Undemocratic Tweets and Where to Find Them
Vol. II

Simon P. von der Maase*

February 8, 2019

*Ms. Student Political Science at the University of Copenhagen

Contents

Abstract

In this assignment I utilize a range of Data Science, Machine Learning and Deep Learning tools to create a text-classification scheme for identifying undemocratic sentiments in the tweets of American politicians. Using various n-grams, tf-idf weighting, word embeddings and dimensionality reduction I create a relatively powerful yet compact representation of the data at hand. From here I use seven Machine Learning classifiers to create a weighted voting ensemble classifier for the prediction task. The ensemble performed well on my inner training set getting an AUC-score of 0.945. Yet it underperformed somewhat when confronted with my outer test-set obtaining only 0.932 - indicating a over-fitting problem which is a challenge for an other time. Lastly I implement a number of deep neural networks, alas without beating the weighted voting ensemble.

1 Introduction

As the title of this project reveals, this is a follow-up on a previous project[1]. The goal, non-the-less, is unchanged:

> *To create an automated text-classification framework for identifying undemocratic sentiments in the tweets of American politicians.*

The approach, however, is far more advanced and systematic. Some of the improvements, in this vol. II, was already purposed in vol. I but never implemented, while others are completely novel.

The following two subsections serve to present approaches and results from both the previous and the present endeavour - thus introducing the project at hand while also highlighting some major differences between vol. I and II.

1.1 Motivation and results of vol. I

The project of identifying undemocratic sentiments amongst politicians is greatly inspired by the work of Mainwaring and Pérez-Liñán (2013). In their endeavour, the two scholars analyze the prerequisites for democratic collapse in South America. Like others before them, they find that polarization and radical positions amongst a given countries' elites, significantly increases the risk of democratic collapse. One of their more original contribution is the demonstration of how intrinsic democratic preferences in political elites can make the democratic system more robust against otherwise undermining polarization.

A central challenge of their undertaking is the construction of a reliable framework for classifying whether or not a given member of the elite has intrinsic preferences for democratic institutions. Their solution is a coding-scheme cataloging whether a politicians or otherwise central figure has, in some speech or statement, directly attacked or discredited fundamental democratic institutions. If this is indeed the case then that figure is classified as *not* having intrinsic democratic preferences. The coding-scheme itself is relatively simple, but proved to be quite reliable[2]. The real challenge lies in applying the scheme

[1] My exam project *Undemocratic Tweets - and Where to Find Them* written for the course *Advanced quantitative methods in the study of political behavior*, autumn 2017. This project and all adjacent scripts have been append the present assignment for survey if needed.

[2] The specific implementation used in vol I. can be found in the appendix, subsection 9.1

to tens of thousands of documents and reports. For this marathon Mainwaring and Pérez-Liñán employed a total of 19 research assistants (Mainwaring and Pérez-Liñán, 2013) - thus, this was no cheap undertaking.

This is where my project enters. In vol. I proposed tackling the situation using Machine Learning and automated text-classification. As such, I assessed to what extent a relatively simple Machine Learning setup could "learn" the coding-scheme and thus be used to classify political statements as either exhibiting undemocratic sentiment or not.

To keep things simple, I used twitter feeds from politicians in the American congress as well as the president in office, Donald J. Trump. As I argued in vol. I, this particular roster does properly not correspond one-to-one to some true or salient political "elite", but for proof of concept, they will do just fine[3]. From these political figures I "hand-labeled" 5099 tweets and, with a relatively simple Machine Learning model[4], I showed it was possible to get and AUC score of 0.929. Figure 1 shows the corresponding ROC curve.

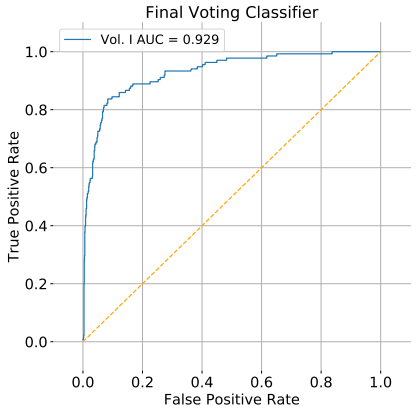

Figure 1: In vol. I, the performance was evaluated via a confusion matrix and the f1 score. However since I employ ROC curves and AUC scores in this project, I have re-assessed my previous results under this framework as to make the results more easily comparable. This did entail making some minor changes to the model such as re-specifying the voting classifier to soft voting - calculating probabilities - rather then hard voting - hard assignment. Similarly I omitted the Support Vector Classifier, since it hard assigns labels, rather then calculating a probability. All changes are specified and accounted for in in the appendix, subsection 9.2

On one hand the results show that the classification-task is indeed feasible. On the other hand, it is my assessment that the methodology still needs quite a bit of refinement if scheme is to be employed in a practical setting. I need more

[3] To identify some "true" political elite is a comprehensive endeavour in its own right.

[4] An ensemble voting model consisting of a Logistic Regression, a Random Forest classifier, a Linear Support vector classifier, a Bernoulli Naive Bayes classifier and a Stochastic Gradient Descent classifier

reliable predictions. This is where vol. II enters.

1.2 Improvements and results of vol. II

For the present project I use the same labeled data as I did in vol. I. That is not to say that the endeavour would not greatly benefit from more labeled data, but noting that hand-labeling is both trivial and time consuming, I have chosen to distribute my resources to more technically demanding tasks such as preprocessing, feature engineering and model constructions.

Disregarding the labeled data, vol. II introduces substantial enhancements on almost every level of the framework. Listing the most salient improvement, vol. II presents:

- A 3-Layered cross-validation design as opposed to 1 layer in vol. I
- A far more systematic and cross-validation based preprocessing
- The inclusion of bi-grams and tri-gram as opposed to only uni-grams
- The inclusion of a tf-idf-vectorization[5] as oppose to only a count-vectorization
- The successful implementation of skip-gram features via word-embeddings
- Systematic feature selection
- Dimensionality reduction using truncated singular value decomposition
- Implementation of an oversampling step to battle the unbalance of the data
- Weighted voting
- Feed-forward and Recurrent Neural Network

The feature specific implementation helps the AUC-score increase from 0.929 in Vol. I to 0.942. The model specific implementation further increases the AUC-score to 0.945. This is only in regard to my inner training-set[6] however.

[5] tf-idf-vectorization was presented and at one point utilized in vol. I but never used in the final model

[6] More on this design later

Utilizing the model on completely new data, it only obtains and AUC-score of 0.932. The reasons for this will be elaborated on throughout the project.

