Machine Learning Final Project
Spam Email Filtering

March 2013

- Shahar Yifrah
- Guy Lev

Table of Content

1. Overview

In this project we were interested in examining spam email filtering.

We used 6 different algorithms introduced in class to implement spam filtering in Matlab, and compared their performance.

The algorithms we used were:

- Adaboost
- Naïve Bayes
- Perceptron
- Winnow
- SVM (Support Vector Machine)
- KNN (K Nearest Neighbors)

In this document we use the following terminology:

- Ham message: legitimate message (i.e. non-spam)
- False positive ratio: (number of false positives) / (number of ham messages)
  (Note that this ratio may be higher than the error rate).

We examine the performance of each algorithm in 2 aspects:

- Error rate
- False positive ratio

False positive ratio is interesting because filtering out a ham message is a bad thing, worse than letting a spam message get through.

2. Dataset

2.1 Source

We used part of the Enron Spam datasets, available at CSMining website:

http://csmining.org/index.php/enron-spam-datasets.html

Note that this dataset contains preprocessed email messages (basic "cleaning"), as described at the above link.

The dataset size is 5172 messages. The ratio between spam and ham messages is: 29% spam, 71% ham.

2.2 Creation of Training and Test Sets

We wrote a Python script to process these messages and create a feature vector out of each message. In the sequel it is described how the feature vectors look like.

The script divides the feature vectors into training set and test set, while preserving the ham-spam ratio in each set.

Actually, the script randomly creates 90 different pairs of training set and test set as follows:

- We used 9 different "training fractions", i.e. the percentage of training set size out of the entire dataset. The training fractions we used were: 0.1, 0.2, …, 0.9.
- For each training fraction, we randomly created 10 different pairs of training set and test set, so we can examine the performance as an average of 10 runs.

Note that for Adaboost and SVM we used only part of these pairs of sets due to long running time.

The Python script's location is: Project\Data\enron1\process.py. In the last section (Project Website and Files) it is explained how to get there.

2.3 Feature Vectors Content

The script selects 100 words out of all words which appear in messages in the dataset (in the sequel it is described how these 100 words are chosen). Each of these words yields a feature: the "within-document-frequency" of this word, i.e.:

(number of appearances of the word in the email) / (total number of words in the email).

A "word" in this case can be also one of the following characters:

;([!$#

We chose the number of features (100) empirically.

2.4 Features (Words) Selection

2.4.1 Preface

Given a training set of spam and ham messages, we want to choose these 100 words according to which we will prepare the feature vectors.

2.4.2 Spamicity

Denote:

Pr(w|S) = the probability that a specific word appears in spam messages

Pr(w|H) = the probability that a specific word appears in ham messages.

We define **Spamicity** of a word:

$$spamicity(w) = \frac{\Pr(w|S)}{\Pr(w|S) + \Pr(w|H)}$$

Each probability is estimated using the relevant proportion of messages in the training set:
Pr(w|S) is estimated by fraction of spam messages containing the specified word.

Pr(w|H) is estimated by fraction of ham messages containing the specified word.

2.4.3 Selection of Words

In this process we rank each word, and finally choose the 100 words with highest ranks.

The intuitive practice is to assign high rank to words whose spamicity is far away from 0.5 (higher or lower). If spamicity is close to 1 then this word should serve as a good spam indicator. If spamicity is close to 0 then it should be a good ham indicator.

We found, however, that looking at spamicity only is not sufficient: words for which both Pr(w|S) and Pr(w|H) are too small cannot serve as good indicators even if their spamicity if far from 0.5.

Therefore we also look at how big the following absolute difference is:

|Pr(w|S) - Pr(w|H)|

The process of words selection is as follows:

- Filter-out words with |spamicity – 0.5| < 0.05
- Filter-out rare words, namely appearing (in ham and spam in total) less than a given threshold (1%).
- For each word which was not filtered-out, calculate |Pr(w|S) - Pr(w|H)|
- Choose the 100 words with biggest |Pr(w|S) - Pr(w|H)|

2.5 Shuffling

Ham-spam ratio is dynamic over time, and it would be interesting to cope with that, but we chose not to due to the limited scope of the project.
Authors of [2006][1] studied this issue and found that Naïve Bayes classifier adapts quickly to the new ratio if trained gradually.

---

[1] [2006] Spam Filtering with Naive Bayes -- Which Naive Bayes? (2006)
by Vangelis Metsis Telecommunications , Vangelis Metsis
http://www.aueb.gr/users/ion/docs/ceas2006_paper.pdf

We randomly shuffle both training and test sets before use so that its ham-spam ratio over time appears uniform for our algorithm.

3. AdaBoost

We implemented the standard AdaBoost algorithm, see [1] or [2] in the *references* section below.

