Machine Learning Engineer Nanodegree

Capstone Project

Steffen Bunzel
January 12th, 2019

I. Definition

Project Overview

This project evaluates the use of machine learning to tackle the omnipresent problem of information overload.
In particular, it targets the ever growing amount of textual information on the web in the form of blog posts.
Blog posts can be a great source of information on a wide variety of specialized topics. They are commonly more accessible than books
and many provide practical insights from experts free of charge. However, there is a large spread in quality between well-written, informative and low quality "clickbait" posts (Gardiner, 2015). Especially in "hot" fields, many authors try to benefit from the hype by putting out posts with catchy titles to impress newcomers and gather clicks without sharing any insightful content.

One of these is the field of machine learning itself. This area has received increasing attention in recent years (compare Figure 1). As a consequence, the topic is covered extensively in both popular media and by technical writers.

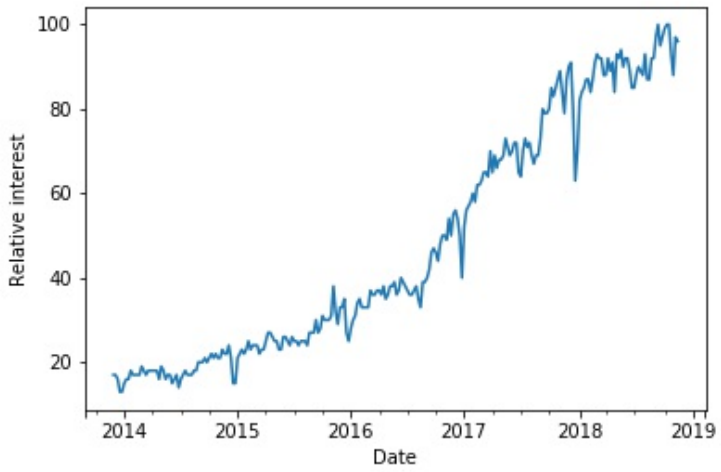

Figure 1: Interest in machine learning as measured by Google Trends

While this growing interest has certainly fueled investment in promising technologies powered by machine learning, it also makes it harder to separate the signal from the noise. The remainder of this report details an approach that leverages machine learning to tackle this problem. To do so, a system is developed that automatically suggests whether or not the user will find a particular machine learning blog post interesting based on the post's text.

Problem Statement

In order to delineate well-defined objectives and benchmarks, the problem space is narrowed down by imposing three central restrictions:

1. The focus is on machine learning blog posts published at medium.com.

2. These posts are grouped in two categories: "Interesting" and "Not interesting". These will be what machine learning is used to differentiate between. The machine learning task will therefore be *binary classification*.
3. The definition of "interesting" and the prioritization of the two types of error a binary classification model can make (1. Cluttering the results by wrongly predicting an uninteresting post to be interesting vs. 2. missing an interesting post by wrongly predicting it as uninteresting) is done based on the authors preferences (see section on *Metrics* below).

Of course, restriction 3) does not imply that the approach is only relevant to the authors preferences. The general framework can be tailored to anyone by substituting the input labels and adjusting the metric prioritization based on their liking.

To illustrate this, compare an exemplary problem statement tailored to the author:

> I love to read blog posts on machine learning (ML) and artificial intelligence (AI) on medium.com, a popular online publishing platform. However, not all of the articles available there fit my taste. Unfortunately, I sometimes find this only after having read the article. In particular, I do not enjoy articles which are superficial and aim to benefit from the hype around ML and AI, thereby confusing beginners. On the other hand, I enjoy articles offering a deep coverage of technical concepts and are written in a way that keeps the reader engaged nevertheless. As I am always short on time, I am particularly interested in a solution that reliably identifies blog posts that I will really enjoy as interesting to me and would be willing to sacrifice a few articles that I might have liked, but which the model misses.

Metrics

The metrics that will be used to assess the classifier's performance need to balance two aspects: The user has limited capacity to read blog posts and wants to find the ones they are interested in with high precision. Thus, the first metric employed is precision:

```
precision = true positives / (true positives + false positives)
```

Based on the author's problem statement above, a **target precision of 95%** will serve as a guideline for this project, which means that only one in twenty posts recommended by the classifier are actually uninteresting.

