A K-NEAREST NEIGHBOR AGGRESSION CLASSIFIER OF TWEETS

Text Mining, Radboud University Nijmegen

VOLKER STROBEL

s4491491

v.strobel@student.ru.nl

December 1, 2014

Abstract

Tweets on Twitter may contain aggressive content. This paper describes *Haggressive*, a supervised, k-Nearest Neighbors machine learning classifier that distinguishes between aggressive and non-aggressive tweets. It uses a bag-of-unigram approach for extracting features. The empirical evaluation of the approach has yielded an average accuracy of 68.57 % for $k = 10$.

1 INTRODUCTION

Twitter is the mouthpiece of the 21st century. Every second, around 6000 Twitter messages (so-called tweets), are sent out to the world. This equates to 500 million tweets per day[1]. As a consequence, it is impossible to manually analyze all the data, extract information from it, and see connections within it. Yet, such a large amount of data could well provide us with valuable insights on the level of the individual, group, culture, or even humankind.
Text classification can help to find and structure information in a large corpus of text. In order to do this, the text is preprocessed first. Afterwards, machine learning algorithms and statistical methods are applied to extract sought after information, perform categorizations, or discover interesting patterns [1]. It has wide-spread applications in spam-filtering [1] but is also used in such domains as search-filtering [2], sentiment analysis, and dimensionality reduction [3].
However, the microblogging site Twitter presents particular challenges in text classification. The large number of different words does not only require extensive computational power but also sophisticated machine learning algorithms. The terseness of messages, with a maximum length of 140 characters[2], and the informality of tweets call for a careful selection of features for supervised machine learning classifying techniques [4].
This work deals with binary text categorization of tweets using a k-Nearest Neighbors (kNN) algorithm. The two pre-defined labels *aggressive* and *non-aggressive* will be used to classify the content of a tweet. In Section 2, I discuss previous work on text classification and Twitter sentiment analysis. Section 3 describes the computation steps and the used kNN algorithm of the proposed tool *Haggressive*. In Section 4 the classification results are presented. Section 5 discusses these results. The paper concludes with a discussion of possible improvements and future extensions for in Section 6.

2 RELATED WORK

Text categorization has experienced a boom since Hearst's article "Untangling text data mining" (1999), in which he states: "In this paper, I have attempted to suggest a new emphasis: the use of large online text collections to discover new facts and trends about the world itself." [5, p. 8]. With the inception of

[1] urlhttp://www.internetlivestats.com/twitter-statistics/
[2] https://dev.twitter.com/overview/api/counting-characters

social networking services, like Facebook[3], Twitter[4], and Instagram[5], the amount of user-generated data has skyrocketed. These sites provide interesting sources of information because people share opinions and thoughts openly, demonstrate what they care and do not care about, and provide insight into their daily lives. Such personal information from such a vast amount of individuals could not be obtained in laboratory settings.
One popular research topic in this field is Twitter sentiment analysis. This is fueled by the interest of companies in the reception of their products [6]. For deciding, if a tweet is either positive or negative, accuracies of 80 % and higher have been achieved [7]. This raises the question whether such opinion mining cannot be taken a step further to classifying tweets according to their underlying emotional timbre, i.e. if the tweet is calm, sad, aggressive, etc. While several studies have focused on emoticons in opinion mining (e.g. [8]), only few have actually attempted to extract affective information from the words in tweets. In 2012, Roberts et al. [9] undertook the task of classifying tweets as expressing *love*, *sadness*, *anger*, *joy*, *disgust*, *fear*, or *surprise*. This was done by combining seven different classifiers, each using a different combination of features suited for classifying one of the affective states. Thus, one tweet could be annotated with several emotions, depending on the number of positive classifications. They obtained the highest F_1 score for *fear*, despite *fear* being less frequent in the training data and the classifier only using two features. Especially in the context of ongoing debates about the dangers of online anonymity and the associated disinhibition effect, a reliable identification of fearful utterances could be helpful in identifying and interfering with phenomena like cyber-bullying and flaming[6]. In the same line, it could be argued that preventing bullying altogether would be even better than just interfering with it [10]. With Twitter, a first step to such an end could be the classification of tweets as being either aggressive or non-aggressive.