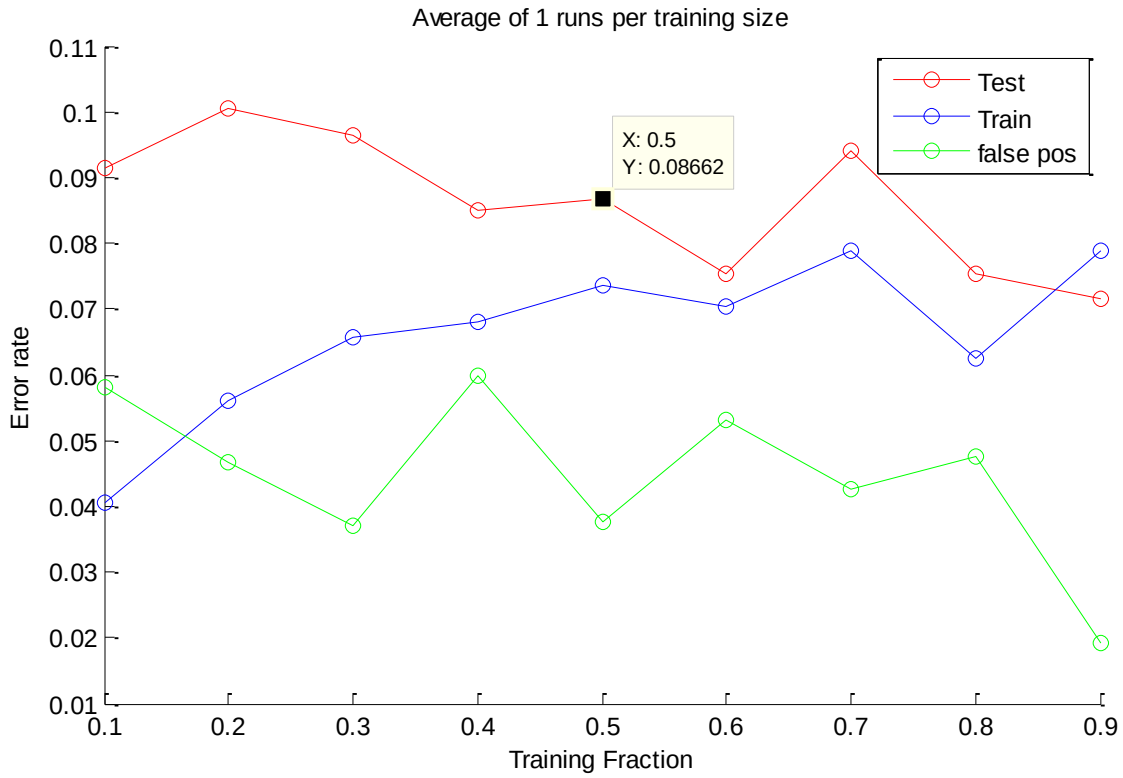
3.1 Weak-Learner Family

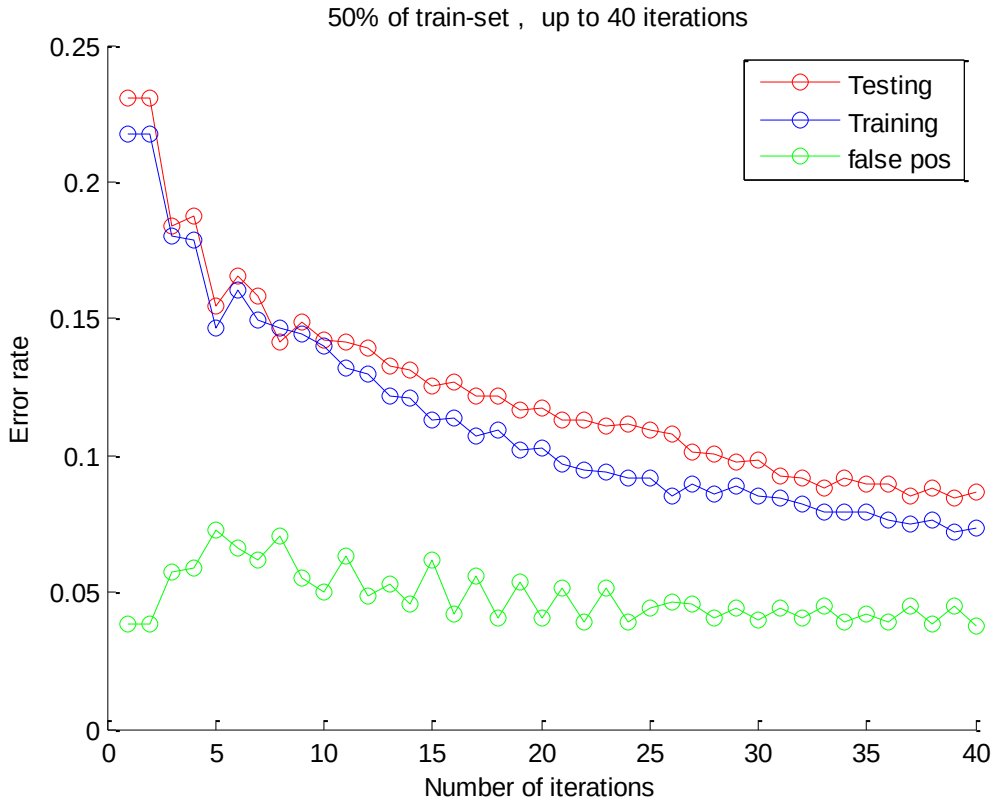
Each classifier from the weak learner family uses a threshold over the "within-document-frequency" of a specific word. It tries all frequencies appearing in the training set, and takes the one minimizing the current "weighted error" of misclassification.
The weights are then updated to emphasize misclassified rows. This is done for a few tens of iterations. The final classifier is a linear combination of all those detected weak-learners.

3.2 Precision / Recall Tradeoff

For general introduction to precision/recall, see [1].

The standard Adaboost algorithm minimizes total of classification error rate.
Our domain is ham-spam filtering, so we prefer precision to recall (lower false-positive error rate).
We would like more control over the trade-off of the two error types, in order to reduce false-positive rate.
But the standard Adaboost algorithm is symmetric by design.
We looked for ways to make it prefer false-positive over false-negative, perhaps by tweaking the weights in the iteration. But the discussion in [4], section 4, "Asymmetric AdaBoost", made us realize that it is not an easy task, and it is out of the scope of this project.


Average of 1 runs per training size
Test
Train
false pos
X: 0.5
Y: 0.08662
Error rate
Training Fraction
0.11
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9


50% of train-set , up to 40 iterations
Testing
Training
false pos
Error rate
Number of iterations
0.25
0.2
0.15
0.1
0.05
0
0
5
10
15
20
25
30
35
40

3.3 References

1. Wikipedia
   http://en.wikipedia.org/wiki/Precision_and_recall
2. Class lecture scribe
   http://www.cs.tau.ac.il/~mansour/ml-course-12/scribe6_boosting.pdf
3. Wikipedia
   http://en.wikipedia.org/wiki/Adaboost
4. Fast and Robust Classification using Asymmetric AdaBoost and a Detector Cascade
   Paul Viola and Michael Jones
   http://research.microsoft.com/en-us/um/people/viola/Pubs/Detect/violaJones_NIPS2002.pdf

4. Naïve Bayes

We implemented binary-domain Naïve-Bayes classifier, as described in class, see [1].
Following is the final formula for classification.
Denote:

X_i = word #i in the message to be classified
ws = frequency of given word in spam messages
wh = frequency of given word in ham messages
sp = proportion of spam messages in training-set, or prior belief about it in testing-set.

$$ratio = \log\left(\frac{sp}{1-sp}\right) + \sum \log\left(\frac{ws(X_i)}{wh(X_i)}\right)$$

The classification is "spam" if this ratio exceeds a threshold of zero.
Note that in order to avoid relying on Matlab behavior for logarithm-of-zero and division-by-zero, we first replace zero frequencies in *ws, wh* with a small number:

