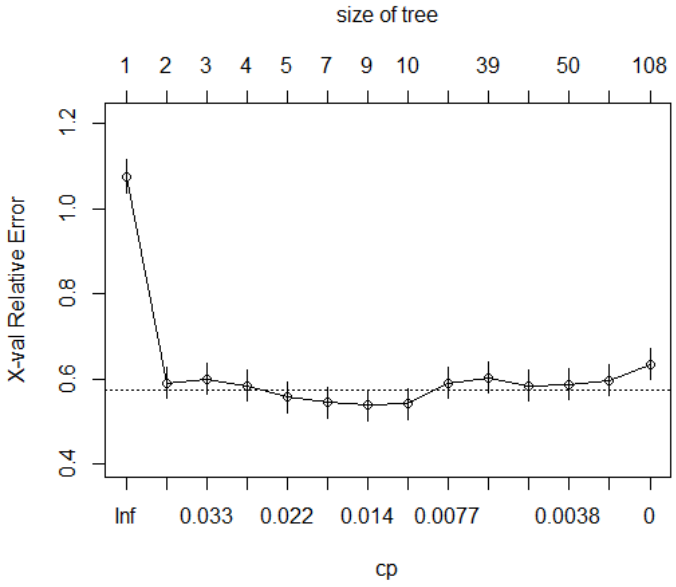
		Review is actually	
		Deceptive	Truthful
Model predicts	Deceptive	45	20
The review	Truthful	35	60

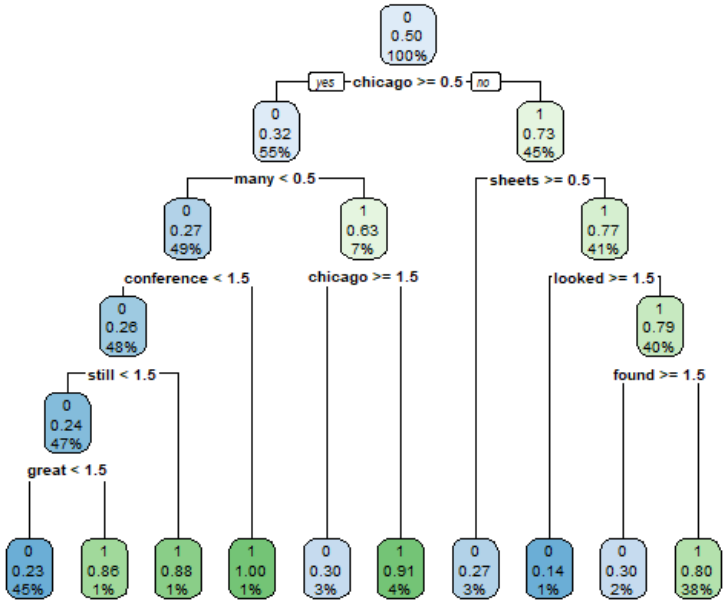
Accuracy	Precision	Recall	F1 score
0,5833	0,6923	0,5625	0,6207

2. With both unigrams and bigrams
   Cp values

	CP	nsplit	rel error	xerror	xstd
1	0.4093750	0	1.000000	1.07500	0.039417
2	0.0343750	1	0.590625	0.59063	0.036064
3	0.0312500	2	0.556250	0.60000	0.036228
4	0.0250000	3	0.525000	0.58437	0.035953
5	0.0187500	4	0.500000	0.55625	0.035423
6	0.0156250	6	0.462500	0.54375	0.035174
7	0.0125000	8	0.431250	0.53750	0.035047
8	0.0093750	9	0.418750	0.54063	0.035111
9	0.0062500	15	0.362500	0.59063	0.036064
10	0.0049107	38	0.215625	0.60313	0.036282
11	0.0046875	46	0.175000	0.58437	0.035953
12	0.0031250	49	0.159375	0.58750	0.036009
13	0.0015625	93	0.021875	0.59688	0.036174
14	0.0000000	107	0.000000	0.63438	0.036792

The best value is `0.0125`. The following table represents the cp value against the error as well, and then there is the pruned tree.





This is the confusion matrix.

		Review is actually	
		Deceptive	Truthful
Model predicts	Deceptive	45	20
The review	Truthful	35	60

Accuracy	Precision	Recall	F1 score
0,5833	0,6923	0,5625	0,6207

The bigrams do not improve the performances, since we have exactly the same tree and then obviously the same performances.