On the other hand, the number of article recommendations should not be restricted too much. Therefore, the model's ability to identify a substantial share of the actually interesting posts as such must be taken into consideration as well. This objective is covered by the `recall` metric. As the author prioritizes precision over recall, the objective for this metric is set to a **recall of 75%**, which means that only one in four interesting posts are classified as not interesting:

```
recall = true positives / (true positives + false negatives)
```

II. Analysis

Data Exploration

To tackle this problem, a dataset of ML blog posts published on medium.com is needed.
Luckily, a Kaggle user has already provided a collection of such data on the platform. This dataset was scraped from medium.com and made available under the `CC0: Public Domain license, i.e. as a work in the public domain without copyright` license. The dataset contains the post's author, its estimated reading time, a link to the post, its title, its text body as well as the number of claps (a form of like) it has received to date.

While the raw data contained 337 rows, it included several duplicates, resulting in 230 unique articles. Of these, six were not written in English, thus reducing the number to 224. To keep the requirements with respect to provisions from the user manageable, the top 130 posts based on their number of claps were labeled as either "interesting" or "uninteresting" by the author after manually assessing each post on medium.com for 2-5 minutes.

Thus, the data contains 130 observations of 5 potential features (ignoring the link to the article) and one target variable ("interesting" or "uninteresting"). Two of these features are numerical: Claps and reading time.

	claps	reading_time
count	130.00	130.00
mean	7064.05	10.67
std	9836.08	5.61
min	10.00	3.00
25%	1500.00	7.00
50%	3300.00	9.00
75%	8450.00	13.00
max	53000.00	31.00

Table 1: Summary statistics on the numerical features

The other columns all contain non-numeric values and are almost always unique.

	author	title	text
count	130	130	130
unique	102	130	130
top	Adam Geitgey	Distributed Neural Networks with GPUs in the A...	What Machine Learning Teaches Us About Ourselv...
freq	6	1	1

Table 2: Summary statistics on the non-numerical features

These basic statistics show that the data underlying this project has a simple structure: It consists of the contents of a blog post (represented by its title and text body) as well as three pieces of metadata (the post's author, the number of claps and the estimated reading time).

As mentioned above, the target variable of this dataset is whether the author found the particular article interesting or not after assessing it manually for 2-5 minutes. This variable is binary: It is 0 for posts the author found uninteresting and 1 for posts he found interesting. The distribution of the classes is slightly imbalanced: Approximately 35% of the posts are labeled "interesting" while the remaining 65% are labeled "uninteresting".

Exploratory Visualization

Arguably the simplest approach to solving this problem with machine learning would be to focus on the numerical columns only, i.e. build a classification model based on number of claps and reading time. One advantage of this approach is that due to the feature space having just two dimensions, the separability of the two classes can be visually assessed.

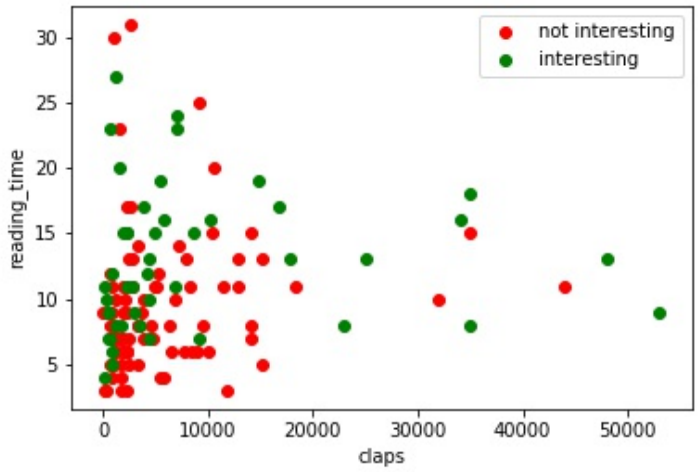

Figure 2: Classes by claps and reading time

Figure 2 shows all posts in the data set in a scatterplot of claps and reading time. The articles labeled "interesting" are marked green, while the articles labeled "not interesting" are colored in orange. The illustration shows two main things:

1. The data is not easily separated linearly.
2. A clear non-linear separation is not obvious either.

These two characteristics underline the need to evaluate alternatives to using metadata to distinguish interesting from uninteresting articles.