$$1 - \left(\frac{N-1}{N}\right)^{100}$$

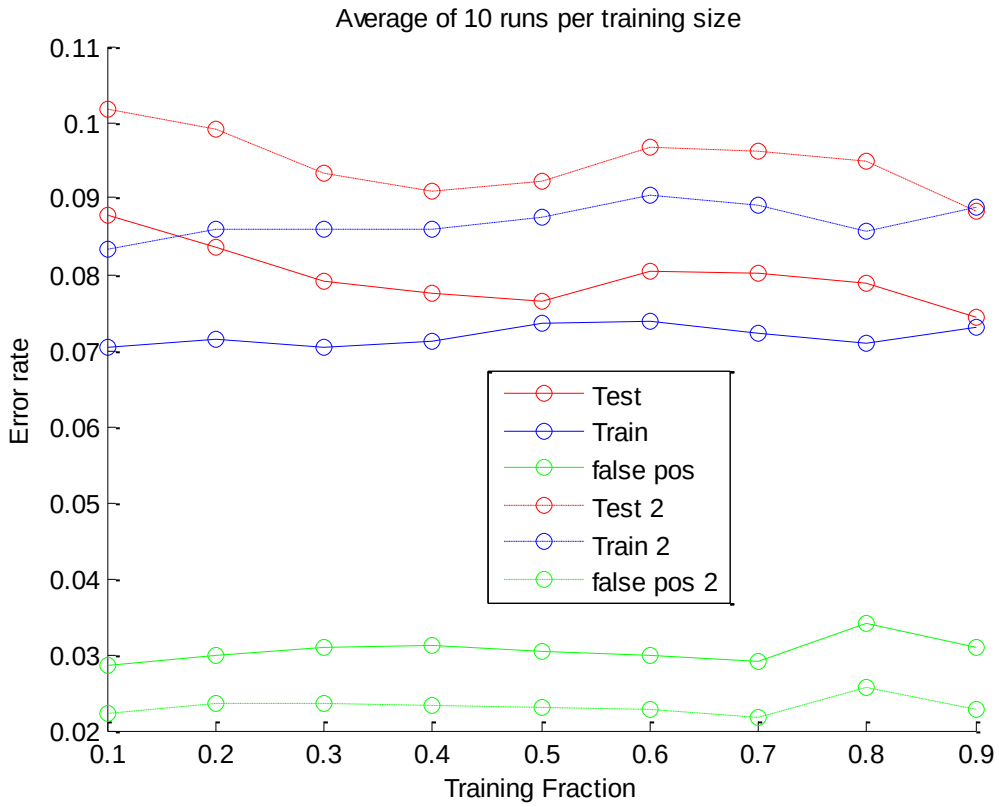
This number comes from estimating that there are N=250,000 English words, and about 100 words in a message. The N for spam dictionary is even larger, say 10 times than the ordinary ham dictionary. This is because spammers use twisted spelling to avoid getting blocked.
If we take higher threshold, we obtain lower false-positive error ratio (0.5%), at the expense of higher total error rate of 20% (specifically, false-negative, of course.)

4.1 Results and Discussion

In the Bayes-formula:

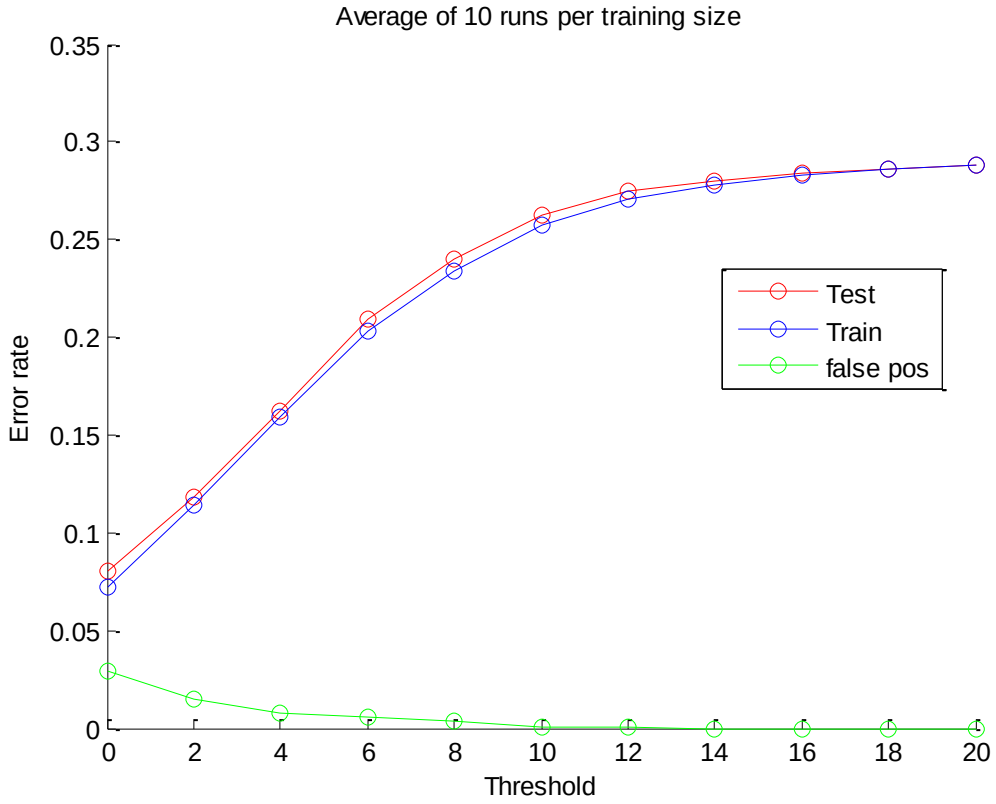
$$\Pr(S|W) = \frac{\Pr(W|S)\Pr(S)}{\Pr(W|s)\Pr(S) + \Pr(W|H)\Pr(H)}$$

The best results were 7.6% error rate, 3% false positive rate, when assuming 1:1 ham-spam ratio.
If using the actual ham-spam ratio (71:29) of the data for Pr(S) and Pr(H), the error is about 1.5% worse than when assuming 1:1 ratio. The false-positive is hurt only by 0.8% when assuming 1:1 ratio:



In order to get near-zero false-positive, we tried a larger threshold, and then we got to

20%-25% error, as seen in the following graph:



4.2 References

1. Class scribe
   http://www.cs.tau.ac.il/~mansour/ml-course-12/Scribe2_Bayes.pdf
2. Wikipedia
   http://en.wikipedia.org/wiki/Bayesian_spam_filter
   (Different formulation but equivalent to the one presented in class)
3. SpamProbe - Bayesian Spam Filtering Tweaks by Brian Burton
   (Addresses "Use of within document frequency" of each word)
   http://spamprobe.sourceforge.net/paper.html
4. A Statistical Approach to the Spam Problem
   (Addresses "Dealing with Rare Words")
   http://www.linuxjournal.com/article/6467
5. Spam Filtering with Naive Bayes -- Which Naive Bayes? (2006)
   by Vangelis Metsis Telecommunications , Vangelis Metsis
   http://www.aueb.gr/users/ion/docs/ceas2006_paper.pdf