3 METHOD

In order to classify new tweets as being either aggressive or not, a k-Nearest Neighbors (kNN) algorithm is implemented. This hinges on the idea that for each tweet in the test set, the k most similar tweets from the training set are picked and the unclassified tweet receives the label of most frequent class among its neighbors. The method was chosen because it is popular in text mining and often performs well in comparison to other methods (e.g. [11, 12]). Furthermore, the programming language Haskell [13] was chosen for the implementation. To the best of my knowledge, there is no general framework for sentiment analysis and text classification using Haskell to date. However, the Haskell's functional paradigm provides an appropriate means for numerous operations on text[7]. The following section explains the approach of the tool *Haggressive* in more detail.

3.1 *Data*

3150 annotated tweets (aggressive, non-aggressive) were used in this study. The tweets were represented in a tab-delimited file consisting of five columns (for an example of tweet, see Table 1):

The complete data set (3150 tweets) consists of an equal amount of aggressive and non-aggressive tweets (1575 examples of each category). The labeling of tweets was conducted by several human annotators over time.

3.2 *Procedure*

3.2.1 *Data set analysis*

To decide which features are the most useful in distinguishing between aggressive and non-aggressive tweets, I checked the distribution of word frequencies in the data set. Figure 1 shows the terms ordered by frequency (log rank) in comparison to the total frequency of occurrence (log frequency). It can be

[3] https://www.facebook.com/
[4] https://twitter.com/
[5] http://instagram.com/
[6] So called 'flaming', is a form of aggression behavior in which other Internet users are insulted.
[7] see, e.g., `http://www.quora.com/Is-Haskell-a-good-fit-for-machine-learning-problems`

label	user name	date	time	message
aggressive	-	-	-	eerst en bitch altijd en bitch
aggressive	Saardehosselaar	2013-07-04	18:56:01	USER Die stoel op je ava , die is pas zielig .
non_aggressive	-	-	-	Toch veel vis in vijver #Veenendaal URL
non_aggressive	DNMS_	2013-07-03	13:55:54	Zelfgemaakte pizza eten URL

Table 1: This table shows four tweets, represented in tab-delimited format that were part of the used data set. The majority of tweets did not contain information about user name, date, and time of the tweet. User names and URLS were already tokenized and normalized in the provided data set.

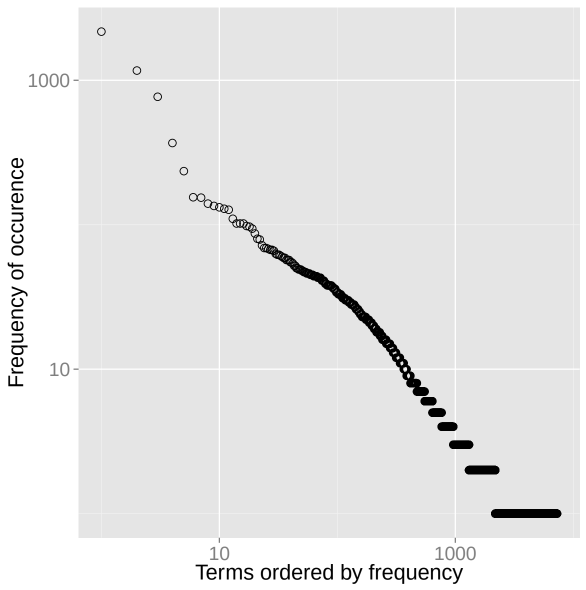

Figure 1: Zipf's distribution of words.

seen that it follows Zipf's law, and therefore represents a proper distribution of words in the data set [8].

Table 2 shows the 15 most frequent words after converting all characters to lower case, split into aggressive and non-aggressive tweets. With "user" and "rt"[8] occurring approximately equally often in both categories, I treated them as stop words.

3.2.2 *Feature extraction*

The implementation uses the bag-of-unigrams approach, where the relative position of a word in a message is completely ignored. In the bag, features are represented as unigrams in lower case letters with their corresponding frequency[9]. A list containing 104[10] common Dutch stop words was used to remove unimportant words. Hashtags were treated as normal words, removing the hash sign.