2 More Scraping

In vol. I, I scraped some 36.000 tweets from American congress members and president Donald J. Trump. As mentioned before, I labeled around 5000 tweets by hand, and as noted these are still the only labeled data I use in the present project. However, in vol. I, I also used the full set of unlabeled tweets to construct a vector space of word-embeddings using the word2vec framework (Mikolov et al., 2013). The goal was to extract skip-gram features from this vector space, which would better take into account the context of a given word, as opposed to a more conventional Bag of Word representation which strips the words of all context[7].

One of the conclusions from vol. I regarding my - rather unsuccessful - implementation of word-embeddings-based features, was that the corpus used was to small to construct a valid vector space. I also showed that the corpus needed to be rather contexts specific, thus it was of little help to implement a larger, pre-compiled, corpus not explicitly connected to American politics.

To meet these two challenges, I have revised my approach, thus now scraping 1,228,346 Tweets from American politicians and further augmenting this corpus with 1,316,217 reddit posts and comments regarding politics and news, as to ensure the corpus context specificity.

2.1 Twitter

Regarding the twitter-scrapping script I implemented a number of improvement. Not least a timer which let my handle twitter's REST API's time-specific rate limit, letting me scrape far more than the 36.000 tweets collected for vol. I. Furthermore, the scraper now takes the congress members twitter handles directly from Propublicas Congress API[8] instead of through a dataset. Naturally this makes it a lot less cumbersome to maintain the script for future endeavours.

[7] More on this topic later

[8] www.propublica.org/datastore/api/propublica-congress-api

As such, the scraping continued until either all of the given politicians tweets were scraped or the REST API's hard cap at around 3200 tweets per time-line was reached[9].

2.2 Reddit

The tweets I scraped might correctly represent the subject at hand - American politics - but they are also short and condensed[10]. To get longer text pieces, more correctly representing the relationship between words in the English language, while still mirroring the subject at hand, I turned to the two sub-reddits r/politics and r/news. From each of these I scraped the sections "hot", "top" and "controversial". "Hot" are posts that at the present moment receives the bulk of attention. "Top" are posts that have been "upvoted" the most times over all[11]. "Controversial" posts are posts that have received both a large amount of "upvotes" and "downvotes", thus splitting the community in twain.

The "hot" and "top" sections where scraped to get the most general and inclusive corpus. The section "controversial" was chosen since the project after all revolves around polarization. As such, it seemed logical to explicitly introduce text pieces related to political controversies.

For each of the two sub-reddits and the respective subsection I let the scraper run until all informations from the specific section was scraped. More information could undoubtedly have been obtained with very little effort - not least since there exist no hard limit to how many posts one can scrape[12]. Non-theless, assessing that the total number of text-pieces was of an acceptable size, I choose to stop here. More data is always better, but alas time is the scarcest resource of all[13].

As such, I have now accumulated a total of 2,544,563 new text pieces for the constructions of the word embeddings. This is nothing compared to a model train on billions of pages - e.g. all of wikipedia - but against the 36.000 tweets used in vol I, it is a big improvement.

[9] A reference to the notebook containing the revised twitter-scraper can be found in the appendix in subsection 9.3

[10] Most are from before the enlarged 280 character limit and thus only 140 characters.

[11] Or given a more specific time frame e.i. Week, moth , year ect.

[12] according to reddit's documentation

[13] Reference to the script for scrapping the various reddit sections can be found in the appendix, subsection 9.4

3 Validation Design

In this section I briefly outline the validation design in all its aspects while also commenting on why I have chosen the paths I have, and what might be done even better - given more resources - in future endeavours.

3.1 The Layers

For the project at hand I utilize a 3-layered cross-validation design. As will be explained more in-depth below, the layered design is a precaution taking to reduce over-fitting inflicted as a side product of the randomness inherent in the intermediate validation process doing preprocessing, feature selection and model optimization.

More specifically the design is as follows. Employing a holdout-cross-validation scheme, the full labeled dataset is first split into an "outer" training- and test-set (80/20). The outer test-set will be hold apart from the project until the very last evaluations. For the intermediate evaluation regarding feature and model selection I also use holdout-cross-validation and the outer train-set will thus itself be split further op in to an "inner" training- and test-set (80/20). The last third layer is utilized for a number of optimization tasks. At this layer I employ k-fold cross validation to fine tune various hyperparameters, and search for the optimal constellations of features.

The reason for this somewhat convoluted setup, is that doing the various selection and optimization tasks I will enviably be over-fitting to the test-data. Evaluation of the final models against data used in the optimization of the model will thus give biased results in favor of the model.

As such it would be even better if I had more layers such that e.g. the feature selection process and the model selection process could have each their own layers. Alas, I only have some 5000 observations, and only about 500 of these constitutes "events"[14]. More layers would spreed the data to thin. As a poor man's solution I use different splits for the feature selection and the model construction process. Same data, but different splits. This is suboptimal, but it is data efficient and it is better than doing nothing.

[14]That is; a tweet in which some politician demonstrated the lacking of intrinsic democratic preferences

3.2 The Metrics

In vol. I, I evaluated the performance of my models via the f1-score and a confusion matrix. There is nothing wrong with this approach and if the model where to be employed in a practical setting f1 might be the correct score to optimize against given the balance of this metric. In all certainty a concrete assessment regarding the trade-of between false positives and false negatives, and what consequences this might entail given the specific utilization, should be undertaken.

Regarding the present project, however, I have chosen to evaluate the results via ROC curves and AUC scores[15]. The reason for this, is that it gives me a broader view of my model's potential, and this metric is less effected by the unbalance of the data. Disregarding what exact balance between false positives and true positives might be appropriate, the ROC curve illustrates the full spectra of possibilities. This also means that for the most part, I aim to optimize in regards to the AUC score. Again this is not the most prudent path to take in an applied setting, but in this explorative phase of validation and proof-of-concept the metric is both informative and easily presented.

3.3 The Logistic Regression

In all of section 4 and section 5 I will utilize a simple logistic regression for all cross-validation purposes. This reason is simply that it is extremely fast, it has no stochastic component to consider, and as I showed in vol. I and later on in subsection 6.1 it performs remarkable well on the problem at hand.

3.4 Grid search

In tandem with the various cross-validation tasks throughout this project I will be doing a lot of grid searches. A grid search is simply to construct a grid of various hyper parameters and then surveying the results over all possible combinations of these hyper parameters. The advantage of this approach is that it is exhaustive and will find the optimal values within the given values of the grid. The disadvantage is that it is very time consuming and only covers values explicitly given by the researcher, thus there is no guaranty of identifying the

[15]Receiver Operating Characteristic curve and Area Under the Curve score

optimal global solution. I will discuss the prospect of utilizing a more intelligent approach in subsubsection 5.3.1, but for the project at hand grid searches will do.