5. Perceptron

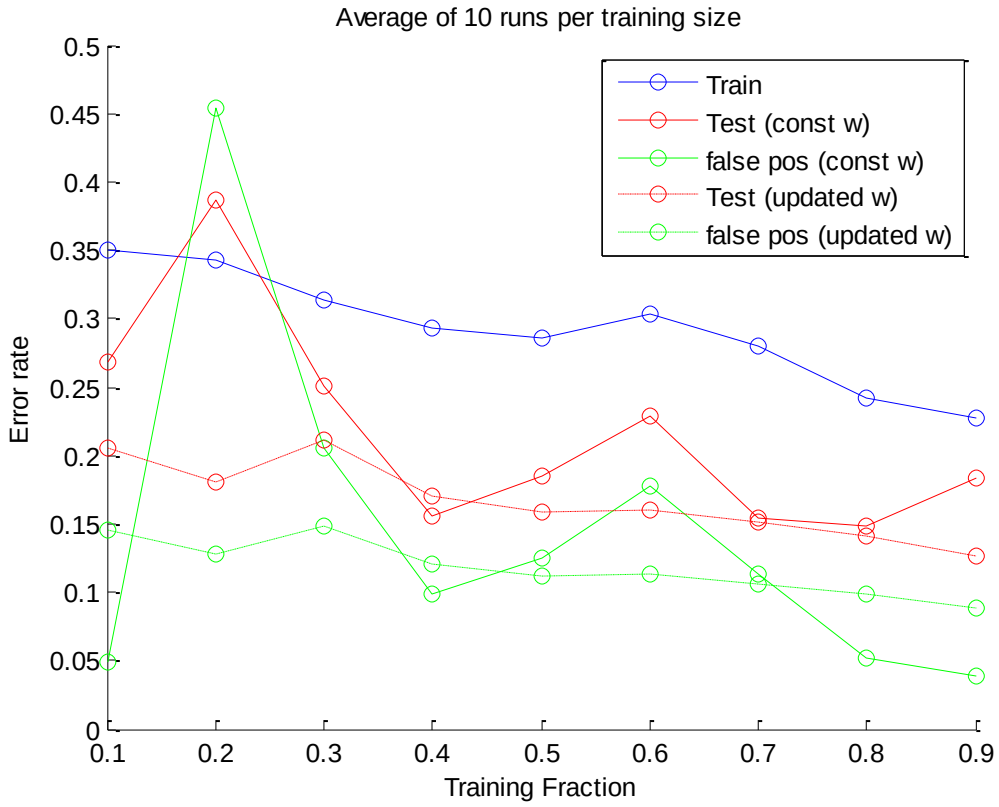
We implemented the Perceptron algorithm as taught in class, see [1].

At the end of training stage the algorithm outputs a weight vector w, which represents a hyperplane (hopefully a separating hyperplane, if one exists).

Then, we classified the vectors in the test set using this vector *w*.

It was also interesting for us to check how better the results are if we continue updating *w* in the test set classification as well.

The following graph shows the results:



We see that, expectedly, the results are better when continuing updating w. The difference is not dramatic though.

Perceptron actually had the worse error rate than all algorithms we checked.

We saw in class that if a separating hyperplane exists, then Perceptron has a mistake bound which depends on the hyperplane's margin.

It is hard to check if a separating hyperplane exists, and what the margin is.

We can conclude that either there is no such hyperplane, or the training set size was too small for the algorithm to reach the mistake bound.

5.1 References

1. Class scribe
   http://www.cs.tau.ac.il/~mansour/ml-course-12/scribe4_online.pdf

6. Winnow

In class we saw Winnow algorithm which learns the class of monotone disjunctions. In this case the feature vectors contain binary values. When the algorithm makes a mistake on a vector *x*, the "active features" (i.e. the indices where the weights should be modified) are chosen according to whether $x_i = 1$.

Our case is different: the feature vectors have real values rather than binary. We had to define how the active features are chosen. We used the following method:

- Given the training set, we calculate the average value of each feature.
- The active features are chosen according to whether $x_i > 10 * AVG_i$

The factor of 10 was chosen empirically – it was the one which gave best results.

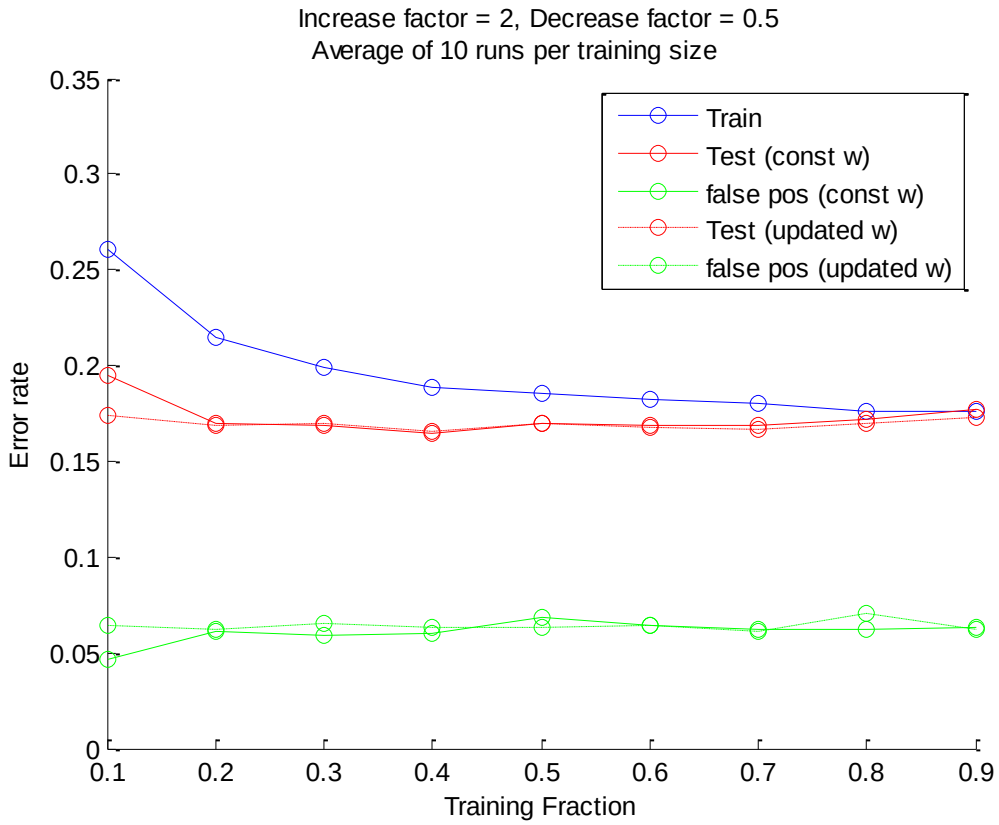
This criterion gave better results than criterion which used the standard deviation of the feature.

Another thing we had to choose was the threshold to which we compare the dot product *<x,w>* (the threshold we used in class, which was the dimension, is irrelevant in our case).

The threshold was chosen empirically to be 20. However, all thresholds far enough from 0 gave pretty similar results.

As discussed in Perceptron section, we also checked the performance in the case we let the algorithm update the weight vector during classification of the test set.

The following results were obtained:



We see that Winnow has error rate around 17%, a bit better than Perceptron.

False positive ratio is also better.

Continuing updating the weight vector during classification of test set made almost no difference.

Like Perceptron, Winnow has a mistake bound which depends on the separating hyperplane's margin. However it is hard to tell whether such hyperplane exists (and how big the margin is).