From the tree, we can spot which features are pointing toward a truthful review and which ones re pointing toward a fake review in this model. Feature importance is calculated as the decrease in node impurity weighted by the probability of reaching that node. The node probability can be calculated by the number of samples that reach the node, divided by the total number of samples. The higher the value the more important the feature. We measure impurities with the Gini index and report the feature importance in the following table.

chicago	0.2084
many	0,1129
conference	0,0924
Chicago (>= 1.5)	0,0096
Still	0,0868
great	0,0809
sheets	0,0785
looked	0,0675
found	0,0650

The presence of “Chicago”, “sheets”, 2 “looked”, 2 “found”, are the most important features pointing toward a deceptive review.

The presence of “many”, 2 “conference”, 2 “still”, 2 “great” are the most important features pointing toward a truthful review.

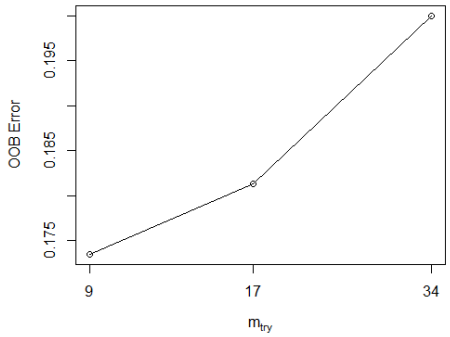
The code of this experiment can be found in *Appendix4*.

Random Forests

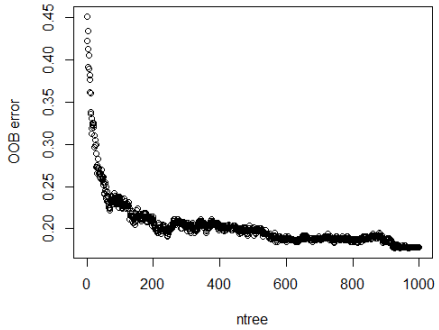
We prepare the training set and test set as before and follow the same procedure to build two forests with optimal hyperparameters, one with only unigrams and the other with both unigrams and bigrams. The hyperparameters to tune are the number of variables randomly sampled as candidates at each split (`mtry`) and the number of trees in the forest (`ntree`). The values of mtry and ntree are a measure of the complexity of the model: depending on the quantity of data, we can afford a model with a certain complexity (in theory, more data allows for more complexity). A more complex model usually performs better, but that’s not true if the sample size does not allow for it, leading to high variance in the model. The procedure is the following (all the code is put inside a `random.forest.function`).

The package `randomForest` contains a function (`TuneRF`) which searches for the best value of mtry and returns a matrix with the values of mtry the algorithm has tried and the corresponding out of bag error estimates (OOB error). We select the value with the smallest OOB error and build a random forest (with the `randomForest` command) with 1000 trees but recording, when adding ethe n-th tree to the forest, the OOB error rate for all trees up to n, with n ∈ [1,1000], with the command `err.rate = TRUE.` We then select the best n and build the optimal tree with the found values of `mtry` and `ntree` (= best n). The results are reported here.

1. With only unigrams
   The algorithm performs the research and founds 9 as optimal mtry.



The algorithm founds 926 as best ntree.

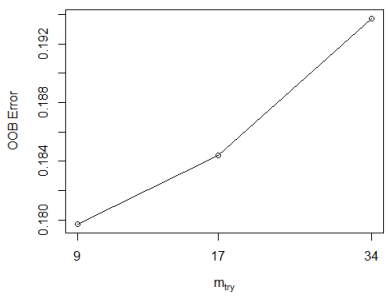


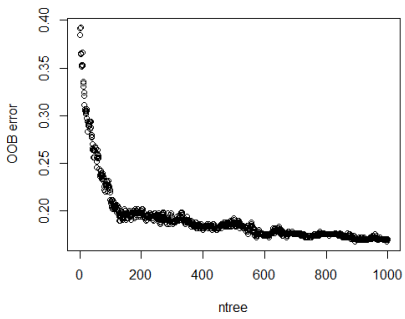
These are the performances.