It should be noted that I do not have the resources survey all combinations of pre-processing choices, features and estimators. I will thus take a rather naïve approach where I assume that each step in my process can be optimized with little or no regard to the processes that follow. This is no doubt a faulty assumption, but necessary if this is to be a feasible undertaking.

4 Pre-processing

In this section I will go through the pre-processing steps implemented in the project at hand. As i spent quite a bit of ink on explaining the fundamentals in vol. I, I will here be very brief regarding the basics, instead spending more time elaborating on my specific approach. I hope the reader can appreciate the trade-off.

Machine learning models only understand numbers. Thus to treat text as data we need to convert it into numbers. The conventional way is a term-document-matrix (TDM) which is simply a matrix where each row corresponds to a document and each column corresponds to one word or some combination of words; so called n-grams[16]. Most often the cells simply holds the number of times the given n-gram appears in the corresponding document. The cells might also hold a binary indicator of whether a given n-gram appears in the document or, as we shall see later, some weighted scalar. Pre-processing is the task of choosing, cleaning and formating the n-grams present in the corpus of documents. As we shall see, this includes discarding high frequency words an symbols, identifying the common stem of the words, lowering all letters, setting the n in n-gram, choosing between count-representation or a weighted scheme and more (Grimmer and Stewart, 2013). In the following section, most of these elements will be assessed and specified via cross-validation, and when I speak of cross-validation in this regard, I naturally refer to the inner test- and train set as laid out in subsection 3.1.

One important lesson to keep in mind as I proceed from here is, that - to a certain extent - we would like to minimize the feature space as much as possible

[16] more on this later

doing preprocessing and feature selection. This can be attributed to three main factors:

- It can improve the model by reducing noise
- It can improve the model by reducing over-fitting
- It speed up every step of the process

The last point might seem like something that is nice-to-have - not need-to-have, but in fact time efficient, quickly becomes crucial not least doing the various grid searches. With this in mind I proceed to the task at hand.

The first step, tokenization, is where each document - here tweets, post and comments - is split up into single words. This can be done according to white-spaces, punctuations or more complex rules. As in vol. I, I choose to implement a more elaborate and pre-constructed tokenizer from the python package NLTK[17]. I will not go into depths with the technicalities of this, but as an example it incorporates language specific features, such that when parsing English text, it will split don't into "do" and "n't" instead of "don", "'" and "t" - which would be the case if we merely unintelligibly split by whitespace and punctuation.

Next, is formatting the words, reducing them to some canonical form such that e.g. city and cities are registration as the same word. As in vol. I, I have chosen lemmatizing over stemming, which provides the grammatical base-form rather then some crude pseudo-base. I also lowered all characters and removed symbols and punctuations. This is not to say that punctuation and symbols necessarily does not hold any information, but cross validation showed that appending various symbol-features after pre-processing only introduced noise[18]. I have also implemented a list of stop-words - that is, very common words that rarely holds any meaning. This is a very simple list, also supplied by the nltk library[19] Contrary to vol. I, I choose not trust the stop-word list unconditional, but cross-validation showed that indeed it did strengthen my predictions[20].

Non-the-less, we can do better than a short pre-made list. I created a grid for testing the benefits of removing low or high frequency words according to

[17]Natural Language Tool Kit

[18]See the appendix, subsection 9.5

[19]A list of both all removed words and symbols can be found in subsection 9.6

[20]See the appendix, subsection 9.5

some thresholds t_L and t_H. Naturally I only include the corpus given by the inner train set.

To illustrate the concept, Figure 2 and Figure 3 display the potential for feature-reduction if we where to set $t_L = 3$ and $t_H = 400$ tweets.

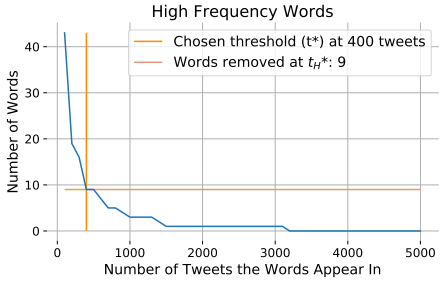

Figure 2:

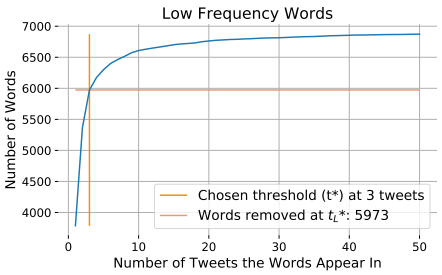

Figure 3:

We see that a lot of words could be omitted by some relatively harmless t_L. On the other hand, it seems we have to set t_H quite low in order to get any feature reduction at all. Surprisingly, doing a quite extinctive grid search over a range of t_L and t_H, I find that there is no gain to have, by removing low frequency words at all. On the other hand there are some gains to be found be setting a t_H. More specifically, I find that the optimal $t_H = 500$[21]. It should be noted that this threshold is only valid on datasets of same size and composition as the one worked on in this case. The approach of identify $t*$, however, is valid whatever data it is presented.

As such, we are ready to move to the creation and specification of the TDM and feature engineering more generally.

5 The Features

The feature engineering part of this project, which will be presented in the following sections, is a far more ambitious endeavour then in vol. I. However, it does need to fit on the same amount of pages. Thus, as in the previous sections, I will only spend a minimum amount of ink on the fundamentals[22].

[21] A reference to the notebook containing the grid search can be found in the appendix, subsection 9.5

[22] More elaborate introductions can be found in vol. I.

5.1 N-grams, Tf-idf and the Term-Document Matrix

In vol. I. I settled for a count-vectorized TDM solely of uni-grams; all features corresponded to one word, and the cells contained the count of the given word in the given tweet. Here, in vol. 2, I have included both bi-grams and tri-grams. Thats is, all occurring pairs and triplets of words in the inner training set. Furthermore, this is implemented in both a count-vectorized setting and a Term-Frequency/Inverse-Document-Frequency (tf-idf) setting. Tf-idf is simply a weighing scheme, which negates the general impotence of words if it appears several times in many documents, but increases the importance of words which appears a lot in only some or few documents[23].

All in all this produces 6 TDM. Utilizing the inner test-set and a simple logistic regression for cross-validation I obtain the results presented in Figure 4 and Figure 5.