Algorithms and Techniques

As the focus of this work is on using just a post's text as an input for a classification task, the potential algorithmic approaches vary greatly depending on the approach used for pre-processing the data. The approaches evaluated in this project can be differentiated in three categories:

1. Transforming the text into a numerical matrix based on word counts (absolute counts or relative to the frequency across all documents) allows training an arbitrary machine learning model. In particular two standard model classes were evaluated: A random forest classifier and a support vector machine classifier.
2. The second class of approaches aims to equip the model with a form of prior knowledge in the form of pretrained word embeddings, i.e. numerical vectors that represent each word and are derived from general purpose corpora in an unsupervised manner. While they could theoretically be used with a variety of modeling approaches, using word embeddings in combination with neural networks has recently become popular. In particular, this work evaluates approaches showcased in a current Kaggle competition on text classification and makes use of the GloVe word embeddings from Stanford NLP.
3. Going one step further, a third class of approaches, popularized through works like fast.ai's ULMFiT or Google's BERT in 2018, encodes prior knowledge not "merely" through word embeddings, but pre-trained language models. These models are trained to predict the next token in a sequence and are used to extract higher-level features from raw text, which then serve as an input to upstream tasks such as classification.

The main difference between class 1 approaches and class 2/3 approaches is in how they treat the texts. The class 1 approaches treat a document as an unordered collection of words (also called `bag-of-words`) and vectorize it based on counts. Class 2 and 3 approaches treat text as a `sequence` of words and conserve information about ordering in vectorizing the texts.

Benchmark

As the data set used in this project is specific to the interest of the author, naturally no publicly available benchmarks exist. However, there is one obvious point of comparison: The performance of a model trained just on the numerical features available.

Thus, three classes of classification models were trained and tuned on the numerical columns `claps` and `reading time` without any preprocessing. The models were evaluated using 6-fold cross-validation and a shared test set containing 30 percent of the full data set used for comparison with the other model classes as well as the text based models introduced above. For evaluation on the cross-validation folds, the Receiver Operating Characteristic Area Under the Curve (roc auc) was used as it allows to estimate performance independent from the threshold chosen for classifying observations as positive. For final evaluation on the hold out test set, the predictions from each of the models trained during cross-validation were averaged and assessed based on precision and recall. Table 3 summarizes the results of this evaluation.

method	mean train auc	mean cv auc	mean test auc
baseline_rf	0.943830	0.707341	0.595714
baseline_svc	0.833333	0.501984	0.524286
baseline_lr	0.740426	0.718254	0.605714

Table 3: Summary of baseline results

III. Methodology

Data Preprocessing

The preprocessing steps required to transform the base data into a format that allows modeling as described above can be distinguished in two categories:

1. General preprocessing: As indicated above, the raw data contained a few duplicate entries as well as blog posts written in a language other than English. These were removed prior to doing any modeling as they would only act as a confusion, regardless of model class. To do so, the Python libraries `pandas` (for data processing in general and duplicate removal in particular) and `spacy` (for language detection) were used.

2. Task-specific preprocessing: When transforming text into a feature matrix that can serve as an input for a machine learning algorithm, one has to make several important choices, for instance:

    - How to convert raw text into tokens, which can be represented by a unique numerical identifier? For example, one might want to represent "we will" and "we'll" by the same two tokens.
    - Which vocabulary to use, i.e. should all unique words in the full corpus be represented as a separate token in the vocabulary?
    - Whether to treat texts as a sequence of tokens or as a bag-of-words

In this work, several open-source machine learning libraries were used: `scikit-learn`, `keras` and `fastai`. All of these come with their own methods for text preprocessing (which might again be outsourced, for example `fastai` uses `spacy's` tokenization engine under the hood). Thus, depending on which modeling approach was used, this work relied on slightly different answers to the questions just posed.

By default, all of these will use the full collection of unique words or a predefined number of most frequent words in the given corpus of documents as their vocabulary. As an alternative, only the words, which are not part of the top 10.000 most frequent English words in general (as determined by n-gram frequency of the Google Trillion Word Corpus), were used as a specific vocabulary.