		Review is actually	
		Deceptive	Truthful
Model predicts	Deceptive	65	19
The review	Truthful	15	61

Accuracy	Precision	Recall	F1 score
0,7000	0,7738	0,8125	0,7927

2. With both unigrams and bigrams
9 is the best mty, 894 is best ntree.





		Review is actually	
		Deceptive	Truthful
Model predicts	Deceptive	62	13
The review	Truthful	18	67

Accuracy	Precision	Recall	F1 score
0,7167	0,8267	0,7750	0,80000

Using also the bigrams improves a bit the performances for the random forests. To be sure of that, we perform a Mc Nemar test.

```
McNemar's Chi-squared test with continuity correction

data:  random.forests.conf.matrix
McNemar's chi-squared = 0.26667, df = 1, p-value = 0.6056
```

We refuse the null ipothesis (the differences are not significant) with an error probability of 60,56%: we can't state at all that the differences are significant, contrary to what previously stated.

The random forest does not improve the performances with respect to the linear classifiers, neither. Again, a Mc Nemar test is performed to support our first impression. We consider only the models with bigrams.

```
McNemar's Chi-squared test with continuity correction

data:  naive.vs.random.conf.matrix
McNemar's chi-squared = 0.59259, df = 1, p-value = 0.4414
```

Comparing Naïve Bayes and Random forests, we refuse the null ipothesis (the differences are not significant) with an error probability of 44,14%: we can't state at all that the differences are significant.

```
        McNemar's Chi-squared test with continuity correction

data:  logreg.vs.random.conf.matrix
McNemar's chi-squared = 1.1364, df = 1, p-value = 0.2864
```

Comparing Logistic regression and Random forests, we refuse the null ipothesis (the differences are not significant) with an error probability of 28,64%: we still can't state that the differences are significant.

To spot the most important features for this model, we could repeat the same procedure of the classification tree to compute feature importance with each tree of the forests and average the results. Unfortunately, we don't do that now and leave it to future work.

The code of this experiment can be found in *Appendix5*.

Conclusions

We couldn't improve the performances of the Naïve Bayes and SVM with linear kernel models of (Ott et al., 2013), since the accuracies of our models are way below the value of 86% reached from

those models. Maybe we could improve our results with the use of more sophisticated feature extraction procedures, like part-of-speech-tagging. We leave this to future work. Moreover, our flexible classifiers do not improve in any way the performances.

Appendix 1: Cleaning function

```
#function used to clean the data
library(tm)

cleaning.function <- function(corpus.dec, corpus.true){
  reviews.dec <-VCorpus(DirSource(corpus.dec,encoding="UTF-8"))
  reviews.true<-VCorpus(DirSource(corpus.true,encoding="UTF-8"))
  review.all<-c(reviews.dec,reviews.true)
  #clean the data
  review.all <- tm_map(review.all, content_transformer(tolower))
  review.all <- tm_map(review.all, removeNumbers)
  review.all <- tm_map(review.all, removePunctuation)
  review.all <- tm_map(review.all, stripWhitespace)
  review.all <- tm_map(review.all,removeWords,stopwords("english"))
  return (review.all)

}
```