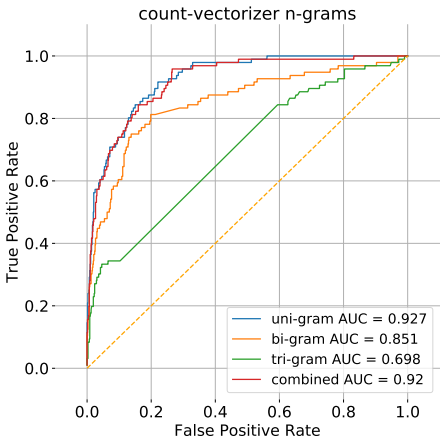

Figure 4:

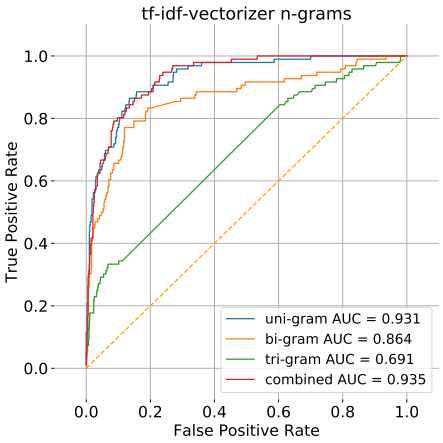

Figure 5:

Comparing these 6 implementations, the highest AUC at 0.935 is produced by the combined tf-idf TDM - if only by a margin. I have also tested a TDM consisting of both the count- and the tf-idf vectorization, but with and AUC-score of 0.921 that did not fare any better[24]. Notably then, with only a simple logistic regression I already get better results than I did with the final voting classifier in vol. I; all due to more systematic pre-processing and the inclusion of bi- and tri-grams, though admittedly the inclusion of the large n-grams only contributes marginally to the predictions. As Grimmer and Stewart (2013) notes, this is often the case.

[23] A notebook containing the implementation of all TDM in subsection 9.5

[24] See appendix, subsection 9.7

Including both uni-grams, bi-grams and tri-grams also comes with a price; a relatively large feature space of 75,852 features. Even more, this is bound to rise dramatically if I where to utilize more data. As such, there are the potential of reducing over-fitting and noise while gaining speed by reducing the number of features. I could of course just move forward including only uni-grams which is often a viable approach according Grimmer & Stewart, but there are more systemic approaches which allows my to retain what useful features might be in the bi- and tri-grams while also getting rid of useless uni-grams. Thus I move forward only with the three[25] TDM, containing all generated n-grams.

5.2 Feature reduction

As mentioned I will seek to reduce the dimensionality of the feature space in a number of ways[26]. In this section I use the most direct tools; simply removing all features with a coefficients of 0 given a logistic regression and finding the optimal feature combination using recursive feature elimination.

The first step is simple. Running a logistic regression on the inner train set and remove all features that obtains the coefficient 0. This is crude but effective and fast.

The second step is a bit more convoluted. I utilize a "third" cross validation layer, simply meaning, that to find the optimal combination of features I run a 3-fold cross validation on the inner train-set. I start with a model containing all features and then recursively eliminates irrelevant features still based on their coefficients. It should be noted that this search is not truly exhaustive - once a feature is omitted, it never reenters the cycle - thus we are not guarantied the most optimal combination of features. Furthermore, the fact that I am only evaluating on the basis of the coefficients also makes this step rather crude. On the flip side, both of these methods are very fast, and as seen in Table 1 they do bring the feature size down quite dramatically.

As such, this brute approach reduces the number of features quite dramati-

[25] The Count TDM with all n-grams, the tf-idf TDM with all n-grams and the aggregated TDM consisting of both

[26] The tools I could employ here are legion, and the order in which I use these tools have an essential part to play. However I do, neither have the space nor the time to conduct an exhaustive evaluation of all tools nor all sequences or combinations possible with the few tools selected. I will argue why I have chosen the road I have as we go along, but know that other combinations or sequences could have yielded to other, perhaps better, results. Only cross validation can tell, and cross validation takes time.

	Original	1. step	2.step
All n-grams count TDM			
Number of features	75,852	17,222	6619
AUC-score	0.92	0.921	0.913
All n-grams tf-idf TDM			
Number of features	75,852	14,125	5858
AUC-score	0.935	0.934	0.931
Combined TDM			
Number of features	151,704	35,451	10,429
AUC-score	0.921	0.921	0.917

Table 1:

cally. Unfortunately we also see that the reduction in most cases diminish the prediction power somewhat. For example the ti-idf TDM, we go from an AUC-score of 0.935, then 0.934, then 0.931. This is only a marginal change taken into account that we reduced the feature space with roughly 70.000, leaving us with only 5858 features or some 8% of the original space. It it also worth remembering that the combined tf-idf TDM was chosen on account of having the single highest AUC-score; some amount of randomness is bound to interfere with this selection process and thus picking the model on account of this randomness leads to over-fitting. A testimony to this effect is the fact that the combined tf-idf TDM is the only one of the TDM which suffers from the first step. As mentioned the solutions here would be to have a new layer to test on. Alas the amount of data hardly permits this.

In subsection 5.4 I will use a more intelligent way of compression the feature space further. Thus I will not yet be passing judgment on which of these TDM's is the "best". Until then, I will leave the TDM's and create a feature space through an altogether different approach; word-embeddings.

5.3 Word-embeddings

The TDM created in the preceding sections all had the same basic architecture; columns corresponding to words or groups of words and rows corresponding the text pieces. The approach taken in this section is a less obvious representations of words, but a mighty effective one. In vol. I, I spend a great deal of time explaining the concept of word-embeddings, alas I never succeed in creating any useful features for my prediction problem. In the spirit of the project, I shall

here spend less time illuminating the concept, but more time explaining how I now got it working.

The specific approach was pioneered by Mikolov et al. (2013) who created the word2vec algorithm, also utilized in the project at hand. It was further championed by Marco Baroni, Georgiana Dinu (2014) who showed that in almost all cases the approach was superior to traditional count based models.

5.3.1 Creating the embeddings and the features

Simply put, one employs an artificial neural network to create a K-dimensional vector space where all words in a given corpus are embedded. Similar words will appear close[27] in the vector space, while very different words will be far apart. Which words are deemed to be similar or connected is of course greatly affected by which corpus one feeds the neural network. As I showed in vol. I, a contemporary politic-specific vectors space might asses that "fake" and "media" are close while are more general vector space asses that "fake" and "ass" are rather close. This is why I need a lot of political-tempered text-pieces; to correctly map the relationship between various words given an explicitly political context.