Implementation

Following the differentiation outlined in chapter `II. Algorithms and Techniques`, the implementation was structured based on the

three classes of models evaluated in this work:

1. Count-based vectorization + random forest or support vector machine using `scikit-learn's` implementations for transforming text into numerical features as well as the machine learning models.
2. Sequence tokenization, word embeddings + neural network using the `keras` implementations for tokenizing text and combining embedding layers, dense layers, rectified linear activations and dropout layers into a deep neural network.
3. Sequence tokenization, pretrained language model + neural network using `fastai's` implementation for tokenizing text (internally using `spacy's` Tokenizer), a pretrained recurrent neural network based language model (pretrained on Wikipedia) provided by the `fastai` library as well as its implementation for combining the features extracted from raw text by the language model with classification layers.

The biggest challenge with these diverse approaches was to evaluate them consistently, especially given the small amount of data available for training and evaluation. The three ML libraries used for this work all have slightly different interfaces. In addition, the evaluation metrics used here are dependent on the treshhold chosen for classifying a post as positive versus negative. While `scikit-learn` offers a variety of purpose-specific metrics to tackle this, the deep learning libraries `keras` and `fastai` ask their users to implement metrics other than a few standards themselves. To facilitate consistency, the author implemented a common interface for `scikit-learn` and `keras` models that allows the latter to be evaluated using convenience methods provided by the former. In this context, a custom strategy was implemented to enable cross validation of models from both libraries through a common interface. `fastai` was not plugged into this interface due to time constraints and their workflow being substantially different from the `scikit-learn` standard.

In addition, due to the data being very small, exploratory analysis showed that the performance obtained from the different methods (especially when comparing the various combinations of vectorization strategy + classifier) differed quite substantially based on how the data was initially split into train and test set. To get an even more robust estimate of out-of-sample performance and to minimize the influence of randomness, all results where averaged across five different random seeds.

Refinement

Class 1: Count Vectorization / Term-Frequency-Inverse-Document-Frequency (TFIDF) + Classifier

To determine which combination of vectorization strategy (CountVectorizer vs. TFIDF, full vocabulary vs. only specific vocabulary) and classification model (random forest vs. support vector machine) is most promising for the given dataset, all possible combinations were tested across five different random seeds using 6-fold cross-validation for each seed. `Table 4` summarizes the results.

	train auc		cv auc		test auc	
	mean	std	mean	std	mean	std
countvec_svc_full	1.000000	0.000000	0.737302	0.043687	0.788000	0.064504
countvec_svc_specific	0.999858	0.000201	0.773413	0.067830	0.754286	0.059625
countvec_rf_full	0.999532	0.000377	0.714087	0.043839	0.751143	0.053427
tfidf_rf_full	0.999191	0.000374	0.665079	0.027228	0.748857	0.069645
countvec_rf_specific	0.999787	0.000194	0.647222	0.036824	0.748571	0.064957
tfidf_rf_specific	0.999518	0.000373	0.611111	0.075511	0.715429	0.059087
tfidf_svc_specific	0.999972	0.000063	0.612302	0.040012	0.490286	0.086958
tfidf_svc_full	0.833333	0.288675	0.529167	0.068083	0.466857	0.093167

Table 4: Results from class 1 approaches averaged over different random seeds

While almost all approaches allow for fitting the training data well, the approaches using count vectorization and support vector machines seem to generalize best. Thus, a stochastic grid search for these two approaches was done using scikit-learn's `RandomizedGridSearchCV`.
The following parameters were optimized:

- CountVectorizer
  - `stop_words`: Whether or not to remove stop words
  - `ngram_range`: Minimum and maximum number of tokens to consider as belonging together
  - `max_df`: Which percentage of most frequent terms to ignore
  - `min_df`: Which percentage of least frequent terms to ignore
  - `max_features`: Maximum number of tokens to be considered ordered by term frequency across the corpus
- Support Vector Machine Classifier
  - `C`: Penalty on the error term, a higher value places more emphasis on classifying all observations correctly and is thus more likely to lead to overfitting
  - `gamma`: Kernel coefficient for the `rbf` kernel. Determines how much influence a single observation has. Higher values are more likely to lead to overfitting
  - `class_weight`: Whether or not to adjust the parameter `C` inversely proportional to class frequencies

Compare chapter `IV. Results - Model Evaluation and Validation` for the results of this optimization.