3.2.3 *Implementation of kNN*

A *k*NN classifier is used to determine the *k* closest tweets. This hinges on the idea that for each unclassified tweet, the *k* most similar tweets from the training set are picked and the unclassified tweet receives the most frequent class among its *k* neighbors[11]. For the implementation of the *k*NN-algorithm,

[8] "RT" or "rt" seems to stem from the pre-processing of the data set.
[9] Capitalization rules are not strictly followed on the Internet
[10] http://www.ranks.nl/stopwords/dutch
[11] The method was chosen because it is popular in text mining and often performs well in comparison to other methods (see, e.g., [11, 12]).

aggressive		non-aggressive	
word	frequency	word	frequency
kkr	69	goed	35
bitch	73	kut	37
gewoon	73	mensen	43
s	83	jij	45
man	84	gaat	46
ga	86	s	46
hou	89	gaan	47
bent	102	ga	49
jij	109	morgen	49
kk	120	weer	79
bek	144	echt	87
echt	148	url	313
rt	392	rt	377
je	782	je	386
user	1107	user	1065

Table 2: Shown are the 15 most common words in the used data set, split into aggressive and non-aggressive tweets.

the features of each tweet in the test set (sum total 315) were compared to each tweet in the training set (2835). This was done for each fold of the data set. For the comparison, the cosine similarity was chosen. As the features only consisted of positive values, the similarity was bounded in $[0, 1]$, where 0 means no similarity, and 1 perfect similarity.

3.3 *Evaluation*

3.3.1 *Accuracy*

For the empirical evaluation, the data set of 3150 tweets was split into 10 folds of 315 tweets each. Since the data set and every fold consisted of an equal amount of aggressive and non-aggressive tweets, chance level for classification was 50 %. A leave-one-out cross-validation was conducted for all values of $k \in \{1, 2, ..., 30\}$. To this end, each of the ten folds served as a test set once, and the other nine folds as training sets. The average accuracy on data-set level is calculated by dividing the correctly assigned labels by the total number of assigned labels for all folds of the data-set.

3.3.2 *Time Resources*

The classsification was running under Arch Linux x64, Linux Kernel Version 3.16.3 with 2688 MB RAM and a Intel(R) Core(TM) i5-4570 CPU 3.20 GHz processor.

4 RESULTS

Figure 2 shows the averages accuracies for all folds (that is, on data set level) for different values of k. The highest accuracy, on data set level, was achieved by using $k = 10$, with an average accuracy of 68.57 %, (range on fold-level for $k = 10$: 65.08 % – 70.79,%) and a standard deviation of 1.75 % (see Figure 3). The highest accuracy on fold-level[12] in the entire data-set was 74.29 %, with $k = 3$.

4.0.3 *Time Resources*

The running time of the leave-one-out cross-validation was 605.20 s.

[12]Using fold_8.txt as test set

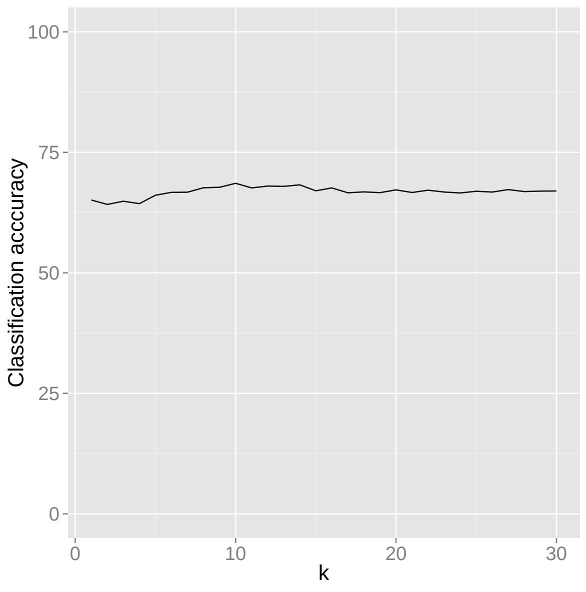

Figure 2: This table shows the average accuracy for different values of *k*

5 DISCUSSION

The results show that approximately two out of three tweets could be correctly classified by using $k = 10$. Figure 2 displays that the classification accuracy is largely independent of the choice of *k*, nevertheless, further research is needed for determining a consistent good *k* value.
In order to improve the accuracy of *Haggressive*, I will discuss two possible options, feature extraction and the choice of the classifier.