The framework is remarkable in that it allows meaningful arithmetic operations between the words. The canonical example is that having trained the word-embeddings on the works of Shakespeare, you will be able to calculate that $king - man + women = queen$. This ability is what I utilize to create the actual features for my prediction problem. As mentioned, each word is represented by a vector of K-length. Each test-piece can thus also be represented by a K-length vector, by taking the k specific means of all relevant word-vectors; that is all words appearing in the text piece:

$$text_v = \frac{1}{n} \sum_{i=1}^{n} word_{vn}$$

Where n is the number of words in the specific text piece v. The number of features I extract for each text piece is thus equal to K; the number of vectors that represent each word. As such, a big mistake I made in vol. was also the average, the text vector reducing it from K-length to 1. Back then, I did reckon

[27] closer measured here by the cosine proximity

this was too crude an operation to retain the needed information, and indeed it was. As we shall see, keeping the dimensionality at K made all the difference.

That leads to the task of specifying K, along a number of other hyper parameters. These includes the size of the window of adjacent words to consider when creating the embeddings, threshold of dismissing of infrequent words, weighted penalty for very frequent words. The keen reader will have guessed that the grid search and cross-validation is the way forward.

Firstly, before feeding the network with the text pieces, they are pre-processed completely according to the procedure in section 4. Then for each hyper-parameter I define a reasonable range of values constitute to the grid which is then exhaustively searched. For each combination of hyper-parameters, a word-embedding model is created and the K feature extracted for the inner train- and test-set. A model is constructed and tested and the results reported. I then revised my grid - the range of respective values assessed for the various hyper-parameters - zooming ind and extended as needed.

To carry out the operation in practice, I created a loop in which, for each iteration a word-embedding model was created, using the word2vec framework and all 2.5 million unlabeled text pieces. From the respective word-embedding I extracted features for the inner training and test-set. With a logistic regression and the inner train-set, a model was created which was evaluated by the inner test-set using the AUC-score. All in all, with the intermediate refinements, the loop ran for 409 iterations and estimated the same amount for embedding-based feature set doing the course of 34.5 hours[28]. Table 2 displays the final chosen hyper parameters.

Vector size	450
Context window	10
Low frequency threshold	40
High frequency penalty	0.0001

Table 2:

Thus the length of the word vectors $K = 450$. Our model considers a context window of 10 words; excludes words that appears in less then 40 out of the 2.5 millions text pieces and the threshold for when higher-frequency words are to be down-sampled is 0.0001.

As noted these settings should *not* be used as specific guides for similar

[28]The notebook containing the script for this process can be found in the appendix, subsection 9.8

projects. The settings work well for the data at hand, but there is no guaranty that this will be the case in other settings. The approach, however, is valid.

There are a couple of drawbacks regarding the approach, which will be amended in future work. A lot of time could be saved by incorporating some Markov chain Monte Carlo to effectively explore the grid and finding the optimal values. No doubt this would also lead to even better results. An intriguing prospect for another time.

5.3.2 Testing the features

In this subsection, having created 450 new features, I first show the prediction power of these features alone, and then in cohort with all 9 models presented in Table 1.

Looking at Figure 6, the results are impressive. Admittedly, the AUC-score is lower than with the TDM's, but this result is obtained via only 450 features.

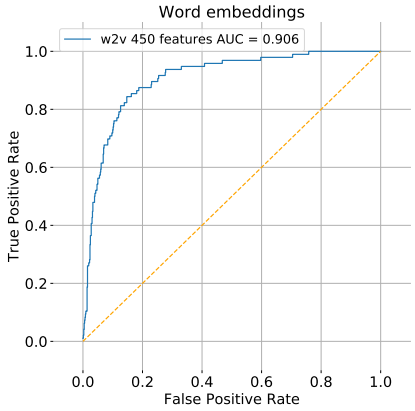

Figure 6:

Even more, as presented in Table 3 combining the embeddings-based features with the constructed TDM, in most cases brings new relevant information to the model and thus increases the respective AUC-scores.

As seen, the the full tf-idf TDM is now matched. Interesting by the 1. step the count TDM now obtain the same AUC score as the full tf-idf TDM, but with "only" 17,672 features against 75,852.

	AUC	w2v incl.	Feature count
Only w2v features	0.906	yes	450
Count TDM original	0.92	no	75,852
Count TDM 1. step	0.921	no	17,222
Count TDM 2. step	0.913	no	6919
Count TDM original incl. w2v	0.934	yes	76,302
Count TDM 1. step incl. w2v	0.935	yes	17,672
Count TDM 2. step incl. w2v	0.93	yes	7069
tf-idf TDM original	0.935	no	75,852
tf-idf TDM 1. step	0.934	no	14,125
tf-idf TDM 2. step	0.931	no	5858
tf-idf TDM original incl. w2v	0.932	yes	76,302
tf-idf TDM 1. step incl. w2v	0.933	yes	14,575
tf-idf TDM 2. step incl. w2v	0.929	yes	6308
Combined TDM original	0.921	no	151,704
Combined TDM 1. step	0.921	no	35,451
Combined TDM 2. step	0.917	no	10,429
Combined TDM original incl. w2v	0.935	yes	152,154
Combined TDM 1. step incl. w2v	0.933	yes	35,901
Combined TDM 2. step incl. w2v	0.933	yes	10,879

Table 3:

5.4 Singular Value Decomposition

In this section I utilize truncated singular values decomposition to reduce the dimensionality of the given feature spaces. I will explore the potential of this approach regarding to all 19 feature spaces displayed in Table 3.

Truncated singular values decomposition (TSVD) is closely related to Principal Component Analysis (PCA), but can be used on sparse matrices[29] which make the operation much faster.

[29] Briefly noted a sparse matrix is simply a representation where instead of having a lot of cells either containing zeros or some other number, you only keep none-zero numbers which are then accompanied by the cells coordinate. Implicitly all coordinates not accounted for, are then interpreted as zeros. In large matrices with a lot of zeros - a sparse matrix - this is a far more efficient representation than the conventional dense matrices. Coincidently, in large matrices the majority of zero's is exactly what the TDM's are. As such I have already handled the TDM's as sparse up till now and I find no reason to change this for the present operation.

Without going too much into the technicalities, The TSVD and PCA alike sorts feature space according to the latent dimensions with the highest variation. Theoretically one can then get as many new TSVD features as original features, but the point is that most of the variation - that is the information - in the original feature-set will be captured in some subset of the TSVD features. As we shall see further down, it is easy to specify how much variation of the original feature space one wants to retain, and from this point infer how many TSVD features to retain. It should be noted, however, that this is an unsupervised approach, and thus there is no guaranty that the variation retained will relate to dimensions deemed most salient by the research-question at hand. Only cross validation will tell.