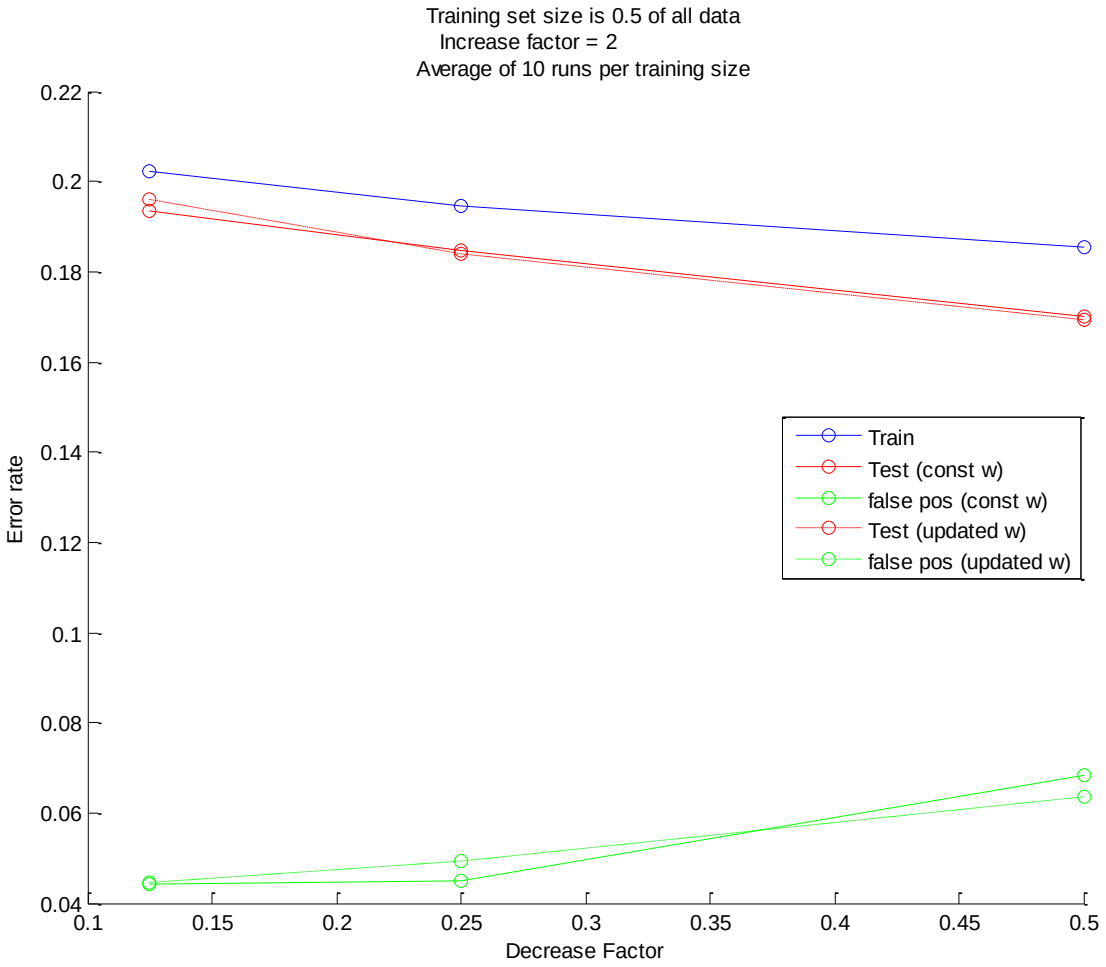
6.1 Reducing False Positive Ratio

In the above we used symmetric increase and decrease factors, i.e.:

- When a weight should be increased, it is multiplied by 2.
- When a weight should be decreased, it is multiplied by 0.5.

We tried to reduce the false positive ratio by using asymmetric factors. We tried to reduce the decrease factor, thus causing bias towards ham classification.

We had the following result:



With decrease factor = 0.125, the false positive ratio is 4.4%, indeed less than 6.8% obtained with decrease factor = 0.5.

The penalty is 2.4% in error rate: 19.3% error comparing to 16.9%.

6.2 References

1. Notes by John McCullock
   http://mnemstudio.org/neural-networks-winnow-example-1.htm
2. Class scribe
   http://www.cs.tau.ac.il/~mansour/ml-course-12/scribe4_online.pdf

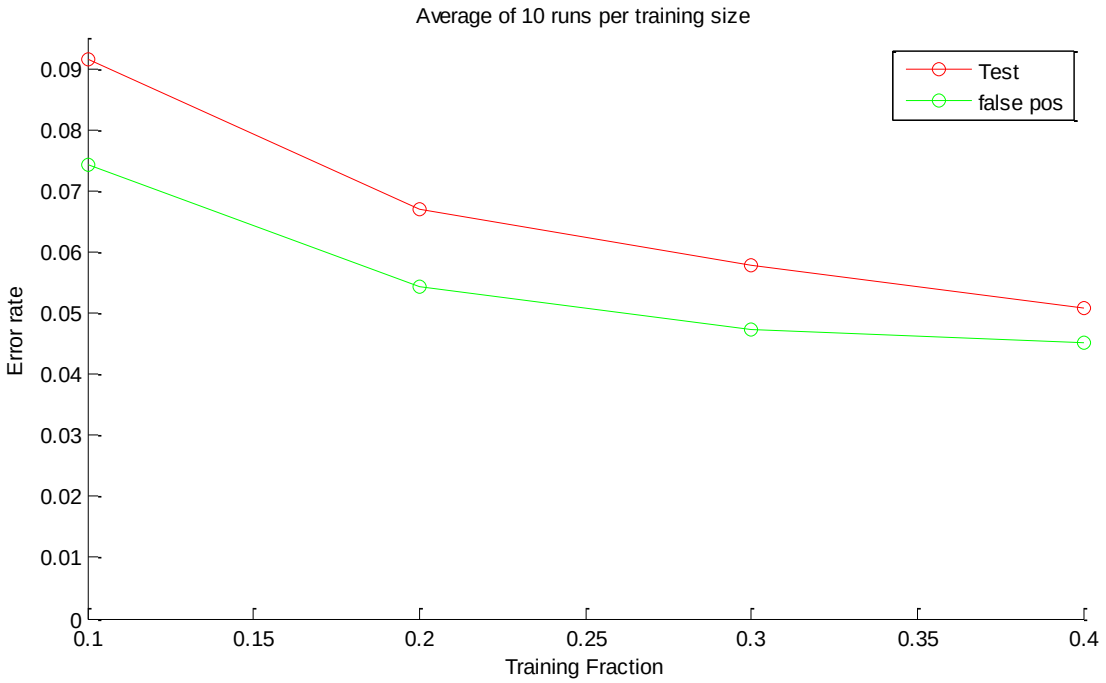
7. Support Vector Machine

We used Matlab built-in SVM function, with QP (quadratic programming) method.