5.1 *Feature extraction*

While in the current data set, date and time were not always available, it could well be that they have an influence an aggression labeling. Major political events at a certain date or higher aggression during night time are not unreasonable.

To check for overfitting, the software should be evaluated on another data set. Especially the value of *k* should be tested again.

5.2 *Classifier*

*k*NN-classifiers often perform well in text mining tasks. The current implementation assigns each tweet the most frequent class among its *k* neighbors. A weighting scheme that takes the distance to each neighbor into account could be beneficial. Metric learning could have further positive effects [14].

Furthermore, several other machine learning classifiers, for example, Support Vector Machines (SVM) and Naive Bayes classifier have proven useful in various text classification tasks.

5.3 *Running Time*

For the first prototype of *Haggressive*, running time was not a concern. The current implementation extracts all features again and again for every tweet comparison. This results in almost 9 million feature extractions for the given corpus when doing ten cross validations[13]. Instead, it would be convenient to store features of the training set in a database with the possibility of updating the database, if the training set changes.

[13]2835 training tweets $\times$ 315 test tweets $\times$ 10 cross validations

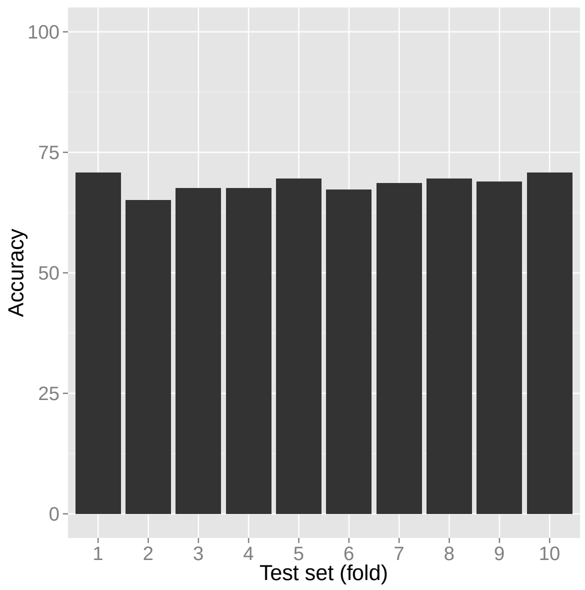

Figure 3: Per-fold accuracy in % for $k = 10$ (best k-value found for the used data set).

6 CONCLUSION & FUTURE WORK

Microblogging offers a valuable means for obtaining large amounts of data and mining this data for important patterns.

This paper presented *Haggressive*, a Haskell framework for classifying tweets by means of their aggressive content. *Haggressive* is, by using the bag-of-unigrams approach, relatively task-independent and should be able to yield above-chance results in other sentiment analyses: Given a labeled tweet set, it is not only able to distinguish between aggressive and non-aggressive tweets but can be used for a wide range of binomial classification tasks.

Since Haggressive is a first prototype, various opportunities for future work exist. For one, it can be further extended by making use of the Twitter API[14] and thereby providing an interface for rating tweets and the aggression ratio of users. Furthermore, it could be worthwhile to develop a web-interface for easy access[15], use the proposed kNN-classifier for other languages, or compare classification rates of the kNN implementation with that of other classification algorithms.