Appendix 2: Code for Naïve Bayes analysis

```
#-------------------------libraries--------------------#
library(entropy)

naive.bayes.function <- function (training.corpus.dec,
training.corpus.true, testing.corpus.dec, testing.corpus.true){
  #training set
  training.dtm <-
cleaning.function(training.corpus.dec,training.corpus.true)

  #extraction of unigrams
  training.dtm.unigrams <- DocumentTermMatrix(training.dtm)
  training.dtm.unigrams <-
removeSparseTerms(training.dtm.unigrams,0.95)
  training.dtm.unigrams <- as.matrix(training.dtm.unigrams)

  #extraction of bigrams
  BigramTokenizer <-function(x) unlist(lapply(ngrams(words(x), 2),
paste, collapse = " "), use.names = FALSE)
  training.dtm.bigrams <- DocumentTermMatrix(training.dtm,control =
list(tokenize = BigramTokenizer))
  training.dtm.bigrams <-
removeSparseTerms(training.dtm.bigrams,0.95)
  training.dtm.bigrams <- as.matrix(training.dtm.bigrams)

  #training set with both unigrams and bigrams
  training.labels <- c(rep(0,320),rep(1,320))
  training.dtm <- cbind(training.dtm.unigrams,
training.dtm.bigrams)


##################################################################
################
  #test set
  test.dtm <- cleaning.function
(testing.corpus.dec,testing.corpus.true)
```

```
  #unigrams
  test.dtm.unigrams<-
DocumentTermMatrix(test.dtm,list(dictionary=dimnames(training.dtm.u
nigrams)[[2]]))
  test.dtm.unigrams <- as.matrix(test.dtm.unigrams)

  #bigrams
  test.dtm.bigrams<-
DocumentTermMatrix(test.dtm,list(dictionary=dimnames(training.dtm.b
igrams)[[2]]))
  test.dtm.bigrams <- as.matrix(test.dtm.bigrams)

  #test set with both unigrams and bigrams
  test.labels <- c(rep(0,80),rep(1,80))
  test.dtm <- cbind(test.dtm.unigrams, test.dtm.bigrams)

##################################################################
####################
  #feature selection (with mutual information, only for unigrams)
  training.dtm.unigrams.mi <-
apply(training.dtm.unigrams,2,function(x,y){
    mi.plugin(table(x,y)/length(y))},training.labels)
  training.dtm.unigrams.mi.order <-
order(training.dtm.unigrams.mi,decreasing = T)

  #feature selection ( with mutual information)
  training.dtm.mi <- apply(training.dtm,2,function(x,y){

mi.plugin(table(x,y)/length(y))},training.labels)
  training.dtm.mi.order <- order(training.dtm.mi,decreasing = T)

##################################################################
########################
  #print
  print(dim(training.dtm.bigrams))
  print(dim(training.dtm.unigrams))
  print(colnames(training.dtm.bigrams))
  print(dim(training.dtm)) ## just to check
  print(training.dtm.mi[training.dtm.mi.order[1:10]])

  #first model ( with feature selection according to mutual
information)(only unigrams)
  accuracies.unigrams.mi.models <- sapply(c(2:307), function
(num.features){
    model.mi <-
train.mnb(training.dtm.unigrams[,training.dtm.unigrams.mi.order[1:n
um.features] ], training.labels)
    predictions.mi <- predict.mnb(model.mi ,
test.dtm.unigrams[,training.dtm.unigrams.mi.order[1:num.features]])
    conf.mat <- table (predictions.mi ,test.labels)
    return (sum(diag(conf.mat))/180)
  } )

  accuracies.unigrams.mat <- cbind (accuracies.unigrams.mi.models,
c(2:307))
  accuracies.unigrams.best.n <-
accuracies.unigrams.mat[which.max(accuracies.unigrams.mat[,1]), 2]
  print(accuracies.unigrams.mat)
  print(accuracies.unigrams.best.n)
```

```
  plot(accuracies.unigrams.mat[,2] ,accuracies.unigrams.mat[,1],
xlab = "n", ylab = "accuracy", type = "l")
  model.unigrams.mi <-
train.mnb(training.dtm.unigrams[,training.dtm.unigrams.mi.order[1:a
ccuracies.unigrams.best.n] ], training.labels)
  predictions.unigrams.mi <- predict.mnb(model.unigrams.mi ,
test.dtm.unigrams[,training.dtm.unigrams.mi.order[1:accuracies.unig
rams.best.n]])
  print(table (predictions.unigrams.mi ,test.labels))


  #second model ( with feature selection according to mutual
information)(both unigrams and bigrams)
  accuracies.mi.models <- sapply(c(2:319), function (num.features){
    model.mi <-
train.mnb(training.dtm[,training.dtm.mi.order[1:num.features] ],
training.labels)
    predictions.mi <- predict.mnb(model.mi ,
test.dtm[,training.dtm.mi.order[1:num.features]])
    conf.mat <- table (predictions.mi ,test.labels)
    return (sum(diag(conf.mat))/180)
  } )

  accuracies.mat <- cbind (accuracies.mi.models, c(2:319))
  accuracies.best <- accuracies.mat[which.max(accuracies.mat[,1]),
2]
  print(accuracies.mat)
  print(accuracies.best)
  plot(accuracies.mat[,2] ,accuracies.mat[,1], xlab = "n", ylab =
"accuracy", type = "l")
  model.mi <-
train.mnb(training.dtm[,training.dtm.mi.order[1:accuracies.best] ],
training.labels)
  predictions.mi <- predict.mnb(model.mi ,
test.dtm[,training.dtm.mi.order[1:accuracies.best]])
  print(table (predictions.mi ,test.labels))

cond.probabilities <- model.mi$cond.probs
difference <- log(cond.probabilities[,2], 10) -
log(cond.probabilities[,1],10)
cond.probabilities <- cbind(cond.probabilities, difference)
print(cond.probabilities[order(-cond.probabilities[,3]), ])

cond.probabilities.unigrams <- model.unigrams.mi$cond.probs
difference.unigrams <- log(cond.probabilities.unigrams[,2], 10) -
log(cond.probabilities.unigrams[,1],10)
cond.probabilities.unigrams <- cbind(cond.probabilities.unigrams,
difference)
print(cond.probabilities.unigrams[order(-
cond.probabilities.unigrams[,3]), ])



}

#Training function for Naive Bayes
#labels = classes
train.mnb <- function (dtm,labels) {
  call <- match.call()
  V <- ncol(dtm) #vocabulary
```

```
  N <- nrow(dtm) #number of documents
  prior <- table(labels)/N
  labelnames <- names(prior)
  nclass <- length(prior)
  cond.probs <- matrix(nrow=V,ncol=nclass)
  dimnames(cond.probs)[[1]] <- dimnames(dtm)[[2]]
  dimnames(cond.probs)[[2]] <- labelnames
  index <- list(length=nclass)
  for(j in 1:nclass){
    index[[j]] <- c(1:N)[labels == labelnames[j]]
  }
  for(i in 1:V){
    for(j in 1:nclass){
      cond.probs[i,j] <-
(sum(dtm[index[[j]],i])+1)/(sum(dtm[index[[j]],])+V)
    #Laplace smoothing
    }
  }
  x <- list(call=call,prior=prior,cond.probs=cond.probs)
  return (x)
}


predict.mnb <- function (model,dtm) {
  classlabels <- dimnames(model$cond.probs)[[2]]
  logprobs <- dtm %*% log(as.matrix(model$cond.probs))
  N <- nrow(dtm) #number of documents to classify
  nclass <- ncol(model$cond.probs) #number of classes
  logprobs <-
logprobs+matrix(nrow=N,ncol=nclass,log(model$prior),byrow=T)
  x <- classlabels[max.col(logprobs)]
  return (x)
}
```

