Project Report

Tina Schuh

June 4th, 2017
IM790 Thesis Project 2

Feature Selection in Automated Detection of Aggressive Language in Online Comments

1. Initial situation

The starting point for the second part of the project was a first version of my Aggressive Language Model which included **42 features** and used scikit-learns Logistic Regression (default settings). At that point the model achieved an **average precision of 75%, recall at 70%** and an **accuracy of 74%.** This amounts to an average **f1-score of roughly 72%.**

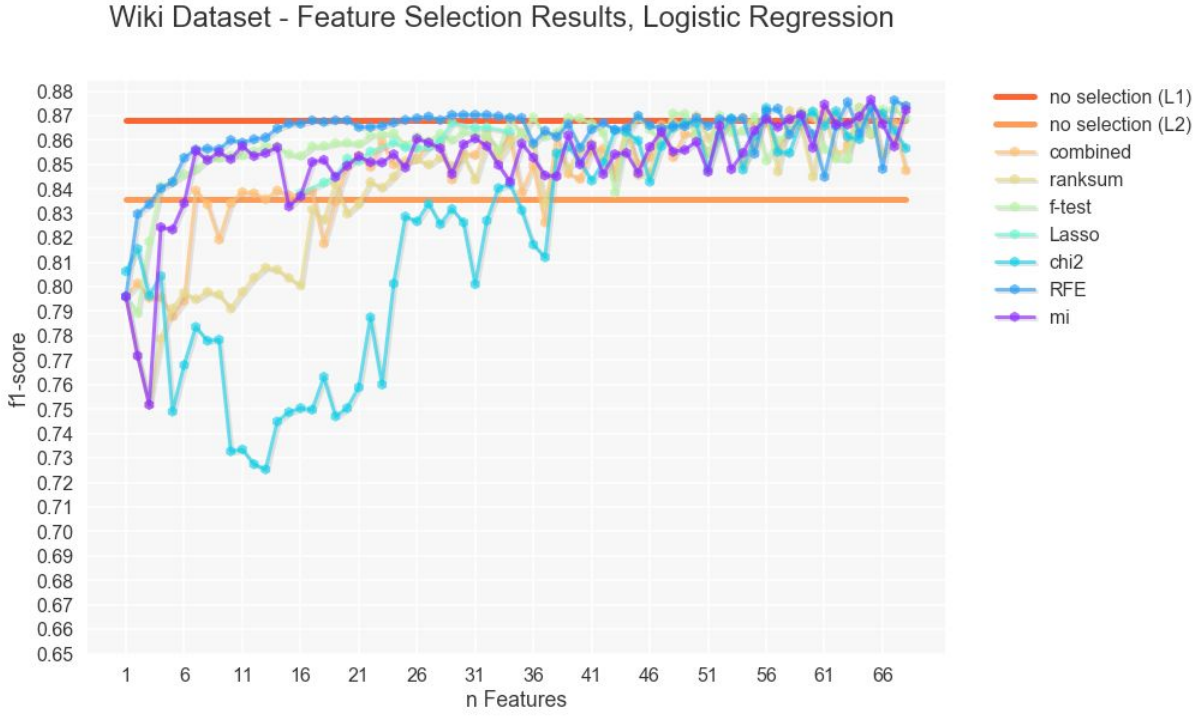
The model was based on a self developed corpus of **685 annotated comments.** I had implemented several methods for dealing with informal user generated text, namely a Text Normalization Module.

2. Accomplished Goals

- Improved and extended Text Normalization
- **26 new features** (total of 68 features)
- A self developed corpus of **1034 annotated comments;** refered to as the 'Martin Corpus'
- An **additional dataset consisting of 6987 samples** based on a newly publicized corpus (Wikipedia Talk Corpus; Label: Aggression[1])
- A clean separatation for each dataset into a **Development Set (¾)** and a **Validation set (¼)**: The Dev-Set is used for computing the feature selections and hyperparameter tuning including cross-validation (Lasso, SVM) as well as for training the classifiers. The Validation Set is used only in the final step to generate the results per classifier.
- **7 different Feature Rankings** derived from univariate feature selection methods (f-test, ranksum, chi squared, mutual information; a combination of these rankings), recursive feature elimination (RFE) and Lasso feature selection method.
- **Two different models: Logistic Regression** (using L1 penalty, except for Lasso selections which use L2 penalty) and **SVM** (Tested Kernels: Gaussian/RFB and Linear) with tuning parameters specific to each feature selection (determined by doing a gridsearch with 5-fold cross-validation)
- Improved results for my own corpus, stating the best selection (consisting of 48 features): **83% f1-score, 81% precision, 85% recall.**
  Best selection with less than 20 features: 82% f1-score, 80% precision, 84% recall (using 19 features). Details for both results: Logistic Regression (L1 penalty), f-test selection.