[14] e.g., with https://github.com/himura/twitter-conduit
[15] similar to http://www.tweetgenie.nl/

The code can also be found at:

- GitHub: `https://github.com/Pold87/Haggressive`
- Hackage[16], including Haddock documentation[17]: `http://hackage.haskell.org/package/Haggressive`

A.1 *Hag.hs*

```haskell
{-# OPTIONS_GHC -Wall #-}
{-# LANGUAGE OverloadedStrings #-}


{-|
Module      : Hag
Description : Classify Tweets (aggressive vs. non_aggressive) and evaluate
              classification performance.
License     : None
Maintainer  : Volker Strobel (volker.strobel87@gmail.com)
Stability   : experimental
Portability : None

This module is the main interface for Tweet classification.

-}
module Hag
( module Hag
 ) where

import qualified Data.ByteString.Lazy as L
import           Data.Char
import           Data.Csv
import           Data.Either
import           Data.List
import qualified Data.Map             as M
import qualified Data.PSQueue         as PS
import qualified Data.Text            as T
import           Data.Text.Encoding
import qualified Data.Text.IO         as TI
import qualified Data.Vector          as V
import           Helpers
import           NLP.Tokenize
import           Preprocess
import qualified System.Directory     as S
import           System.Environment
import           Tweets


-- | Features are represented by a 'M.Map', where the keys are
-- 'String's (e.g., the words in the message of a Tweet) and the
-- values are 'Float's (e.g., the number of occurrence of a word)
type FeatureMap = M.Map String Float
```

