Vandalism Detection (WSDM Cup 2017)

Matthew Dragotto
Andy Kuang
Shuhao Sun

Table of Contents:

Abstract….....2

1. Introduction……...2

2. Project Implementation.....4

3. Performance and Results.....7

4. Related Work.....9

5. Conclusion.....10

6. References.....11

7. Task and Workload Distribution.....11

Abstract

The Vandalism Detection implemented in the WSDM Cup 2017 was won by team Buffaloberry with an ROC score of .947. Our model is implemented two different ways using tensorflow and sk-learn. Both use linear classifiers in their implementation. Our implementations managed to produce a ROC score of .853, which is weaker but still sufficient to classify a majority of true positives and true negatives.

1. Introduction

The project we attempted to complete was Vandalism Detection, which was part of the WSDM Cup 2017. This cup was initiated in order to search for ways to improve vandalism detection in changes done to wikidata. Wikidata is the database for several systems, including the very well know Wikipedia.

A revision that would be classified as vandalism is one that is done with the intent to harm. This is usually done through falsifying information, and is extremely harmful as it can result in misinformation being spread across the web. This is emphasized due to the fact that Wikipedia and related sites can be edited by anyone. Therefore, it is easy for data to be vandalized if there are no checks in place.

The aim of Vandalism Detection is to detect vandalism in edits as each edit is submitted. This means that if vandalism is detected in a submission, the revision will be rolled back in real time and the edit will not make it through for display to others.

1.2. Project Specification and Details

For the project, we were given three different datasets. These are the training, validation, and testing datasets. The training datasets were to be used to train our classifier. The validation dataset was used to “validate” or test our classifier. Finally, the testing dataset was released after the deadline of the competition and is the dataset that the classifier was supposed to be applied to in order to make predictions. In our case, the testing dataset simply serves as another validation dataset, as we were not actually a part of the competition.

For each dataset, there are several parts. The first is a revisions dataset that contain information about the revisions. This includes twenty-one different datasets of varying

sizes across four years. These datasets are each contained in separate xml documents, and parsing the xml gives us the stored information. In each xml document, several pages are contained. Each of these pages also contain several revisions, which are all edits that have been done to the specific page.

Along with the training datasets, there is a metafile document stored as a csv. This document includes REVISION_ID, REVISION_SESSION_ID, USER_COUNTRY_CODE, USER_CONTINENT_CODE, USER_TIME_ZONE, USER_REGION_CODE, USER_CITY_NAME, USER_COUNTY_NAME, and REVISION_TAGS. These categories provide specifics of the revisions and users who submit the revisions.

Finally, the last provided part of the dataset is the truth file, also stored as a csv. The categories of this csv are REVISION_ID, ROLLBACK_REVERTED, UNDO_RESTORE_REVERTED. This helps us train our classifier by showing us whether or not the revision was actually vandalism.

By joining together all the data we are provided and training a classifier, we can use the classifier to predict whether revisions are or are not vandalism. In order to decide how “good” our classifier is at doing this prediction, the ROC_AUC score is used. Essentially, ROC_AUC emcompasses true positives, false positives, true negatives, and false negatives. True positives and true negatives contribute to a higher score, while false positives and false negatives reduce the score. In using such a scoring system, we can test how well our classifier actually classifies as opposed to just getting a pure accuracy test. This is important as there are a few cases where pure accuracy test would lead us to believe our classifier is better than it actually is (such as when the ratio of vandalism to non vandalism is higher and our classifier simply classifies every single case as vandalism).

2. Project Implementation

2.1. Processing the data

The data provided on the project competition webpage was split into 3 sets: Training, Validation, and Testing data. For the Training set, over 15GB of compressed XML files were provided, from the years 2012 to 2016. Keep in mind this data was compressed, and when extracted, it amounted to hundreds of gigabytes of xml data. The largest of the files also happened to be the most recent, which is good because we hypothesized that they would be the most useful when training our model. This is because data from 2 months ago would contain a lot of the same users and webpages, as compared to data that was taken 4 years ago. The base of frequent users and frequent pages visited changes rapidly, and we wanted to stay as close to the relevant data as possible.

For these reasons, the subset of Training data we used was from 2015 and 2016. The unpacked data from late 2015 alone was 100GB in size, and the 2016 data in the training set amounted to just over 50GB. Due to the fact that this data was so large, it took a long time for our code to parse through it all, so while developing our code, we used a much smaller subset of data that was only about 100MB.