Class 2: Word Embeddings + Neural Network

The second class of models tokenizes the blog posts as sequences instead of bag-of-words (i.e., they preserve the order in which words appear and do not "just" look at the number of occurences in a document), makes use of pretrained word embeddings (i.e. numerical vectors representing a word) and combines these in a neural network architecture with recurrent layers and a classification layer. Inspiration for these models was taken from a current Kaggle competition aimed at identifying insincere questions on quora.com. For the evaluation of these models, a simple architecture of one embedding layer, one bidirectional layer of gated recurrent units, a max pooling layer and one dense layer leading to a final dense layer with sigmoid activation for binary classification was used. `Adam` was used as the optimizer and `binary crossentropy` as the loss function. Different strategies were tried for applying `dropout` and `weight decay` for regularization. Furthermore small changes to the architecture (e.g. changing the number of units per layer) and different preprocessing strategies were experimented with. The latter was done primarily based on this Kaggle kernel and was aimed at getting the vocabulary used in sync with the vocabulary for which word embeddings are available.

Unfortunately, all of these measures did not lead to any substantial improvements to the performance metrics shown in `table 5`. `Chapter V` discusses a potential reason why this class of approaches does not seem to be well suited for the problem at hand.

	train auc		test auc	
	mean	std	mean	std
embeddings_keras	0.943432	0.039151	0.650857	0.113234

Table 5: Results from class 2 approach averaged over different random seeds

Class 3: Pretrained Language Model + Neural Network

The experience with the last class of approaches, i.e. using a language model to extract features from the raw text which are then fed into a two layer neural network for classification, was similar to that described above for class 2: Independent from the method tried to improve the results (mainly longer finetuning of the langugage model, adding regularization), improvements did not

really show.

`Table 6` summarizes the results.

	train auc		test auc	
	mean	std	mean	std
lm_fine_tuning	0.909781	0.045641	0.66	0.098602

Table 6: Results from class 3 approach averaged over different random seeds

IV. Results

Model Evaluation and Validation

The best models found during the extensive hyperparameter search and evaluation described in chapter `III. Methodology - Refinement`, used n-gram tokenization in combination with machine learning models which do not consider sequential information (in particular, random forests and support vector machines). The model refinement phase included a sensitivity analysis in changing the random seed used for sampling the training and test set as well as the model parameters which entail randomness.

Table 7 summarizes the resulting performance metrics for the modeling approaches most successful with default parameters. All metrics are averaged over the same random seeds used for the default models. These results show slight improvements in all metrics.

	train auc		cv auc		test auc	
	mean	std	mean	std	mean	std
best_params_countvec_svc_full	1.0	0.0	0.757143	0.053839	0.798286	0.056590
best_params_countvec_svc_specific	1.0	0.0	0.775794	0.068431	0.762857	0.052294

Table 7: Results from top class 1 approaches after parameter tuning

For the parameter settings found by the grid searches please refer to `Appendix IV`.

So far, we have used the area under the ROC curve as an auxiliary metric as this allowed us to compare models independent of the threshold chosen for classifying observations as positive. The performance metrics described in chapter `I. Definition - Metrics`, however, do require to choose a threshold. For the final evaluation, only the scikit-learn pipeline using count vectorization on the full vocabulary and a support vector machine (the parameters of which have both been optimized via grid search) was considered. Compare `Appendix 4.A` for a full specification of the pipeline used. The final model was evaluated on five different random seeds. `Table 8` summarizes the best combination of precision and recall given the target ratio of approximately 1.25 (0.95 / 0.75). For comparison, the baseline model was evaluated using the same random seeds (and ensuing train-test splits). Please refer to `Appendix V` for a detailed overview of the results. The summary table shows that the best text-based model performs significantly better than the simple benchmark model across all metrics except when looking at recall in isolation. The thresholds used to calculate precision and recall were determined based on the f_beta score, using a beta of 0.8 (= 1 / 1.25) to reflect the relative weighting of precision and recall determined by the objective of this work.

		baseline_model_lr	final_model_svc_countvec_full
roc auc	**mean**	0.711429	0.802286
	std	0.063728	0.054601
precision	**mean**	0.533736	0.670779
	std	0.082874	0.083081
recall	**mean**	0.785714	0.742857
	std	0.133631	0.148117
f_0.8	**mean**	0.620629	0.713890
	std	0.037521	0.029165

Table 8: Comparison of best text based model vs. benchmark model on numerical features

Justification

As the results presented in chapter `IV. Results - Model Evaluation and Validation` show, the best text-based model has a significantly higher general classification performance than the baseline model. The test set ROC AUC averaged over five different random seeds is approximately 12.77% higher. The average precision, which is the prioritized metric in this work, increased by 25.68%. On the other hand, the average recall is 5.77% higher for the baseline model.