Regarding the decision as to how much variation to retain, cross validation would be optimal. Alas the time-cost of running this operation is simply to great in this case. As such - after some trial and error - I choose to retain 90% of the variation in various TDM. I then run the operation on all the TDM's and asses the performance and dimensionality of each reduced set. Three of these stand out[30]; The reduced original tf-idf TDM with and AUC-score of 0.936 at 2627 features, the reduced 1. step count TDM with w2v-features gets AUC-score of 0.936 at 1190 features and surprisingly also the reduced 1. step combined TDM with w2v-features obtains an AUC-score of 0.937 at 1422 features. As an improvised manual pseudo grid search, I fiddle around with the specific number of features for each set to see what difference it might do. I find that I can go down to 1400 features for the reduced 1. step combined TDM, while still retaining an AUC-score of 0.937; while increasing or decreasing the two other TDM's never benefits the performance. As such, in regards to the predictions endeavour, I move forward with the 1. step combined TDM with w2v-features reduced to 1400 features through TSVD. All procedures used here are thus now employed on the outer training- and test set.

5.5 Final note on oversampling and standardization

To counter the balance of the data, I implemented both various forms of oversampling and undersampling, but this did not help the results. I also applied both standardizing and normalizing the features without any benefits.

[30] A table with all the relevant results have been judge to cumbersome to fit into the main text, but can be found in the appendix, subsection 9.9.

6 Prediction

I now move forward to the prediction task at hand. As teased, I also need an inner train- and test set here. Instead of using the same split as in the previous sections, I split the Outer training set according to new lines. As mentioned, this is a poor-mans attempt to prevent over-fitting, but a completely fresh layer would be the right way to go in an applied setting with more hand-labeled data than in this case.

To set a benchmark I have used the voting ensemble classifier constructed in vol. I on the new data. Figure 7 shows the results.

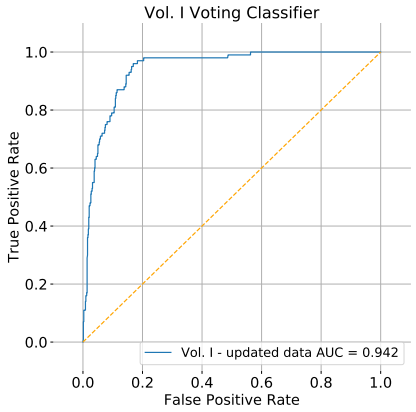

Figure 7:

Compared to the initial benchmark shown in Figure 1 I have been able to increase the AUC-score from 0.929 to 0.942 through only my handling of the data. No doubt this could be further increased given time and resources.

For the prediction task I have augmented the voting ensemble classifier quite bit, with more classifiers, expanded grid search, performance specific weights and cross validation in order to determine which classifiers to include and at what weight.

6.1 Weighted Voting ensemble

As in vol. I, I create a voting ensemble of a number of different machine learning classifiers, 7 instead of 5. I optimize various hyperparameters through grid search and cross validation - k-fold cross validation on a third layer, where $k = 8$.

A overview of the classifiers used along the respective ROC curves and and AUC-scores can be found in Figure 8. The number of parameters and the size of the grid, is somewhat more comprehensive, than in vol. I, but the parameters I am optimizing still mostly concerns regularization to counter over-fitting[31], and weighting to counter the unbalance[32]. A few parameters of a more general nature are also tuned this time. For the random forest's impurity gain, the grid search picked gini over entropy; for the Multi-layered Perceptron, the tahn activation function is chosen over identity, relu and logistic; and for the Adaboost classifier, the base estimator is chosen to be a Logistic Regression over Decision Trees[33]. I should note that all these are optimized separately. If optimized on account of what they contributed with for the aggregated voting classifier, one might encounter different results. However, this would also increase the number of models to train exponential. For the grid search I trained a total of 2211 models. If I where to include the conditional evaluation of the each of the models I would have to train some 1.04×10^{12} models all in all[34]. The naïve, if faulty, assumptions regarding unconditionality, thus serves to make the endeavour feasible. Retuning to the matter at hand Figure 8 presents the optimized classifiers.

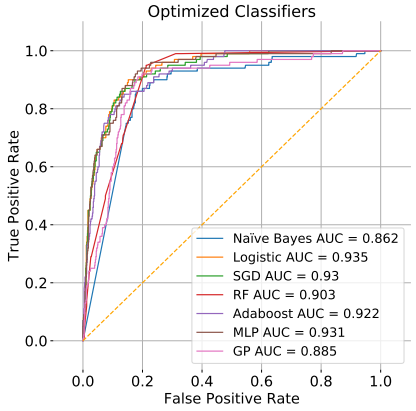

Figure 8:

According to Figure 8, most of the classifiers do quite good, but none as good as the voting ensemble from vol. I. Naïvely combining all with weights = 1, as in vol. I, I get the same result; and AUC-score of 0.942. Luckily, I have one more trick up my sleeve. There are no reason for us to trust the worst classifiers as

[31] e.g. the alpha parameter

[32] e.g. the class-weight parameter

[33] A reference to the notebook containing the optimization procedure can be found in the appendix, subsection 9.10

[34] $1520 \times 16 \times 435 \times 32 \times 16 \times 192$ rather then only $1520 + 16 + 435 + 32 + 16 + 192$

much as the best classifiers. On the other hand if a good classifier is lone in its assertion, we might be better off trusting the combined wisdom of the sub-par classifiers. The challenge is to determine the optimal weight for each classifier. Again, the dimensionality quickly makes comprehensive grid search somewhat intractable. Instead, I employ a number of less exhaustive approaches; a binary grid search, Adaboost optimizing, and simple rank.

The binary grid search simply consists of an exhaustive grid search, where the models can take the weight w where $w \in \{0, 1\}$. Effectively this search does not find weights as such, but rather sorts out the classifiers impairing the power of the voting ensemble.

The Adaboost optimization algorithm is normally used in sequence to determined the different weights of a number of identical classifiers where for each subsequent classifier the observations are weighted accordingly to how hard they are to classify. Thus, the procedure being rather simple, it is also a obvious choice given the problem at hand[35].

