# INFO371 Problem Set 4: logistic regression, SVM, classification

Your name:

Deadline: Tue, Mar 5 midnight

## Introduction

This is the last problem set of this course ☺And you will again work with rotten tomatoes. The good news are that you already know this dataset somewhat. However, now we will explore the data a little bit more, and thereafter you will implement your own brand new shiny Naive Bayes to categorize the quotes into rotten/fresh, and find the optimal smoothing parameters with your own brand-even-newer and even-flashier k-fold cross validation. We also implement the three-fold data split with test data set aside for the final performance measure only.

Please submit a) your code (notebooks, rmd, whatever) *and* b) the results in a final output form (html or pdf). You are free to choose either R or python for solving this problem set.

You are welcome to answer some of the questions on paper but please include the result as an image in your final file. Note that you can easily include images in both notebooks and .rmd—besides of the code, both are just markdown documents.

Working together is fun and useful but you have to submit your own work. Discussing the solutions and problems with your classmates is all right but do not copy-paste their solution! Please list all your collaborators below:

1.
2. ...

## Rotten Tomatoes

Our first task is to load, clean and explore the Rotten Tomatoes movie reviews data. Please familiarize yourself a little bit with the webpage. Briefly, approved critics can write reviews for movies, and evaluate the movie as "fresh" or "rotten". The webpage normally shows a short "quote" from each critic, and whether it was evaluated as fresh or rotten. You will work on these quotes below.

The central variables in *rotten-tomatoes.csv* are the following:

**critic** name of the critic

**fresh** evaluation: 'fresh' or 'rotten'

**quote** short version of the review

**review_date** when the review was written.

There are more variables like links to IMDB.

## 1 Load data

Load data and split it into working and testing chunks. But before you begin: ensure you can save a dataframe in a format you can load back in afterwards. `pd.to_csv` is a good bet, but it has a lot of options which may screw up the way you read data. Ensure you can store data in a way that you can read it back in correctly, including that missings remain missings.

1. create a tiny toy data frame that includes some numbers, strings, and missings. Save it and ensure you can reload it in the correct form.

Now you are good to go:

2. load the data (available on canvas: files/data/rotten-tomatoes.csv). **DO NOT LOOK AT IT**!
3. split the dataset into working-testing parts (80/20 or so). Note that *sklearn*'s `train_test_split` can easily handle dataframes. Just for your confirmation, ensure that the size of the working and testing data look reasonable.
4. now save the test data and *delete it from memory*. Use python's `del` statement, or R-s `rm` function.

## 2 Explore and clean the data

Now when the test data is put aside, we can breath out and take a closer look how does the work data look like.

1. Take a look at a few lines of data (you may use `pd.sample` for this).
2. print out all variable names.
3. create a summary table (maybe more like a bullet list) where you print out the most important summary statistics for the most interesting variables. The most interesting facts you should present should include: a) number of missings for *fresh* and *quote*; b) all different values for fresh/rotten evaluations; c) counts or percentages of these values; d) number of zero-length or only whitespace *quote*-s; e) minimum-maximum-average length of quotes (either in words, or in characters). (Can you do this as an one-liner?); f) how many reviews are in data multiple times. Feel free to add more figures you consider relevant.
4. Now when you have an overview what you have in data, clean it by removing all the inconsistencies the table reveals. We have to ensure that the central variables: *quote* and *fresh* are not missing, and *quote* is not an empty string (or just contain spaces and such).

   I strongly recommend to do it as a standalone function because at the end you have to perform exactly the same cleaning operations with your test data too.

## 3 Naïve Bayes

Now where you are familiar with the data, it's time to get serious and implement the Naive Bayes classifier from scratch. But first things first.

1. Ensure you are familiar with Naive Bayes. Consult the readings, available on canvas. Schutt & O'Neill is an easy and accessible (and long) introduction, Whitten & Frank is a lot shorter but still accessible introduction.