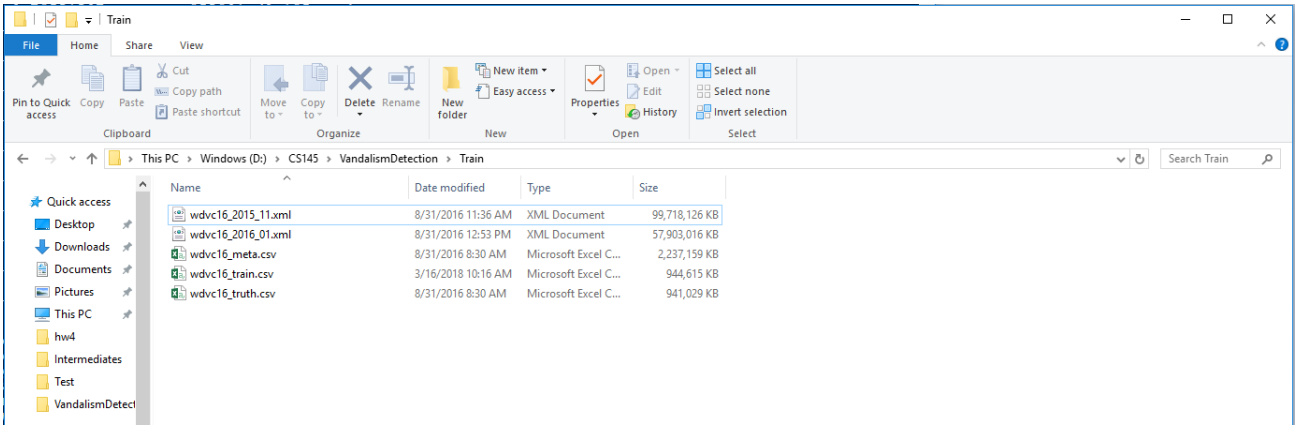

Figure 1: *The data from January 2015 and 2015 together comprised of about 150GB of data. Our output file after parsing that and joining it with the meta and truth information was almost 1 GB worth of CSV information.*

Our main objective while parsing through the xml files was to generate a csv file with all the relevant features we would eventually end up needing for our model. The code for parsing through the xml, known as data_to_csv.py, worked by iterating through each xml document page by page, and then using the lxml library in python to make a tree out of each page. Using this library, finding the data we wanted to extract was easy, because we could just use the xpath() function to retrieve element data in the tree. After

we retrieved all the features, we then used Python's csv library to write the entire row directly to a csv file. Once the entire csv file of features was generated from the entire set of xml documents, it was joined with the meta information and truth information into one giant csv file with the fields from each file. The meta and truth information was formatted in a way that made this simple, since the rows in each of the two files were sorted by revision id. In this way, the numerical revision id matched up row by row across each of the three documents, and joining the files was as simple as just going row by row and putting them together. This process of parsing and joining was repeated for the Training, Validation, and test sets, until we were left with three csv files, one for each set.

2.2. Choice of features

The xml document contained many different elements, and processing it all would take a lot of iterations, so it was important that we decided beforehand what the relevant fields would be. We decided that it would probably be a good idea to capture page titles, relevant user information, including user name, user id, user ip address, and also comment tags. We later ended up taking comment information out of our parsing, because some of our models actually had worse performance with them being used as a feature.

For user name and ID, it was only appropriately included when the user was known. Users with a good history of not submitting vandalism would indicate that continued submissions from them would also not include vandalism.

Otherwise, for anonymous users, we used their geographical information. If we have a high density of vandalism from a certain location, we can assume that revisions done by an anonymous user from that region has a high chance of being vandalism as well.

For IP in particular, we chose to split the IP into multiple parts in order to create more general categories. It is quite unlikely for the whole IP to be exactly the same, so using the IP as is would not be very helpful to our classifier. However, it is common for parts of the IP to be same, so in splitting the IP, we provide valuable information to our classifier. IPs are quite in line with location, so providing IP gives additional information to improve our classifier.

One feature that most probably would have improved our model but was impossible for us to implement is a dictionary of all "real" words. If we knew all the real words, we could be more certain of vandalism if a revision contained many fake words. Unfortunately, implementing such a feature would take much higher processing power than we could afford, as the amount of words in a single language is already vast.

Finding all real words in all the languages supported would simply take way too many resources.