	Used in Vol. I.	Tested for Vol. II	Weight Vol. II
Naïve Bayes	yes	yes	1
Logistic Regression	yes	yes	6
Linear Support Vector	yes	no	-
SGD	yes	yes	4
Random Forest	yes	yes	2
Ababoost(Logistic)	no	yes	3
Multi-layer Perceptron	no	yes	5
Gaussian Process	no	yes	0
Total	5	7	

Table 4: The classifiers tested for vol. II are those run through a grid search and tested as a part of the final voting classifier. The Weight is the weight assigned on the basis of the applied boosting algorithm, where weight = 0 effectively omits the classifier. Note that a number of weight-configurations give the similar results when rounded down to three decimals. In the notebook the scores are presented down to five decimals and here we see that this configuration the best - even if the difference is marginal

Both of these weighting schemes were able to boost the predictions to 0.943, which is no impressive gain. As such, I tried a number of more or less improvised schemes[36]. Remarkable, what turned out to obtain the best AUC-sore of 0.945

[35] A reference to the notebook containing my implementation can be found in the appendix, subsection 9.10

[36] such as using the normalized AUC-score as weights as well as various combinations of the

was simply ranking the classifiers according to performance with weights from 0 to 6. That being said, for future endeavours a less arbitrary scheme will be found and utilized. The classifiers and their assigned weights are presented in Table 4.

The results from the final weighted voting ensemble is presented in Figure 9. Of course this is in regard to the inner test-set. Utilizing the outer-test set for the first time we see a somewhat less impressive results in Figure 10. The reason for the decline in AUC-score from 0.945 to 0.932 is due to over-fitting. As I warned in section 3 this is the danger of not having more layers and thus more labeled data.

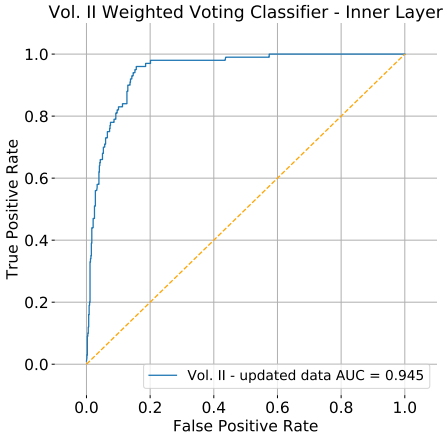

Figure 9:

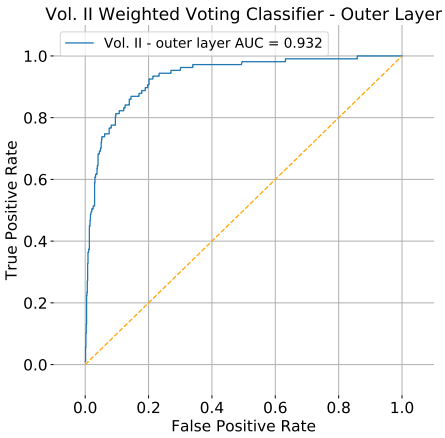

Figure 10:

Using a more intelligent weighting scheme than in vol. I, I increased the auc-score from 0.942 to 0.945 regarding the inner test-set. To be fair, this is not a lot, and it is indeed less than the gains achieved in the feature engineering part of the endeavour. Furthermore testing the model on the outer test-set revealed that at least some over-fitting caused the "gain", which in turn might have decreased the final results.

On the technical side, there might be further steps to be taken to win some marginal AUC-points, but if we are truly to see improvement - without resorting to more manual hand-labeling - we need to move beyond traditional machine learning and into the depth of peed learning. The following section will very briefly discuss what deep learning methods I have dappled with, regarding the problem at hand.

approaches

6.2 Deep Neural Network - and the way forward

For this project I implemented three types of artificial neural networks; a conventional feed forward network; a simple Recurrent networks (RNN) and a Long Short-Term Memory networks (LSTM). There are neither room nor the need for a comprehensive explanation of neural networks at this stage - but a few points will serve as justification for the inclusion of these models.

The feed forward network is implemented primarily as a baseline. It is rather similar to the multi-layered perceptron implemented as part of weighted voting ensemble. With an AUC-score around[37] 0.921 it is comparable to individual classifiers presented in last section[38]. No doubt, it could be optimized further but at its core it introduces no real novelty.

More interesting is the RNN and the LSTM. These are models which are particularly suited to problems where information can be found in the sequence preceding or following element of interest. That would be the case in time series, but indeed also when working with text and language. Note the following sentence:

> I visited $word_a$ last year. Most people there speaks $word_b$.

Trying to guess $word_b$ only on account of the two or three words preceding it, is pretty hard. Even knowing the vector of $word_b$ given some relevant embedding can be of little help. Knowing that $word_a$ is *France* makes all the difference. Thus, we want a model which can remember all elements from our text pieces. Without getting technical, both RNN and LSTM achieves this by feeding our network with our features, while also "refeeding" the model with its own output(Géron, 2017, 381-383).

Admittedly my results are underwhelming. Getting both RNN and LSTM to perform no better than the worst of the individual classifiers introduced in last section; respectively with and AUC-score around 0.89 and 0,86. And this is after what can only be described as a very extensive use of very precious time. I have subsequently come to the conclusion, that while my feature engineering might have improved the Machine Learning results, it has robbed the data of crucial information which could have been used in a Deep Learning setting. One

[37] Again using parallel processing to speed up the process makes the network inherently indeterministic.

[38] A reference the notebook containing the implementation and results can be found in the appendix, subsection 9.11

model's noise is an other model's treasure. In future projects I will keep this lesson in mind. Furthermore, since neural networks are able to conduct of some modest feature engineering automatically, I might be able to save some time in the process - time still being the most valuable resource.

7 Conclusion

This project was a follow-up to a previous project I wrote regarding the classification of tweets exhibiting undemocratic sentiments - specifically tweets from American politicians. In the previous project I showed how to obtain an AUC-score of 0.929 given rather simple Machine Learning techniques.

In the project at hand I have shown how I was able to increase the AUC-score from 0.929 to 0.942 via more extensive and advanced feature engineering and further from 0.942 to 0,945 by adding more classifiers to the voting scheme and weighting these classifiers in my voting ensemble, according to their performance rank. Most of the gain here must be attributed to comprehensive cross validation, systematic grid searches and the implementation of embedding-based features.

These results are, however, according to my inner test-set, which was what I recurrently optimized against. Tested on completely foreign data - the outer test-set - the model only managed an AUC-score of 0.932. This indeed underpins that I must have over-fitted the model to my inner test-set doing the optimization process. A problem only solved by more layers and more cross validation - two solutions both requiring more labeled data.

I also advanced beyond conventional Machine Learning with Artificial Neural Networks, more specifically Feed Forward networks, Recurrent Networks, and Long short-term memory networks. However, the impressive results will have to wait for my next endeavour since the networks constructed here, did no better than the Weighted Voting ensemble constructed out of conventional Machine Learning classifiers.

This concludes Vol. II

8 Bibliography

References

Géron, A. (2017). Hands-on machine learning with Scikit-Learn and TensorFlow concepts, tools, and techniques to build intelligentsystems.