2. Convert your data (quotes) into bag-of-words. Your code should look something along the lines as in PS4:

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
# define vecotrizer
X = vectorizer.fit_transform(cwork.quote.values)
# vectorize your data.  Note: this creates a sparce matrix,
# use .toarray() if you want a dense matrix.
words = vectorizer.get_feature_names()
# in case you want to see what are the actual words
```

   However, now we don't want BOW that contains counts of words in quotes, but just 1/0 (or true/-false) for the presence/non-presence of the words. Convert the count-based BOW into such a presence BOW. Hint: think in terms of vectorized (universal) functions.

3. Split your work data and target (i.e. the variable *fresh*) into training and validation chunks (80/20 or so). Later we also do cross-validation, but for now, a simple training/validation will do.

Good. Now you are ready with the preparatory work and it's time to dive into the real thing. Let's implement Naive Bayes. Use only training data in the fitting below.

4. Compute the unconditional (log) probability that the tomato is fresh/rotten, $\log \Pr(F)$, and $\log \Pr(R)$. These probabilities are based on the values of *fresh* variable but not on the words the quotes contain.

5. For each word $w$, compute $\log \Pr(w|F)$ and $\log \Pr(w|R)$, the (log) probability that the word is present in a fresh/rotten review. These probabilities can easily be calculated from counts of how many times these words are present for each class.

   Hint: these computations are based on your BOW-s $X$. Look at ways to sum along columns in this matrix.

Now we are done with the estimator. Your fitted model is completely described by these four probability vectors: $\log \Pr(F), \log \Pr(R), \log \Pr(w|F), \log \Pr(w|R)$. Let's now turn to prediction, and pull out your validation data (not the test data!).

6. For both destination classes, $F$ and $R$, compute the log-likelihood that the quote belongs to this class. I think we mentioned *log-likelihood* in the class, it is what is given inside the brackets in equation (1) on slide 28, and the equations on Schutt "Doing Data Science", page 102. On the slides we have the log-likelihood essentially as (although we do not write it out):

$$\ell_i(c) = \log \Pr(c) + \sum_j \log \Pr(w_{ij}|c)$$

   where $c \in \{F, R\}$ is the class, $i$ is the review, $j$ indexes words, and $w_{ij}$ is the $j$-th word of the review $i$.

   Computing these likelihoods involves sums of the previously computed probabilities, $\log \Pr(w|F)$, and BOW elements $x_{ij}$. Check out `np.apply_along_axis` (or R's `apply`) that can be used to apply a function on matrix columns/rows so you can create a fairly good one-liner to compute log-likelihood. Loops are fine too, just slower and less compact.

   Based on the log-likelihoods, predict the class $F$ or $R$ for each quote in the validation set.

7. Print the resulting confusion matrix and accuracy (feel free to use existing libraries).

## 4 Interpretation

Now it is time to look at your fitted model a little bit closer. NB model probabilities are rather easy to understand and interpret. The task here is to find the best words to predict a fresh, and a rotten review. And we only want to look at words that are reasonably frequent, say more frequent than 30 times in the data.

1. Extract from your conditional probability vectors $\log \Pr(\mathsf{F})$ and $\log \Pr(\mathsf{R})$ the probabilities that correspond to frequent words only.
2. Find 10 best words to predict $\mathsf{F}$ and 10 best words to predict $\mathsf{R}$. Hint: imagine we have a review that contains just a single word. Which word will give the highest weight to the probability the review is fresh? Which one to the likelihood it is rotten?

   Comment your results.
3. Print out a few missclassified quotes. Can you understand why these are misclassified?

## 5 NB with smoothing

So, now you have your brand-new NB algorithm up and running. As a next step, we add smoothing to it. As you will be doing cross-validation below, your first task is to mold what you did above into two funcions: one for fitting and another one for predicting.

1. Create two functions: one for fitting NB model, and another to predict outcome based on the fitted model.

   As mentioned above, the model is fully described with 4 probabilities, so your fitting function may return such a list as the model; and the prediction function may take it as an input.
2. Add smoothing to the model. See Schutt p 103 and 109. Smoothing amounts to assuming that we have "seen" every possible work $\alpha \geqslant 0$ times already, for both classes. (If you wish, you can also assume you have seen the words $\alpha$ times for $\mathsf{F}$ and $\beta$ times for $\mathsf{R}$). Note that $\alpha$ does not have to be an integer, and typically the best $\alpha < 1$.
3. Now fit a few models with different $\alpha$-s and see if the accuracy improves compared to the baseline case above.

## 6 Cross-Validation

Finally (well, almost finally), we do cross-validation. This is another piece of code *you have to implement yourself*, not use existing libraries.

- Implement $k$-fold CV. I recommend to implement it as a function that a) puts your data into random order; b) splits these into $k$ chunks; c) selects a chunk for testing and the others for training; d) trains your NB model on the training chunks; e) computes accuracy on training chunk; f) returns mean accuracy over all these $k$ trials. The function should also take $\alpha$ as an argument, this is the hyperparameter you are going to optimize.
- Find the optimal $\alpha$ by 5-fold CV using your own CV code. You have to find the cross-validated accuracies for a number of $\alpha$-s between 0 and 1. Present the accuracy as a function of $\alpha$ on a plot and indicate which one is the best $\alpha$.

## 7 Final model performance

Finally (and now I mean finally☺), estimate the model performance on the testing data. Complete this section after everything else is done and you are ready to submit your work. **Don't improve model after you have loaded testing data!**

1. Fit your NB model using the cross-validated optimal alpha using your complete work data (both training and validation). This is your best and final model.
2. Load your testing data. Clean it using exactly the same procedure (you made a function for this, right?) and transform it into BOW-s.

   Note: above I suggested using `vectorizer.fit_transform(quote)` function to create the BOW. Here I recommend to use `vectorizer.transform(quote)`. This is because we don't want to change the vocabulary (that's what the `fit`-part does), only to transform it into the BOW.
3. Predict the F/R class on testing data. Compute accuracy. Present it.
4. Did you get a better or worse result compared to the k-NN and TF-IDF in PS04?

That is it. This is your final model performance measure. Feel free to compare it with your peers, but even if abysmal, **don't play with the model any more**! Just submit, and you are done ☺... really done, this was your last problem set!