We were able to test training fractions of up to 0.4, as Matlab crashed with larger training sets.

For large training sets the running time was long: with training fraction = 0.4 (about 2000 samples), the training takes about 40 minutes.

The results were very good though:



With training fraction = 0.4, we obtained 5% error rate and 4.5% false positive ratio.

It yields similar ratio of false positives and false negatives.

We tried to use kernel functions which Matlab provides but got worse results with them.

7.1 References

1. Class scribe
   http://www.cs.tau.ac.il/~mansour/ml-course-12/scribe10_svm.pdf
   http://www.cs.tau.ac.il/~mansour/ml-course-12/scribe11_SVM.pdf

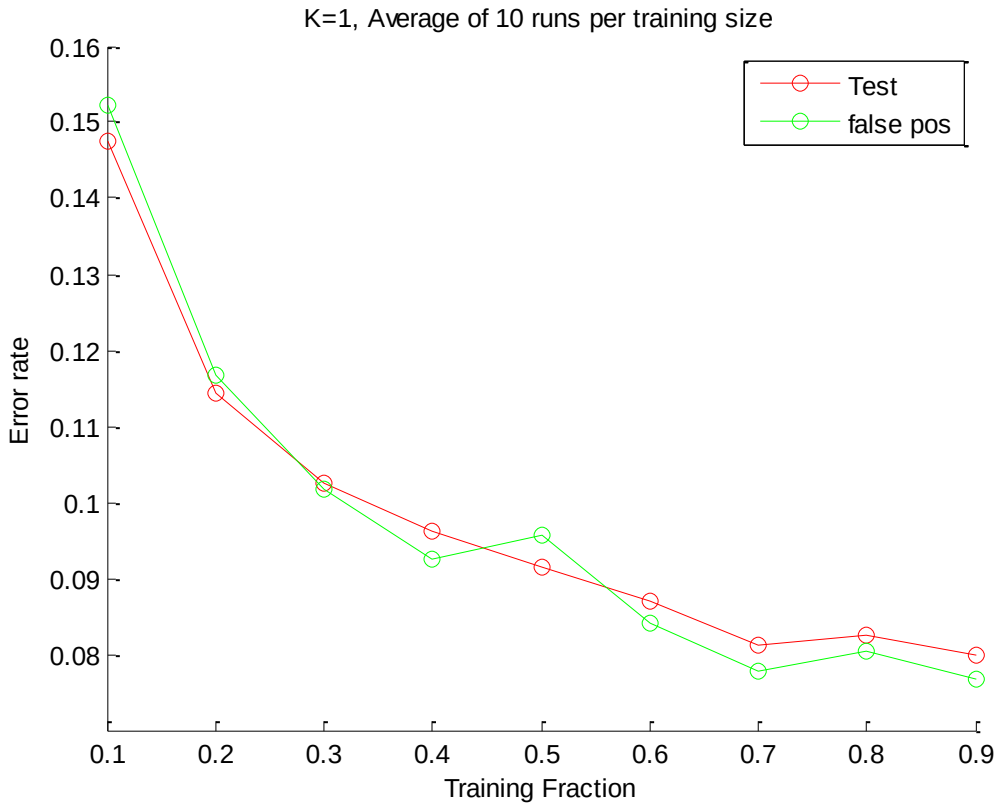
8. K Nearest Neighbors

8.1 Testing Different Training Fractions

It was interesting for us to check another algorithm which is not based on finding a separating hyperplane.

First, we set K (number of neighbors) to 1 and checked the results for different training fractions.

The following results were obtained:

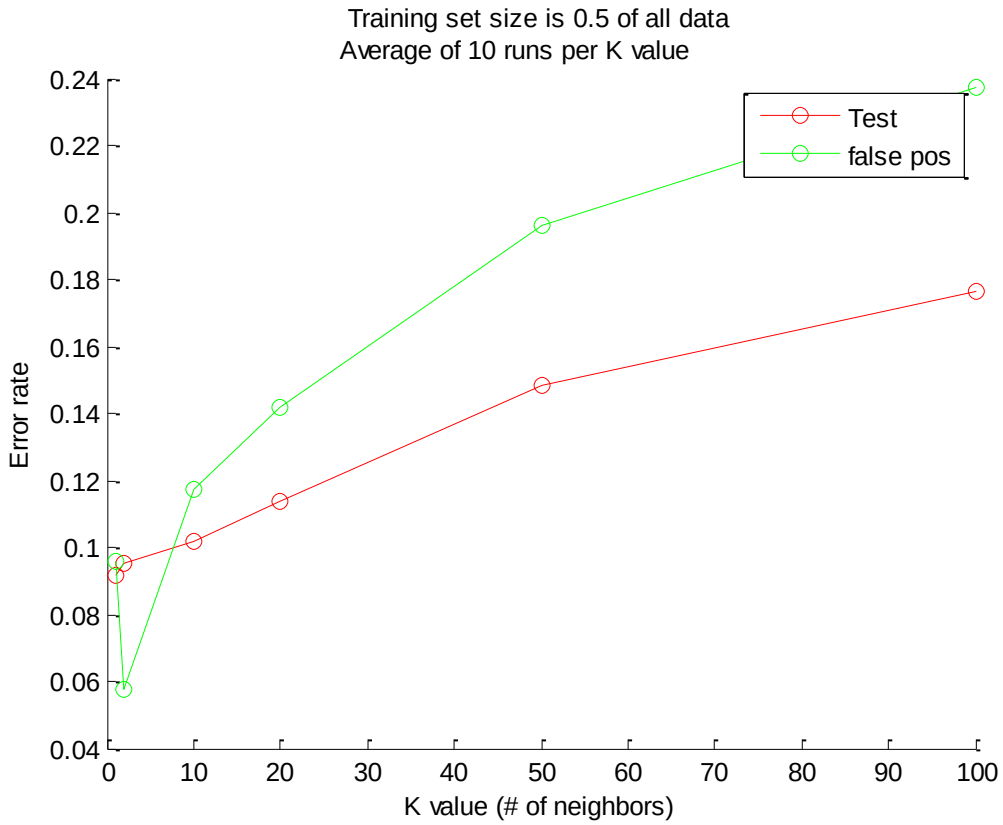


For training fraction >= 0.4 we get less than 10% error rate, which is not bad compared to the other algorithms.

8.2 Testing Different Numbers of Neighbors

The next stage was to test different values for K. We set the training fraction to 0.5 and checked the following values for K: 1, 2, 10, 20, 50, 100.

The following results were obtained:



It was interesting to see that K=1 gave the best error rate. K=2 error rate was just a bit higher, but the false positive ratio was significantly lower.