Grimmer, J. and Stewart, B. M. (2013). Text as data: The promise and pitfalls of automatic content analysis methods for political texts. *Political Analysis*, 21(03):267–297.

Mainwaring, S. and Pérez-Liñán, A. (2013). *Democracies and Dictatorships in Latin America: Emergence, Survival, and Fall*. Democracies and Dictatorships in Latin America: Emergence, Survival, and Fall. Cambridge University Press.

Marco Baroni, Georgiana Dinu, G. K. (2014). Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. *52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014 - Proceedings of the Conference*, 1:238–247.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

9 Appendices

The follow section contains all appendices or in the case of scripts reference to the specific file accompanying the project along with a Github link to the same script. If the reader does not use Jupyter notebooks it might be easier to asses the scripts on Github then by opening the notebooks as .txt files.

9.1 The coding-scheme of Mainwaring and Pérez-Liñán

Can be found in at appendix of Vol. I.

9.2 Results from vol. I Script

A notebook containing the script can be found in the appended file vol_1_roc.ipynb
or on Github: https://github.com/Polichinel/PDS_notebooks_done/blob/master/vol_1_roc.ipynb

9.3 Revised twitter-scrapping script

A notebook containing the script can be found in the appended file new_scrapper.ipynb
or on Github: https://github.com/Polichinel/PDS_notebooks_done/blob/master/new_scrapper.ipynb

9.4 Reddit-scrapping script

For the sake of prudence I created separate word-embeddings on the basis of each subreddit and subsection respectively. I assessed the substantive quality of the models through a number of sanity checks e.g. by testing which word where closet to "president" and other words of political relevance. Here I find no reason not to include them in the final aggregated corpus.

I also created separate predictions tests on the basic of twitter, the various subreddits and the combined reddit comments. Surprisingly the Reddit-based word-embeddings not only performed as well as the the twitter-based word-embeddings , the combined reddit comments out-performed the predictions of the twitter-based word-embeddings. This is interesting, not least since to two corpses are of comparable sizes. Exactly why reddit worked so well is hard to

say, but it does highlight the potential of using subreddits to obtain subject specific text-pieces.

A notebook containing the script can be found in the appended file reddit_scrapper.ipynb or on Github: https://github.com/Polichinel/PDS_notebooks_done/blob/master/reddit

9.5 Pre-processing and feature engineering script

A notebook containing the script can be found in the appended file feature_enginering_bow.ipynd
or on Github: https://github.com/Polichinel/PDS_notebooks_done/blob/master/feature_enginering_bow.

9.6 List of stopwords and high frequency words

Stop words: ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', 'her', 'hers', 'herself', 'it', 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', 'should', 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', 'couldn', 'didn', 'doesn', 'hadn', 'hasn', 'haven', 'isn', 'ma', 'mightn', 'mustn', 'needn', 'shan', 'shouldn', 'wasn', 'weren', 'won', 'wouldn']

High frequency "words": ['@', 'a', '...', 'rt', ':', ',', 'to', 'the', '#']

9.7 Combined TDM

The script regarding the cross validation steps taken here can be found in the appended file feature_enginering_bow.ipynd or on Github: https://github.com/Polichinel/PDS_notebooks_do

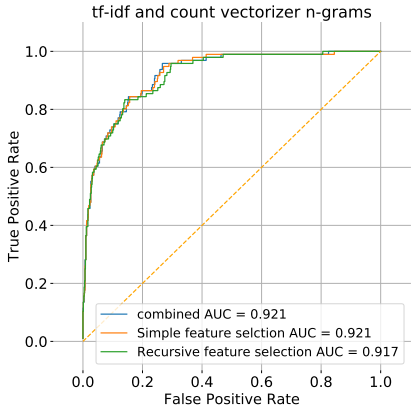

Figure 11:

9.8 Word-embeddings: grid search for hyper-parameters

It is here important to note the the process of creating these models are not deterministic. Even with seeds set for the random number generator the fact that I used parallel-processing to speed up the process induced some randomness - see more in the Gensim documentation radimrehurek.com/gensim/models/word2vec.html. One can save a specific model, and then of course it will never change. But without a valid method for reliable settings for the hyper-parameters one is somewhat at lucks mercy if one wants to create a new word-embedding model perhaps using a larger or otherwise different corpus. Though it could be greatly streamlined and further automated the approach taken here relies on cross-validation over luck. In this spirit I was also more interested in identifying fertile neighborhoods of hyper parameter which consistently produced good results - rather then single settings which spiked at one point.

The script regarding the cross validation steps taken here can be found in the appended file w2v_gridS.ipynd or on Github: https://github.com/Polichinel/PDS_notebooks_done/blob/ma

9.9 Table regarding TSVD

9.10 Classifiers and optimization script

The script regarding the cross validation steps taken here can be found in the appended file prediction.ipynb or on Github: https://github.com/Polichinel/PDS_notebooks_done/blob/maste

	AUC	TSVD AUC	w2v incl.	# Features	# TSVD F.
Count TDM original	0.92	0.914	no	75,852	2351
Count TDM 1. step	0.921	0.921	no	17,222	1164
Count TDM 2. step	0.913	0.914	no	6919	614
Count TDM original incl. w2v	0.934	0.932	yes	76,302	2353
Count TDM 1. step incl. w2v	0.935	0.936	yes	17,672	1190
Count TDM 2. step incl. w2v	0.93	0.93	yes	7069	677
tf-idf TDM original	0.935	0.936	no	75,852	2627
tf-idf TDM 1. step	0.934	0.934	no	14,125	1401
tf-idf TDM 2. step	0.931	0.932	no	5858	661
tf-idf TDM original incl. w2v	0.932	0.933	yes	76,302	2444
tf-idf TDM 1. step incl. w2v	0.933	0.934	yes	14,575	1073
tf-idf TDM 2. step incl. w2v	0.929	0.928	yes	6308	572
Combined TDM original	0.921	0.917	no	151,704	2386
Combined TDM 1. step	0.921	0.922	no	35,451	1405
Combined TDM 2. step	0.917	0.917	no	10,429	822
Combined TDM original incl. w2v	0.935	0.933	yes	152,154	2386
Combined TDM 1. step incl. w2v	0.933	0.937	yes	35,901	1422
Combined TDM 2. step incl. w2v	0.933	0.933	yes	10,879	869

Table 5:

9.11 ANN, RNN and LSTM script

The script regarding the cross validation steps taken here can be found at the end
of the appended file prediction.ipynb or on Github: https://github.com/Polichinel/PDS_notebooks_done/blo