Appendix 3: Code for regularized logistic regression analysis

```
#-------------------------libraries--------------------#
library("glmnet")

logistic.function <- function (training.corpus.dec,
training.corpus.true, testing.corpus.dec, testing.corpus.true){
  #training set
  training.dtm <-
cleaning.function(training.corpus.dec,training.corpus.true)

  #extraction of unigrams
  training.dtm.unigrams <- DocumentTermMatrix(training.dtm)
  training.dtm.unigrams <-
removeSparseTerms(training.dtm.unigrams,0.95)
  training.dtm.unigrams <- as.matrix(training.dtm.unigrams)

  #extraction of bigrams
  BigramTokenizer <-function(x) unlist(lapply(ngrams(words(x), 2),
paste, collapse = " "), use.names = FALSE)
  training.dtm.bigrams <- DocumentTermMatrix(training.dtm,control =
list(tokenize = BigramTokenizer))
  training.dtm.bigrams <-
removeSparseTerms(training.dtm.bigrams,0.95)
  training.dtm.bigrams <- as.matrix(training.dtm.bigrams)

  #training set with both unigrams and bigrams
```

```
  training.labels <- c(rep(0,320),rep(1,320))
  training.dtm <- cbind(training.dtm.unigrams,
training.dtm.bigrams)

  #test set
  test.dtm <- cleaning.function
(testing.corpus.dec,testing.corpus.true)

  #unigrams
  test.dtm.unigrams<-
DocumentTermMatrix(test.dtm,list(dictionary=dimnames(training.dtm.u
nigrams)[[2]]))
  test.dtm.unigrams <- as.matrix(test.dtm.unigrams)

  #bigrams
  test.dtm.bigrams<-
DocumentTermMatrix(test.dtm,list(dictionary=dimnames(training.dtm.b
igrams)[[2]]))
  test.dtm.bigrams <- as.matrix(test.dtm.bigrams)

  #test set with both unigrams and bigrams
  test.labels <- c(rep(0,80),rep(1,80))
  test.dtm <- cbind(test.dtm.unigrams, test.dtm.bigrams)

  #first model (only unigrams)
  reviews.glmnet.unigrams <-
cv.glmnet(training.dtm.unigrams,training.labels,
family="binomial",type.measure="class")
  print(coef(reviews.glmnet.unigrams,s="lambda.min"))
  print(reviews.glmnet.unigrams$lambda.min)
  plot (reviews.glmnet.unigrams)
  reviews.logreg.pred.unigrams <- predict(reviews.glmnet.unigrams,

newx=test.dtm.unigrams,s="lambda.min",type="class")
  print(table(reviews.logreg.pred.unigrams,test.labels))

  #second model (with bigrams)
  reviews.glmnet <- cv.glmnet(training.dtm,training.labels,
family="binomial",type.measure="class")
  print(coef(reviews.glmnet,s="lambda.min"))
  print(reviews.glmnet$lambda.min)
  plot(reviews.glmnet)
  reviews.logreg.pred <- predict(reviews.glmnet,

newx=test.dtm,s="lambda.min",type="class")
  print(table(reviews.logreg.pred.unigrams,test.labels))
}
```