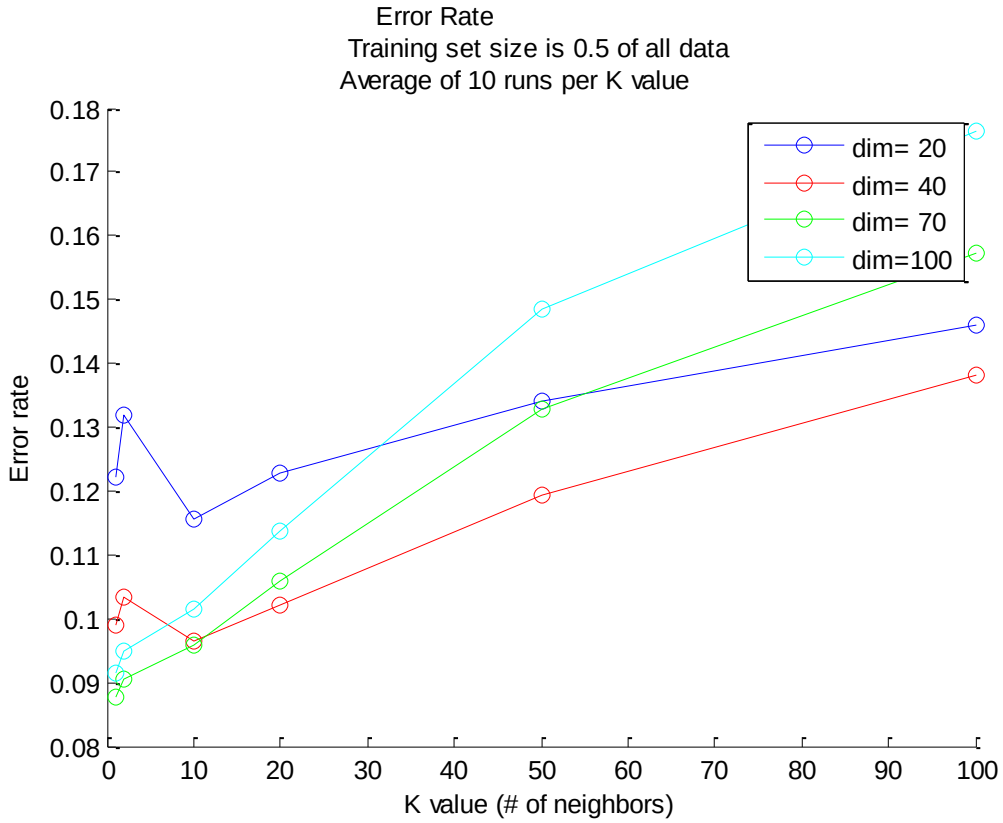
As K grew, the error rate and false positive ratio became significantly worse.

8.3 Testing Different Dimensionalities

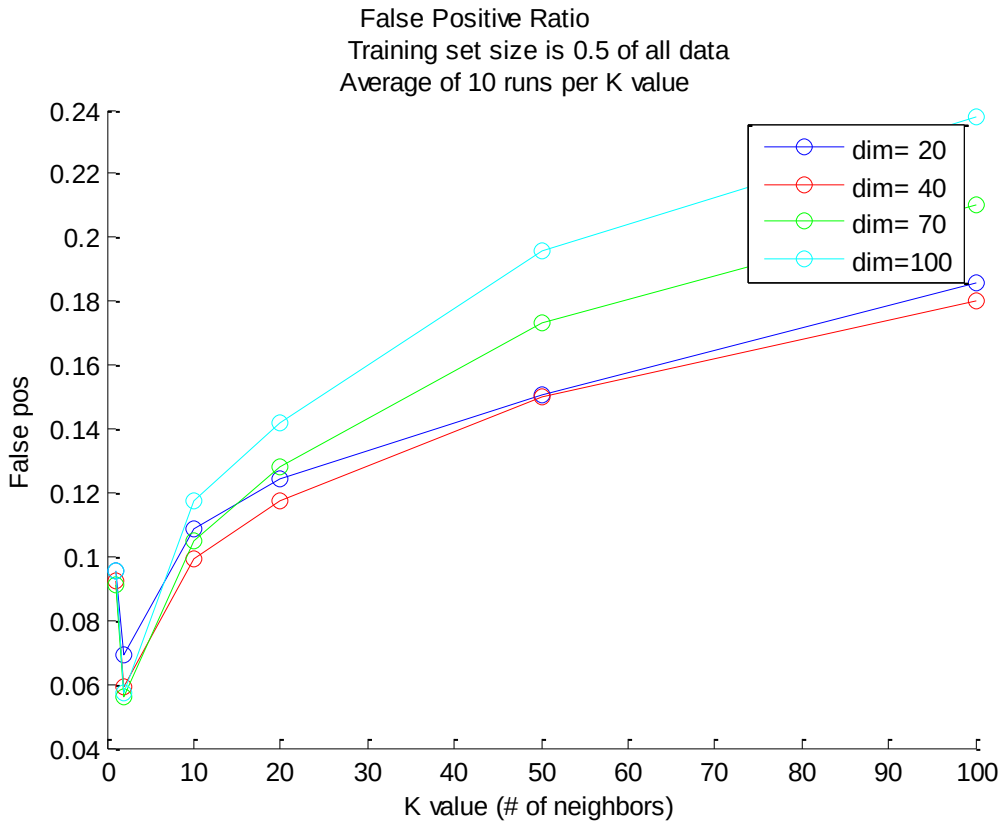
We wanted to check if the behavior shown in the previous section was due to the "curse of dimensionality" effect. We checked what happens if we reduce the dimensionality by using less features. We tested the following values for dimension: 20, 40, 70, 100 (at each case we chose the *d* features with highest rank).

The following results were obtained:

Error rate:



False positive ratio:



We see that indeed, in KNN it is better to use less than 100 features. For the larger K values the best dimension was 40. For the smaller K values the best dimension was 70.