In absolute terms the initial objectives of this work were not achieved. The final results of an `average precision of 0.67` and an `average recall of 0.74` are in aggregate still not close to the objective of `precision >= 0.95` and `recall >= 0.75`.

V. Conclusion

Free-Form Visualization

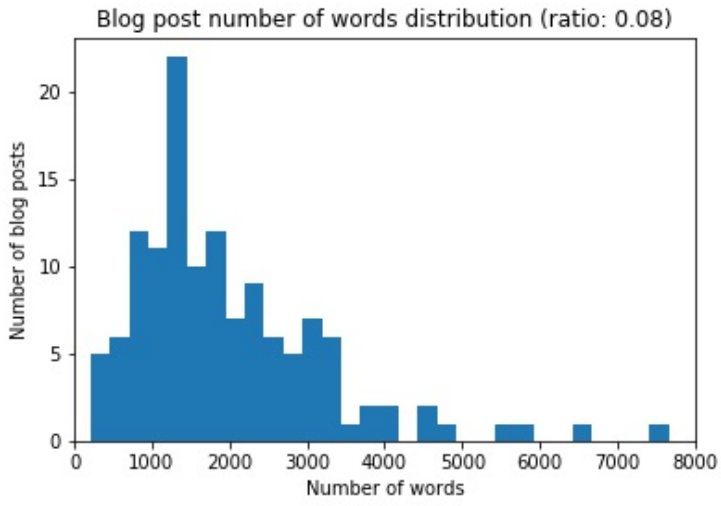

Figure 3: Distribution of word count per blog post

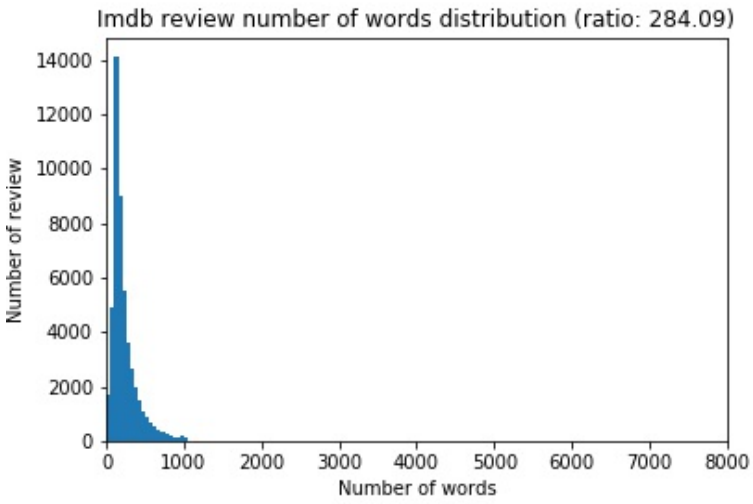

Figure 4: Distribution of word count per imdb review

`Figures 3 and 4` illustrate a characteristic about the dataset that was underappreciated at the outset of this project, but could serve as an explanation as to why the approaches treating the texts as bag-of-words and using a "shallow" classifier work better on the given problem than those treating the texts as sequences and making use of deep learning. `Figure 3` shows a histogram of word counts per blog post. The median length is `1683` words. For comparison, the same histogram is shown for the popular imdb sentiment classification dataset (more precisely, the version of this dataset that comes with keras). Here, the median length is `176` words. When we consider that the number of samples is `130` for the blog posts and `50.000` for imdb, we see that the `number of samples / number of words per sample` is considerably lower for the blog posts than for the imdb data (`0.08 vs. 284.09`). In other words, the problem at hand is one of a comparatively `very low number of samples of very long texts`. Others have found that these properties tend to favor bag-of-words + shallow classifier vs. sequence tokenization + sequence aware classifiers as well (Google Developers Guide for Text Classification). Google's developer guide mentions a ratio of `1500` as a heuristic for which approach is likely to work better (if it's smaller than 1500, use bag-of-words, if it's larger than 1500 use sequence tokenization). While it seems like this boundary is rather high (very competitive results have been achieved on the imdb dataset using class 2 and 3 approaches), the results of this work certainly confirm the general direction.