Appendix 4: Code for classification tree

```
library(tm)
library(entropy)
library(SnowballC)
library(rpart)
library(rpart.plot)
classification.tree <- function (training.corpus.dec,
training.corpus.true, testing.corpus.dec, testing.corpus.true){
  #training set
  training.dtm <-
cleaning.function(training.corpus.dec,training.corpus.true)
```

```
  #extraction of unigrams
  training.dtm.unigrams <- DocumentTermMatrix(training.dtm)
  training.dtm.unigrams <-
removeSparseTerms(training.dtm.unigrams,0.95)
  training.dtm.unigrams <- as.matrix(training.dtm.unigrams)

  #extraction of bigrams
  BigramTokenizer <-function(x) unlist(lapply(ngrams(words(x), 2),
paste, collapse = " "), use.names = FALSE)
  training.dtm.bigrams <- DocumentTermMatrix(training.dtm,control =
list(tokenize = BigramTokenizer))
  training.dtm.bigrams <-
removeSparseTerms(training.dtm.bigrams,0.95)
  training.dtm.bigrams <- as.matrix(training.dtm.bigrams)

  #training set with both unigrams and bigrams
  training.labels <- c(rep(0,320),rep(1,320))
  training.dtm <- cbind(training.dtm.unigrams,
training.dtm.bigrams)


################################################################
################
  #test set
  test.dtm <- cleaning.function
(testing.corpus.dec,testing.corpus.true)

  #unigrams
  test.dtm.unigrams<-
DocumentTermMatrix(test.dtm,list(dictionary=dimnames(training.dtm.u
nigrams)[[2]]))
  test.dtm.unigrams <- as.matrix(test.dtm.unigrams)

  #bigrams
  test.dtm.bigrams<-
DocumentTermMatrix(test.dtm,list(dictionary=dimnames(training.dtm.b
igrams)[[2]]))
  test.dtm.bigrams <- as.matrix(test.dtm.bigrams)

  #test set with both unigrams and bigrams
  test.labels <- c(rep(0,80),rep(1,80))
  test.dtm <- cbind(test.dtm.unigrams, test.dtm.bigrams)
  ########################################################
  # grow the tree with only unigrams

  reviews.rpart.unigrams <- rpart(label~.,

data=data.frame(training.dtm.unigrams,label = training.labels),
                           cp=0,method="class", minbucket = 1,
minsplit = 2 )

  # tree with lowest cv error
  opt.cp.unigrams <-
reviews.rpart.unigrams$cptable[which.min(reviews.rpart.unigrams$cpt
able[,"xerror"]),"CP"]
  print(opt.cp.unigrams)
  plotcp(reviews.rpart.unigrams)
  printcp(reviews.rpart.unigrams)
  reviews.rpart.unigrams.pruned <- prune(reviews.rpart.unigrams,cp
=
```

```
reviews.rpart.unigrams$cptable[which.min(reviews.rpart.unigrams$cpt
able[,"xerror"]),"CP"] )
  rpart.plot(reviews.rpart.unigrams.pruned, roundint = FALSE)
  # make predictions on the test set
  reviews.rpart.unigrams.pred <-
predict(reviews.rpart.unigrams.pruned,

newdata=data.frame(as.matrix(test.dtm.unigrams)),type="class")
  # show confusion matrix
  print(table(reviews.rpart.unigrams.pred,test.labels))

########################################################################
####################
  #unigrams and bigrams
  reviews.rpart <- rpart(label~.,
                         data=data.frame(training.dtm,label =
training.labels),
                           cp=0,method="class", minbucket = 1, minsplit
= 2)
  # tree with lowest cv error
  opt.cp <-
reviews.rpart$cptable[which.min(reviews.rpart$cptable[,"xerror"]),"
CP"]
  print(opt.cp)
  plotcp(reviews.rpart)
  printcp(reviews.rpart)
  reviews.rpart.pruned <- prune(reviews.rpart,cp =
reviews.rpart$cptable[which.min(reviews.rpart$cptable[,"xerror"]),"
CP"] )
  rpart.plot(reviews.rpart.unigrams.pruned, roundint = FALSE)
  # make predictions on the test set
  reviews.rpart.pred <- predict(reviews.rpart.pruned,

newdata=data.frame(as.matrix(test.dtm)),type="class")
  # show confusion matrix
  print(table(reviews.rpart.pred,test.labels))
}
```