[16]Hackage is Haskell's central package archive
[17]Haddock is Haskell's documentation generation system

```
-- |
-- =IO and Parsing

-- |'parseCsv' parses a 'T.Text' input for fields in CSV format and
-- returns a 'Vector' of 'Tweet's
parseCsv :: T.Text -> Either String (V.Vector Tweet)
parseCsv text = decodeWith
    defaultDecodeOptions {decDelimiter = fromIntegral $ ord '\t' }
    NoHeader
    (L.fromStrict $
     encodeUtf8 text)

-- |Get directory contents of 'FilePath'. A better variant is at:
getFiles :: FilePath -> IO [FilePath]
getFiles dir = S.getDirectoryContents dir
               >>= return . map (dir ++) . filter (`notElem` [".", ".."])

-- |Extract features (for the bag of words) for one Tweet.
-- Thereby, the Tweet will be (in order of application):
-- * tokenized
-- * converted to a 'V.Vector'
-- * 'String's will be converted to lowercase
-- * 'String's that are not 'isAlpha' are removed
-- * 'String's that are element of 'stopWords' are removed
-- * Empty 'String's will be removed
extractFeatures :: Tweet -> FeatureMap
extractFeatures tweet = bagOfWords
  where
    pre = V.filter (`notElem` ["USER", "RT"])
                 . V.fromList
                 . tokenize
                 . tMessage
    bagOfWords = frequency
                 . V.filter (/= "")
                 . V.filter (`notElem` stopWords)
                 . V.map (map toLower)
                 . pre
                 $ tweet

-- |Calculate the 'frequency' of items in a 'V.Vector' and return them
-- in a 'M.Map'.
frequency :: V.Vector String -> FeatureMap
frequency = V.foldl' countItem M.empty

-- |Insert an item into a 'M.Map'. Default value is 1 if the item is
-- not existing. If the item is already existing, its frequency will
-- be increased by 1.
countItem :: M.Map String Float -> String -> FeatureMap
countItem myMap item = M.insertWith (+) item 1 myMap

-- |Take a 'M.Map', consisting of
-- key: 'Tweet'
-- value: 'FeatureMap'
-- and one Tweet and create a new 'M.Map' with the added features
-- from the 'Tweet'
```

```haskell
insertInMap :: M.Map Tweet FeatureMap
               -> Tweet
               -> M.Map Tweet FeatureMap
insertInMap oldMap tweet = M.insert tweet val oldMap
  where val = extractFeatures tweet

-- | Compare two vectors of 'Tweet's, the first is the test vector,
-- the second the train vector and return the all neighbors for each
-- 'Tweet'. 'grandDict' is a 'M.Map', where each entry consits of a
-- 'Tweet' and its features
getNeighbors :: (V.Vector Tweet, V.Vector Tweet)
            -> V.Vector (Tweet, PS.PSQ Tweet Float)
getNeighbors (v1,v2) =  V.map (featureIntersection dictionary) v1
  where dictionary = V.foldl insertInMap M.empty v2 :: M.Map Tweet FeatureMap

-- | Take a dictionary and a 'Tweet' and return a pair of this 'Tweet'
-- and all its nearest neighbors
featureIntersection :: M.Map Tweet FeatureMap
                            -> Tweet
                            ->  (Tweet, PS.PSQ Tweet Float)
featureIntersection tweetMap tweet = (tweet, mini)
  where
    mini = PS.fromList
               $ M.elems
               $ M.mapWithKey (mergeTweetFeatures cosineDistance tweet) tweetMap

-- |Take a distance function, 'Tweet' 1, 'Tweet' 2 and a dictionary as
-- 'FeatureMap' and create a 'PS.Binding' between 'Tweet' 2 and the
-- distance from this 'Tweet' to the other 'Tweet'.
mergeTweetFeatures :: (FeatureMap -> FeatureMap -> Float)
                      -> Tweet
                      -> Tweet
                      -> FeatureMap
                      -> PS.Binding Tweet Float
mergeTweetFeatures distF t1 t2 _ = queue
  where featuresT1 = extractFeatures t1
        featuresT2 = extractFeatures t2
        distance = distF featuresT1 featuresT2
        queue = t2 PS.:-> distance

-- |Take the features of two 'Tweet's and return the distance as
-- 'Num'.
cosineDistance :: FeatureMap ->  FeatureMap -> Float
cosineDistance t1 t2 = negate (mySum / (wordsInT1 * wordsInT2))
  where
    wordsInT1 = M.foldl (+) 0 t1
    wordsInT2 = M.foldl (+) 0 t2
    intersection = M.elems $ M.intersectionWith (*) t1 t2
    mySum = foldl (+) 0 intersection

-- |Takes a dictionary and a mini dictionary (frequency of words in
-- one Tweet) and calculates the idftf values for all words in the
-- mini dictionary.
idftf :: FeatureMap -> FeatureMap -> FeatureMap
idftf grandDict miniDict =  M.mapWithKey (iFrequency grandDict) miniDict
```

```haskell
iFrequency :: FeatureMap -> String -> Float -> Float
iFrequency dict word freq = freq * (log (totalNumberOfWords / freqWord))
  where freqWord =  M.findWithDefault 1 word dict
        totalNumberOfWords = M.foldl (+) 0 dict

-- | Calculate the amount of tweets where the predicted label matches
-- the actual label.
compareLabels ::  Int -> V.Vector (Tweet,PS.PSQ Tweet Float) ->  V.Vector Float
compareLabels k vec = V.map
                    (\(a,b) -> if (tLabel a) == getLabel k b then 1 else 0)
                    vec

compareLabelsForScheme :: [V.Vector (Tweet,PS.PSQ Tweet Float)] -> Int -> [Float]
compareLabelsForScheme vecs k = map (getAccuracy . compareLabels k) vecs


-- | Get the label for a 'Tweet' by looking at the k nearest
-- neighbors. If there are more aggressive than non_aggressive
-- 'Tweet's, the label will be aggressive, otherwise, it will be
-- non-aggressive.
getLabel :: Int -> PS.PSQ Tweet Float -> String
getLabel k queue = if agg >= nonAgg then "aggressive" else "non_aggressive"
  where tweets = queueTake k queue
        labels = map tLabel tweets
        agg = length $ filter (== "aggressive") labels
        nonAgg = length $ filter (== "non_aggressive") labels

-- | Get sum total of a vector of floats (i.e., the number of
-- correctly classified tweets) and return the accuracy
getAccuracy :: V.Vector Float -> Float
getAccuracy vec =   (V.foldl (+) 0 vec) / fromIntegral (V.length vec)

main :: IO ()
main = do
  -- Retrieve command line args
  (dir:_) <- getArgs

  -- Get list of files in directory dir
  files <- getFiles dir

  -- Read content of all files into list, one Text per item
  csvs <- mapM TI.readFile $ sort files

  let
    -- Add quotation marks for CSV parsing
    processedCsvs = map preprocess csvs

    -- Create Tweets from Text
    r = map parseCsv processedCsvs

    -- If parsing was successful, extract Right elements from the
    -- Either list
    tweets = rights r

    -- Create a leave-one-out cross-validation scheme
    scheme = mkCrossValScheme tweets
    -- Get all neighbors for all tweets for all schemes
    allNeighbors = map getNeighbors scheme

    -- k should be in {1..100}
    ks = [1..100]

    -- Compare all training tweets to test tweets for all ks
    comparedTweets = map (compareLabelsForScheme allNeighbors) ks

    -- Prepare for CSV
    results = encode comparedTweets

  -- Create header
    header = encode
             $ map (("fold_" ++) . show) ([1..10] :: [Integer])

  -- Write output to a file
  L.writeFile "resultsK.csv" $ header `L.append` results
```

```haskell
module Helpers
       (mkCrossValScheme
       , queueTake
       , stopWords)
       where

import qualified Data.PSQueue     as PS
import qualified Data.Vector      as V
import           Debug.Trace
import           System.IO.Unsafe

-- | 'FilePath' of the 'stopWords' that are removed before the
-- dictionary is created
stopWordsFile :: FilePath
stopWordsFile = "../dutch-stop-words.txt"

-- | Read the 'stopWords' from the 'stopWordsFile'
stopWords :: [String]
stopWords =  lines file
  where file = unsafePerformIO $ readFile stopWordsFile

-- | Make a cross-validation scheme from a list of vectors
mkCrossValScheme :: (Eq a) =>  [V.Vector a] -> [(V.Vector a,V.Vector a)]
mkCrossValScheme xs = map (leaveOneOut xs) xs

-- | Create pair of a list of vectors and a vector that specifies
-- which vector should be left out
leaveOneOut :: (Eq a) => [V.Vector a] -> V.Vector a -> (V.Vector a,V.Vector a)
leaveOneOut all test = (test, V.concat $ filter (/= test) all)

-- | Take the first k elements of a queue
queueTake :: (Ord k, Ord p, Show k) => Int -> PS.PSQ k p -> [k]
queueTake k queue = queueTake' k queue []

-- | Helper function for 'queueTake'
queueTake' :: (Ord k, Ord p) => Int -> PS.PSQ k p -> [k] -> [k]
queueTake' 0 _ acc = acc
queueTake' k queue acc = case mini of
  Nothing -> []
  Just m -> queueTake' (k - 1) (PS.deleteMin queue) (PS.key m:acc)
  where mini = PS.findMin queue
```