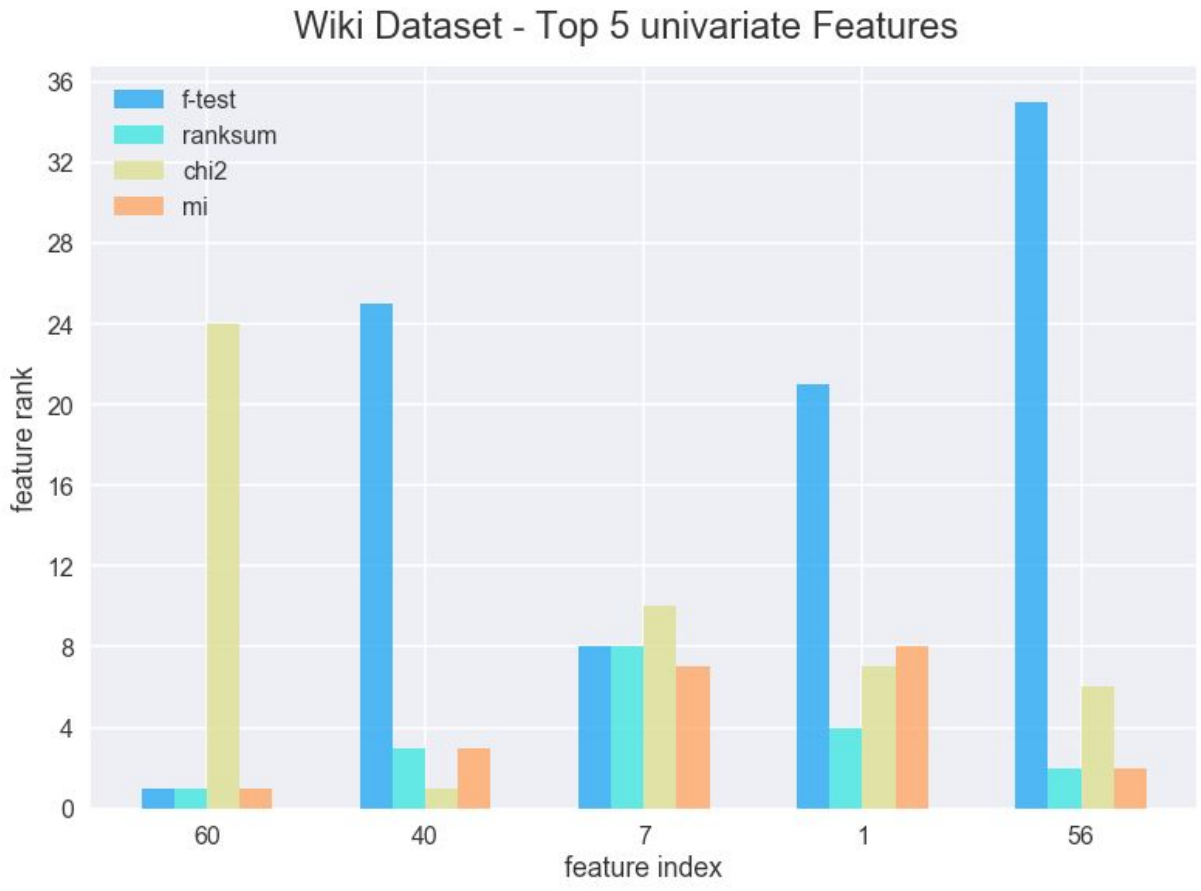
[1] The corpus had been developed for the "Wikipedia Detox project". For more information see: https://meta.wikimedia.org/wiki/Research:Detox/Data_Release

- Results for the wiki dataset, stating the best selection (consisting of 65 features, mutual information selection): **88% f1-score, 92% precision, 84% recall**.
  Best selection with less than 20 features: 87% f1-score, 92% precision, 82% recall (17 features used, RFE-selection).
  Model for both results: Logistic Regression (L1 penalty)