2.3. Sklearn approach

For the sklearn approach, we basically use scikit-learn machine learning library for classification of wikidata revisions. Pandas is used in this approach to handle csv file input of features to create dataframe. The dataframe contains rows which represent samples of revisions and columns which represent features. Then the processing of features is done in the dataframe. In this approach all features of one sample will be made into one string and then the collection of strings will be passed to vectorizer to be transformed to a scipy.sparse matrix holding token occurrence counts. So in this model token patterns will be extracted as features from the string representing a sample, and the way to process and rearrange features is important. At first I tried to merge all raw values of features from csv file directly and the result is horrible with about 0.4 ROC_AUC score. Then in the processing of features, we add all values of features with prefixes like “page_title=”, “country_code=”. The USER_NAME and USER_ID is combined together and if USER_NAME is empty, “anonymous” is used instead. USER_IP is processed separately to split as mentioned in 2.2. After the string representations of features are transformed to a matrix by vectorizer, the matrix is passed to linear support vector classifier with every sample’s class label to train. After fitting the data into the classifier, the label prediction and scores of testing data can be get directly from builtin methods of classifier. The classifier used in this approach is from scikit-learn’s SVM (Support Vector Machines) library. From SVM, we tried both SVC with linear kernel and LinearSVC. They are built from different libraries and have different characteristics. SVC with linear kernel has a much higher ROC_AUC score for small datasets and LinearSVC model is better to fit large scale data. In the process of transforming feature strings to a matrix, the features number of matrix we choose will also influence algorithm behavior greatly. And the penalty parameter C for classifier also influences the result. When C is larger, the hyperplane tends to have a smaller margin. We tried different values of C, which in some situation can enhance both time and accuracy performances.

2.4. Tensorflow approach

TensorFlow has a very easy to use and approachable library for machine learning, which is why we chose to try it out. It has great documentation, and is written in python, so it was a natural fit for our project. It is also very modern, having released late 2015,

which means that at the time of the actual WSDM competition, not many people had probably even heard about it yet.

For this reason we wanted to use TensorFlow alongside of SK-Learn in order to see if would have any significant edge over it. In short, it didn't, but it was very easy to use. Most of the csv_to_tensor.py code comes straight from the example in the documentation about linear models. For our project, since most of our features were categorical instead of quantitative, and since it was a binary classification problem, we just used the wide learning model. TensorFlow does however offer support for deep learning as well, and also a combination of wide and deep learning.

The main part of implementing the TensorFlow approach was extracting our feature columns into what are called "categorical_column_with_hash_bucket"s. In doing so, we used the pandas library to determine how many unique entries were in a feature, and dynamically created that many buckets for holding information. This way we could have categorical data for as many possible values as we could when it came testing time.

3. Performance and Results

3.1. Sk-learn

First we trained on a small set of data for testing, which is about first 20000 revisions from wdvc16_2012_10.xml. The validation data is files of wdvc16_2016_03. The result is:

LinearSVC: C=1, Time used: 95.90s, ROC_AUC=0.806970, ACCURACY=0.998508

In this test C is default 1, the classifier used is LinearSVC, and Time used is time consumed for training and testing.

Then we tried to have 5 times features number of matrix and get the result:

LinearSVC: C=1, Time used: 302.83s, ROC_AUC=0.811228, ACCURACY=0.998508 (5 times features)

The time used hugely increased, and the roc score increased, too.

Then we changed value of C and testing again:

LinearSVC: C=5 Time used: 222.12s, ROC_AUC=0.806233, ACCURACY=0.998508

LinearSVC: C=0.5 Time used: 38.92s, ROC_AUC=0.807052, ACCURACY=0.998508

LinearSVC: C=0.1 Time used: 23.08s, ROC_AUC=0.807475, ACCURACY=0.998508

LinearSVC: C = 0.01 Time used: 13.96s, ROC_AUC=0.818697, ACCURACY=0.998508

LinearSVC: C = 0.001 Time used: 9.75s, ROC_AUC=0.831132, ACCURACY=0.998508

LinearSVC: C = 0.0001 Time used: 6.71s, ROC_AUC=0.830820, ACCURACY=0.998508

We can get the best roc score when C = 0.001, which is 0.831.

For the SVC with linear kernel we also made some tests.