Reflection

This work describes an end-to-end machine learning project: It started with gathering data as well as cleaning and hand-labeling it. After the user requirements were translated into metrics that can measure the performance of an analytical model, relevant combinations of preprocessing and modeling techniques were identified. To enable their consistent evaluation, helper functionality was developed. For each candidate approach, a model using default parameters was trained and evaluated. Only the most promising ones were finetuned to find the final model. This model was finally evaluated based on the performance metrics defined at the start of the project and compared to a simple baseline.

What was interesting about this project was the large variety of potential combinations of preprocessing and modeling steps that can be used when working with text. In addition, the field of natural language processing has seen many new and promising approaches from the deep learning community over the last year. It was very interesting to explore these and try to get a better understanding of how they differ and relate to each other.

However, keeping track of all of these developments and at the same time keeping a clear focus on a few promising approaches was challenging. What's more, given that many techniques are still very new, implementations are scattered across different libraries and do not necessary follow a consistent framework. Making these comparable was a key challenge of this project, which was not fully met. Another challenge was to define the right evaluation metric. To assess the performance of models across classification thresholds, ROC area under the curve was used. However, the initial objective metrics were stated in terms precision and recall, metrics which require to choose a threshold. For the final evaluation, the threshold was optimized for based on a custom `f_beta` score with `beta = 0.8` based on the target ratio between precision and recall. Consistently using this metric from the start would have reduced complexity.

The practical use of the final model is limited, even though it represents a substantial improvement over the baseline. It seems like the costs for collecting the amount of labeled data that would be needed to achieve the objectives for precision and recall would outweigh the benefits of the classifier. Follows the right people on twitter and using their recommendations as an indication on which articles might be interesting seems like a more efficient approach for now.

Improvement

Above all, more labeled data would most likely be needed to improve the results of this projects. In addition to that, one could try to "augment" or "transform" the existing data in some way to make it easier for the models to learn. One such approach was tried in this project: Split each article on the newline characters, treat every paragraph as a separate training example and then average the predictions on the article level. However, this was only briefly tried on the class 3 approach and not thoroughly evaluated. For image classification problems, data augmentation is a large component of a successful modeling pipeline and helps greatly in improving generalization. However, for text data, it is less clear which kinds of transformations could work, if any.

On top of that there are a few more fine-grained improvement opportunities:

- Develop a unified interface for comparing the results from different libraries and enable ensembling for all of them (e.g. by storing model architectures and results separately on disk and reloading them for evaluation instead of holding them in memory as was done in the project). This would allow for more consistent evaluation of different techniques making use of different libraries.
- Add feature selection for the bag of words tokenization approaches. The Google developer guidelines referenced above adds this step after vectorization of the texts and before modeling.
- Add other models (such as naive bayes or multilayer perceptron) to the evaluation pipeline for bag-of-words approaches.
- …

There are plenty of things one could try to improve the results. However, extensive experimentation has already been done and it seems like more data would be needed to make any substantial improvements.

Appendix

Appendix I

method	mean train auc	mean cv auc	mean test auc
countvec svc full v3	1.000000	0.787698	0.900000
countvec rf specific v2	0.999929	0.636905	0.847143
countvec rf full v3	0.999220	0.734127	0.827143
countvec svc specific v5	0.999716	0.674603	0.822857
tfidf rf full v4	0.998794	0.656746	0.815714
countvec svc specific v3	1.000000	0.855159	0.797143
tfidf rf full v5	0.999716	0.624008	0.792857
tfidf rf full v1	0.999007	0.670635	0.787143
countvec svc full v4	1.000000	0.763889	0.782857
tfidf rf specific v4	0.999716	0.690476	0.781429
countvec svc specific v1	1.000000	0.744048	0.765714
countvec rf specific v5	0.999787	0.641865	0.765714