A.3 *Preprocess.hs*

```haskell
{-# OPTIONS_GHC -Wall #-}
{-# LANGUAGE OverloadedStrings #-}

module Preprocess
(preprocess)
where

import qualified Data.Text as T

preprocess :: T.Text -> T.Text
preprocess txt = T.cons '\"' $ T.snoc escaped '\"'
  where escaped = T.concatMap escaper txt

escaper :: Char -> T.Text
escaper c
  | c == '\t' = "\"\t\""
  | c == '\n' = "\"\n\""
  | c == '\"' = "\"\""
  | otherwise = T.singleton c
```

A.4 *Tweets.hs*

```haskell
module Tweets
  (filterByLabel
, Tweet(..)) where

import           Control.Applicative ((<$>), (<*>), (<|>))
import           Control.Monad       (mzero)
import           Data.Csv
import qualified Data.PSQueue        as PS
import qualified Data.Vector         as V



-- | Parsing Record to Tweet
instance FromRecord Tweet where
  parseRecord v
         | V.length v == 5 = Tweet <$>
                             v.! 0 <*>
                             v.! 1 <*>
                             v.! 2 <*>
                             v.! 3 <*>
                             v.! 4
         | otherwise = mzero



-- | A Tweet consists of a category, a user, a date, a time, and a
-- message
data Tweet = Tweet { tLabel   :: String
                   , tUser    :: String
                   , tDate    :: String
                   , tTime    :: String
                   , tMessage :: String
                   } deriving (Show, Eq, Ord)


-- | Filter 'Tweet's by label
filterByLabel :: V.Vector Tweet -> String -> V.Vector Tweet
filterByLabel tweets label = V.filter (\t -> tLabel t == label) tweets
```