8.4 Reducing False Positive Ratio

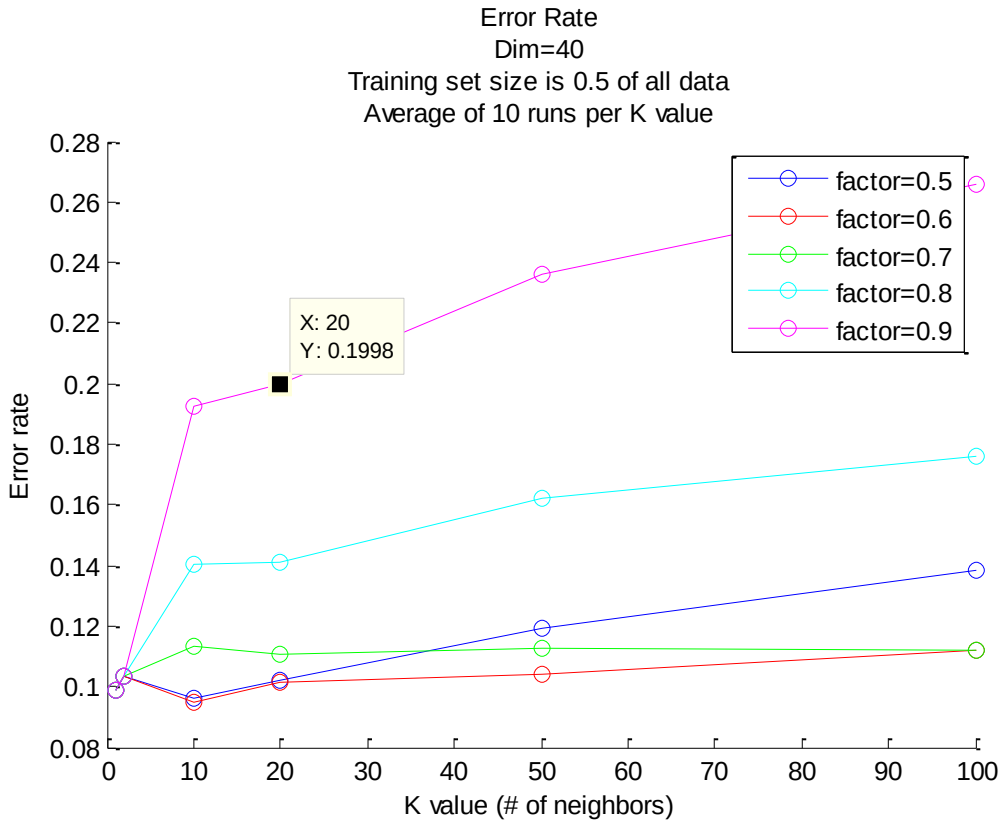
The last stage was to try reducing the false positive ratio. The idea was to classify a message as spam according to supermajority rather than regular majority. It means that a sample is classified as spam, if at least αK of its nearest neighbors are labeled as spam, where the "supermajority factor" α is between 0.5 and 1.

We checked the following supermajority factors: 0.5, 0.6, 0.7, 0.8, 0.9.

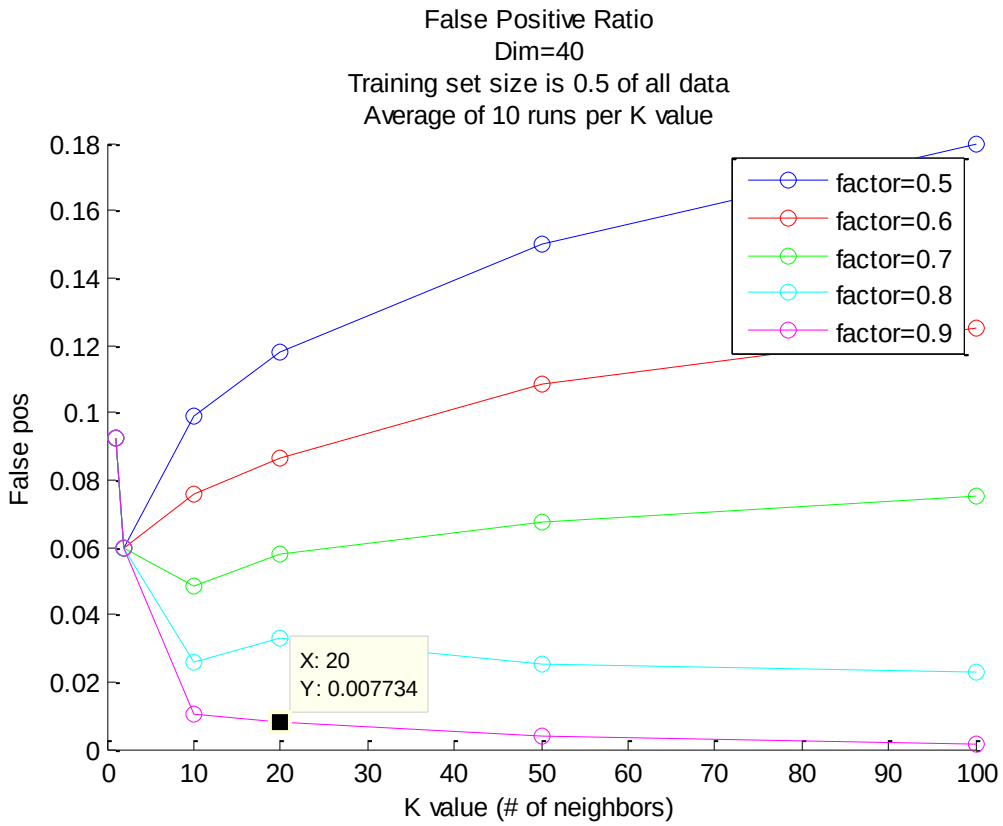
In this test we set the dimensionality (number of features) to 40.

We had the following results:

Error rate:



False positive ratio:



We see that with K=20 and supermajority factor = 0.9, we can achieve 0.77% false positive ratio. The penalty is error rate of 20%.

The lowest false positive ratio obtained in this test was 0.14%, with K=100 and supermajority factor = 0.9. However, it is not valuable, as the error rate in this case is 26.6%, very close to 29% which would be the error rate of the trivial algorithm which classifies every message as ham (29% is the proportion of spam messages in the dataset).

8.5 References

1. Class scribe

   http://www.cs.tau.ac.il/~mansour/ml-course-12/scribe7_nn.pdf

9. Conclusion

The following table shows the best results of each algorithm using its optimal settings.

Here all results are with training fraction of 0.5 except for SVM for which the training fraction is 0.4 (which was the maximum we could test).

	Minimizing Error (%)		**Minimizing False Positive Ratio (%)**	
	Error Rate	**FP Ratio**	**Error Rate**	**FP Ratio**
AdaBoost	8.7	4	NA	NA
Naïve Bayes	7.6	3	20	0.5
Perceptron	18.5	12.5	NA	NA
Winnow	16.9	6.8	19.3	4.4
SVM	5	4.5	NA	NA
KNN	8.8	9.5	20	0.77

We see that the best two algorithms are Naïve Bayes and SVM.

Considering the importance of low false positive ratio in spam filtering problem, it is understandable why Naïve Bayes is most widely used in real life.

Adding the facts that Naïve Bayes is much easier to implement and has much lower running-time, it becomes clear why Naïve Bayes makes a natural choice.

SVM and AdaBoost don't lend themselves to adjusting the balance of the error types, so it's hard to reduce their false-positive ratio.

10. Project Website and Files

The project website is at:

http://www.cs.tau.ac.il/~shaharyi/ml_final_project_2013.html

A zip file containing all project files (data & code) is available for download there.

In addition, these files are available under TAU server, at:

~shaharyi/ML/Project

Under "Project" directory there is a read-me file (Readme.txt) which contains explanation about the directory tree and about all project files.