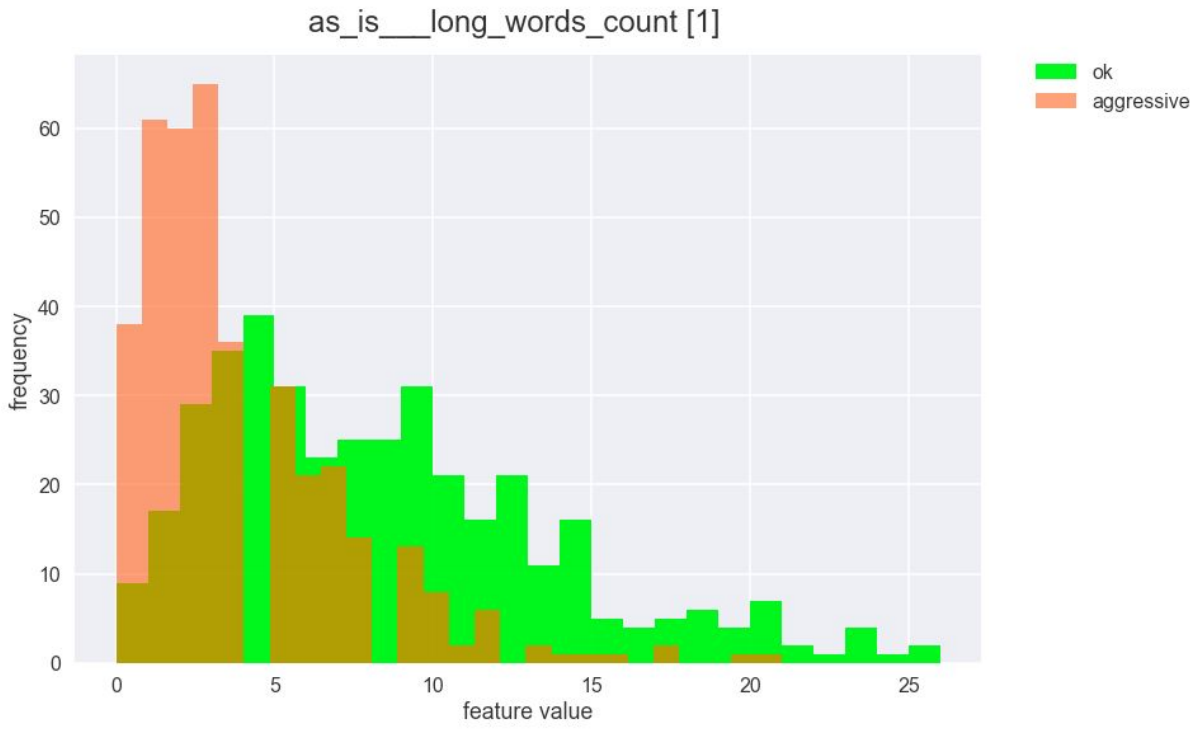


- **Demos:**
  - TextNormalization Demo versus AutoCorrection
  - Imperative Detection Demo
  - Prediction Demo: For testing a single, user defined comment
  - DIY-Feature Selection: For testing a user defined Feature Selection; displays results for both Datasets. Model: Logistic Regression (L1 penalty)
  - Display of the Feature Selection Pipeline, step by step
  - Results: Plot and explore the result DataFrames (examples provided)
  - Feature Plots: Comparison of Univariate Feature Rankings and Distribution (for Integer type Features)

Example of a Univariate Feature Ranking Comparison:



Example of a Feature Distribution (long_words_count stands for the number of words that are longer than 7 characters; this distribution is based on the Dev Set of the Martin Corpus):



3. Background Information

a. Lessons learned

- Standardize the dataset! This not only speeds up the classification process (SVM especially), the input features for the machine learning estimators are actually assumend to be scaled. Usually they are centered around zero, but if all features are non-negative the MinMaxScaler (scales the features to values between a minimum and maximum; default [0, 1]) might be a better fit.
- This was not true for my situation but if the whole dataset is available from the beginning of a classification project, it is good practice to seperate the dataset into a Development Set and a Validation Set (after shuffling), right from the start and not touch the validation set until the very end.
- Keep Unit testing! I ditched my Unit Tests at some point in the winter semester when lots of code needed to be refactored and the tests didn't match any more, and at the same time deadlines were fast approaching. After that I still kept testing new code, but not as systematically. Every now and then I stumble across a bug that could have easily been avoided. Bugger!

b. Experiences with engaged technology

The whole project is written in **Python**, which I was not very experienced in initially. I actually enjoyed learning the language, diving deeper, experimenting with concepts I had encountered in other languages (e.g. List comprehensions, Lambdas) and trying new things (e.g. Function Decorators). Great libraries, a fun and beautiful language that I will happily continue to use. **Scikit Learn**, a powerful python tool for machine learning, though I occasionally struggled with the documentation. Another powerful library is **Pandas** for data manipulation and analysis, which was especially useful for storing results, parameters and other information (in DataFrames). It took some effort to learn how to access and manipulate information in DataFrames, but once I understood I was hooked. There are probably a lot more libraries I should mention here, but most of them are so intuitive to use I don't even remember using them. I also learnt to love the art of writing **Regular Expressions**, especially after finding this fantastic visualization tool: www.debuggex.com.

c. Reasons for design decisions

I chose a modular approach and coded only 2 classes for the whole project in total. Personally I found this a lot more convenient, easier to read and easier to maintain.

d. Possible enhancements

The project is a research prototype, which means that some parts would benefit from refactoring regarding readability, maintainability.

In terms of Feature Selection I think it would be interesting to implement the following approach: Compute the best selection for less than `m` features and subsequently use this selection as the starting point for forward selection, optionally to a predifined maximum number of features: `m + n` features. I got this idea when experimenting with selections that I had compiled manually, out of curiousity. It became apparent that some features seem to complement each other well, while other combinations degrade the result. I also observed that in some cases lowly ranked features did have a positive impact when combined with the right set of features.