countvec svc full v1	1.000000	0.746032	0.762857
countvec rf full v1	0.999433	0.731151	0.762857
countvec svc full v5	1.000000	0.676587	0.754286
countvec rf full v4	0.999149	0.760913	0.751429
countvec svc full v2	1.000000	0.712302	0.740000
tfidf rf specific v3	0.999149	0.682540	0.737143
countvec rf full v5	0.999929	0.646825	0.735714
countvec rf specific v3	0.999716	0.673611	0.731429
countvec rf specific v1	0.999504	0.594246	0.728571
tfidf rf specific v1	0.999858	0.521825	0.720000
tfidf rf specific v2	0.999787	0.608135	0.718571
countvec svc specific v4	1.000000	0.799603	0.694286
countvec svc specific v2	0.999574	0.793651	0.691429
tfidf rf full v2	0.999433	0.697421	0.688571
countvec rf full v2	0.999929	0.697421	0.678571
countvec rf specific v4	1.000000	0.689484	0.670000
tfidf rf full v3	0.999007	0.676587	0.660000
tfidf svc full v1	1.000000	0.472222	0.622857
tfidf rf specific v5	0.999078	0.552579	0.620000
tfidf svc specific v4	1.000000	0.567460	0.597143
tfidf svc specific v1	1.000000	0.619048	0.554286
tfidf svc specific v2	0.999858	0.630952	0.482857
tfidf svc full v4	0.833333	0.483135	0.471429
tfidf svc full v5	1.000000	0.593254	0.437143
tfidf svc specific v3	1.000000	0.665675	0.434286
tfidf svc full v3	0.333333	0.484127	0.422857
tfidf svc specific v5	1.000000	0.578373	0.382857
tfidf svc full v2	1.000000	0.613095	0.380000

Table 9: Detailed results from class 1 approaches

Appendix II

method	train auc	test auc
embeddings keras v1	0.981992	0.805714
embeddings keras v4	0.930614	0.677143
embeddings keras v3	0.952860	0.668571
embeddings keras v5	0.969280	0.608571
embeddings keras v2	0.882415	0.494286

Table 10: Detailed results from class 2 approach

Appendix III

method	train auc	test auc
lm fine tuning v1	0.886123	0.757143
lm fine tuning v2	0.880826	0.662857
lm fine tuning v3	0.962394	0.560000

Table 11: Detailed results from class 3 approach

Appendix IV

A. Best Parameters for CountVectorizer, Support Vector Machine and Full Vocabulary

```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
           dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
           lowercase=True, max_df=0.957962317501337, max_features=6020,
           min_df=0.06021641222487595, ngram_range=(1, 1), preprocessor=None,
           stop_words=None, strip_accents=None,
           token_pattern='(?u)\\b\\w\\w+\\b', tokenizer=None)

SVC(C=2.2281934634464564, cache_size=200, class_weight='balanced', coef0=0.0,
               decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
               max_iter=-1, probability=True, shrinking=True,
               tol=0.001, verbose=False)
```

B. Best Parameters for CountVectorizer, Support Vector Machine and Specific Vocabulary

```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
           dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
           lowercase=True, max_df=0.9253642126352534, max_features=11106,
           min_df=0.05762831674562723, ngram_range=(1, 2), preprocessor=None,
           stop_words=None, strip_accents=None,
           token_pattern='(?u)\\b\\w\\w+\\b', tokenizer=None)
```

```
SVC(C=1.6612794146075704, cache_size=200, class_weight=None, coef0=0.0,
                decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
                max_iter=-1, probability=True, shrinking=True, tol=0.001,
                verbose=False)
```

Appendix V

	method	roc_auc	threshold	precision	recall	f_0.8
0	final_model_svc_countvec_full_v3	0.885714	0.579919	0.714286	0.714286	0.746812
0	final_model_svc_countvec_full_v5	0.811429	0.548128	0.750000	0.642857	0.739479
0	final_model_svc_countvec_full_v4	0.805714	0.403528	0.590909	0.928571	0.711615
0	final_model_svc_countvec_full_v2	0.765714	0.542546	0.727273	0.571429	0.691983
0	final_model_svc_countvec_full_v1	0.742857	0.646661	0.571429	0.857143	0.679558
0	baseline_model_lr_v3	0.808571	0.394291	0.642857	0.642857	0.672131
0	baseline_model_lr_v4	0.734286	0.379447	0.600000	0.642857	0.642857
0	baseline_model_lr_v1	0.640000	0.315321	0.500000	0.857143	0.615770
0	baseline_model_lr_v5	0.688571	0.277611	0.464286	0.928571	0.592881
0	baseline_model_lr_v2	0.685714	0.285543	0.461538	0.857143	0.579505

Table 12: Full evaluation results