Appendix 5: Code for random forests

```
library(randomForest)
random.forest.function <- function (training.corpus.dec,
training.corpus.true, testing.corpus.dec, testing.corpus.true){
  #training set
  training.dtm <-
cleaning.function(training.corpus.dec,training.corpus.true)

  #extraction of unigrams
  training.dtm.unigrams <- DocumentTermMatrix(training.dtm)
  training.dtm.unigrams <-
removeSparseTerms(training.dtm.unigrams,0.95)
  training.dtm.unigrams <- as.matrix(training.dtm.unigrams)

  #extraction of bigrams
  BigramTokenizer <-function(x) unlist(lapply(ngrams(words(x), 2),
paste, collapse = " "), use.names = FALSE)
  training.dtm.bigrams <- DocumentTermMatrix(training.dtm,control =
list(tokenize = BigramTokenizer))
  training.dtm.bigrams <-
removeSparseTerms(training.dtm.bigrams,0.95)
```

```
  training.dtm.bigrams <- as.matrix(training.dtm.bigrams)

  #training set with both unigrams and bigrams
  training.labels <- c(rep(0,320),rep(1,320))
  training.labels <- as.factor(training.labels)
  training.dtm <- cbind(training.dtm.unigrams,
training.dtm.bigrams)


#####################################################################
################
  #test set
  test.dtm <- cleaning.function
(testing.corpus.dec,testing.corpus.true)

  #unigrams
  test.dtm.unigrams<-
DocumentTermMatrix(test.dtm,list(dictionary=dimnames(training.dtm.u
nigrams)[[2]]))
  test.dtm.unigrams <- as.matrix(test.dtm.unigrams)

  #bigrams
  test.dtm.bigrams<-
DocumentTermMatrix(test.dtm,list(dictionary=dimnames(training.dtm.b
igrams)[[2]]))
  test.dtm.bigrams <- as.matrix(test.dtm.bigrams)
  #test set with both unigrams and bigrams
  test.labels <- c(rep(0,80),rep(1,80))
  test.labels <- as.factor(test.labels)
  test.dtm <- cbind(test.dtm.unigrams, test.dtm.bigrams)
  ########################################################
  #with only unigrams
  training.dtm.unigrams <- as.data.frame(training.dtm.unigrams)
  test.dtm.unigrams <- as.data.frame(test.dtm.unigrams)
  OOB.matrix.unigrams <- tuneRF(x =training.dtm.unigrams,
                       y = training.labels,
                       ntreeTry = 500, doBest = FALSE, plot = TRUE)
  print(OOB.matrix.unigrams)
  optimal.mtry.unigrams <-
OOB.matrix.unigrams[which.min(OOB.matrix.unigrams[,2]),1]
  print(optimal.mtry.unigrams)
  classifier.unigrams <- randomForest(x = training.dtm.unigrams,
                              y = training.labels, ntree = 1000,
                              mtry = optimal.mtry.unigrams, type =
"classification", err.rate = TRUE)
  err.rate.unigrams <- cbind(
c(1:1000),classifier.unigrams$err.rate[,1])
  optimal.ntree.unigrams <-
err.rate.unigrams[which.min(err.rate.unigrams[,2]), 1]
  print(optimal.ntree.unigrams)
  colnames(err.rate.unigrams) <- c( "ntree","OOB error")
  plot(err.rate.unigrams)
  classifier.unigrams <- randomForest(x = training.dtm.unigrams,
                              y = training.labels, mtry =
optimal.mtry.unigrams,
                              ntree = optimal.ntree.unigrams, type =
"classification")
  # Predicting the Test set results
  predictions.unigrams <- predict(classifier.unigrams, newdata =
test.dtm.unigrams)
```

```
  print(table(predictions.unigrams,test.labels))
  ########################################################
  #with both unigrams and bigrams
  training.dtm <- as.data.frame(training.dtm)
  test.dtm <- as.data.frame(test.dtm)
  OOB.matrix <- tuneRF(x =training.dtm,
                               y = training.labels,
                               ntreeTry = 500, doBest = FALSE,
plot = TRUE)
  print(OOB.matrix)
  optimal.mtry <- OOB.matrix[which.min(OOB.matrix[,2]),1]
  print(optimal.mtry)
  classifier <- randomForest(x = training.dtm,
                                    y = training.labels, ntree =
1000,
                                    mtry = optimal.mtry, type =
"classification", err.rate = TRUE)

  err.rate <- cbind(c(1:1000),classifier$err.rate[,1] )
  optimal.ntree <- err.rate[which.min(err.rate[,2]), 1]
  print(optimal.ntree)
  colnames(err.rate) <- c( "ntree","OOB error")
  plot(err.rate)
  classifier <- randomForest(x = training.dtm,
                                    y = training.labels, mtry =
optimal.mtry,
                                    ntree = optimal.ntree, type =
"classification")
  # Predicting the Test set results
  predictions <- predict(classifier, newdata = test.dtm)
  print(table(predictions,test.labels))

}
```

Appendix 6: Code for Mc Nemar tests

```
reviews.logreg.pred.is.correct <-
sapply(c(1:length(reviews.logreg.pred)),function(index){
  if(reviews.logreg.pred[index] == test.labels[index]){
    return (1)
  }
  else (return (0))
})

reviews.logreg.pred.unigrams.is.correct <-
sapply(c(1:length(reviews.logreg.pred.unigrams)),function(index){
  if(reviews.logreg.pred.unigrams[index] == test.labels[index]){
    return (1)
  }
  else (return (0))
})

naive.bayes.predictions.mi.is.correct <-
sapply(c(1:length(naive.bayes.predictions.mi)),function(index){
  if(naive.bayes.predictions.mi[index] == test.labels[index]){
    return (1)
  }
  else (return (0))
})
```

```
naive.bayes.predictions.unigrams.mi.is.correct <-
sapply(c(1:length(naive.bayes.predictions.unigrams.mi)),function(in
dex){
  if(naive.bayes.predictions.unigrams.mi[index] ==
test.labels[index]){
    return (1)
  }
  else (return (0))
})

random.forests.predictions.is.correct <-
sapply(c(1:length(random.forests.predictions)),function(index){
  if(random.forests.predictions[index] == test.labels[index]){
    return (1)
  }
  else (return (0))
})

random.forests.predictions.unigrams.is.correct <-
sapply(c(1:length(random.forests.predictions.unigrams)),function(in
dex){
  if(random.forests.predictions.unigrams[index] ==
test.labels[index]){
    return (1)
  }
  else (return (0))
})

reviews.rpart.pred.is.correct <-
sapply(c(1:length(reviews.rpart.pred)),function(index){
  if(reviews.rpart.pred[index] == test.labels[index]){
    return (1)
  }
  else (return (0))
})

reviews.rpart.unigrams.pred.is.correct <-
sapply(c(1:length(reviews.rpart.unigrams.pred)),function(index){
  if(reviews.rpart.unigrams.pred[index] == test.labels[index]){
    return (1)
  }
  else (return (0))
})


naive.vs.logreg.conf.matrix <-
table(naive.bayes.predictions.mi.is.correct,
reviews.logreg.pred.is.correct)
mcnemar.test(naive.vs.logreg.conf.matrix)

naive.vs.logreg.conf.matrix.unigrams <-
table(naive.bayes.predictions.unigrams.mi.is.correct,
reviews.logreg.pred.unigrams.is.correct)
mcnemar.test(naive.vs.logreg.conf.matrix.unigrams)

random.forests.conf.matrix <-
table(random.forests.predictions.unigrams.is.correct,
random.forests.predictions.is.correct)
mcnemar.test(random.forests.conf.matrix )
```

```
naive.vs.random.conf.matrix <-
table(naive.bayes.predictions.mi.is.correct,
random.forests.predictions.is.correct)
mcnemar.test(naive.vs.random.conf.matrix)

logreg.vs.random.conf.matrix <-
table(reviews.logreg.pred.is.correct,
random.forests.predictions.is.correct)
mcnemar.test(logreg.vs.random.conf.matrix)
```

References

[1] Myle Ott, Yejin Choi, Claire Cardie and Jerey T. Hancock, Finding deceptive opinion spam by any stretch of the imagination. Proceedings of the 49th meeting of the association for computational linguistics, pp. 309-319,2011.

[2] Myle Ott, Claire Cardie and Jerey T. Hancock, Negative deceptive opinion spam. Proceedings of NAACL-HLT 2013, pp. 497-501, 2013.

[3] Cone. 2011. 2011 Online Influence Trend Tracker.