SVC-linear-ker: C = 1 Time used: 72.12s, ROC_AUC=0.873228, ACCURACY=0.998508

SVC-linear-ker: C = 0.001 Time used: 63.76s, ROC_AUC=0.873797, ACCURACY=0.998508

The time used for SVC with linear kernel is pretty long but the roc score is also much higher than scores of LinearSVC.

After training on small sets, we tried to train on bigger dataset. Since sklearn part of this project is finished on a macbook with small storage, the dataset used for training is about 65 GB. We choose data for every several months. The files used are of months 2012_11, 2013_05, 2014_01, 2014_11, 2015_09. The test set is still the same 2016_03. We tried tests on SVC with linear kernel, but since SVC with linear kernel has horrible time performance for large scale, the test doesn't stop in a reasonable time. So the tests are made on LinearSVC. The result is :

LinearSVC: C = 1 Time used: 4314.51s, ROC_AUC=0.838248, ACCURACY=0.998252

LinearSVC: C = 0.001 Time used: 2077.74s, ROC_AUC=0.823285, ACCURACY=0.998505

Both results with different C have reasonable performance similar to training on small dataset.

3.2. Tensorflow

Similar testing was done on the Tensorflow model that was done on the SK-Learn model. Having already realized what data sets would work best from the testing with SK-Learn, we conducted similar but slightly different tests. The hardware that we were running the Tensorflow tests on was a desktop with 8GB of RAM. Tensorflow does have some optimizations for using GPU's as well, but it doesn't support AMD, the type of graphics card in the desktop, so that was not utilized. We also had access to a Terrabyte hard drive, which allowed us to test on larger data. Surprisingly however, it didn't yield much better results.

LinearWide: 2012 Time used: ~300s, ROC_AUC=0.833686, ACCURACY=0.998508

LinearWide: 2015 Time used: ~300s, ROC_AUC=0.853677, ACCURACY=0.998508

LineaWide: 2016 Time used: ~300s, ROC_AUC=0.854809, ACCURACY=0.998508

LinearWide: all Time used: ~300s, ROC_AUC=0. 853314, ACCURACY=0.998508

As can be seen, the data set that actually gave us the best results was the 2016 training data, which was only about 50GB worth of xml, which produced about a 500MB csv file. The combined data was upwards of 200GB of xml data, and produced a 1GB csv file, but ended up

having worse results and taking far longer to process. Timing metrics are not accurate, since they are machine dependent, but in general building the models took far longer than testing them, and testing was about the same, all taking around 5 minutes to run the test.

All in all, the best result we had was an ROC_AUC score of 0.854809.

```
INFO:tensorflow:Starting evaluation at 2018-03-16-17:40:48
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from ./models\model.ckpt-854809
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Finished evaluation at 2018-03-16-17:45:32
INFO:tensorflow:Saving dict for global step 854809: accuracy = 0.99826247, accuracy_baseline = 0.9985073, auc = 0.8532993, auc_precision_recall = 0.18431994, average_loss = 0.006858366, global_step = 854809, label/mean = 0.0014926568, loss = 0.27433395, prediction/mean = 0.002251777
Results at epoch 2
------------------------------------------------------------
accuracy: 0.99826247
accuracy_baseline: 0.9985073
auc: 0.8532993
auc_precision_recall: 0.18431994
average_loss: 0.006858366
global_step: 854809
label/mean: 0.0014926568
loss: 0.27433395
prediction/mean: 0.002251777
PS D:\CS145\VandalismDetection>
```

Figure 2: *An example run of the TensorFlow program. For this particular test for the 2016 training data, we received an ROC_AUC score of 0.8532993 with an accuracy of 0.998*

4. Related Work

4.1 Human Review

One method of vandalism detection is that of human review. By having people rather than machines manually judge whether or not a revision is considered vandalism, we can determine vandalism in a way that it may be hard for a classifier to. The problem with human review is multiple. One problem with human review is that revisions will not be reviewed equally. It is unreasonable to force every user to review every page, and therefore the reviews of each page will differ page by page. Furthermore, certain vandalism cases may be ambiguous. In that situation, users may have large disagreement about how consequential the vandalism is. Finally, putting trust on the users to properly review may lead to improper review for even obvious vandalism cases, and as such machines may serve to judge vandalism more properly.

4.2 Vandalism Prevention

Instead of detecting the vandalism as it occurs, another approach is to prevent the vandalism from occurring at all. One such way of doing this, for example, is limiting who is allowed to edit. A simple change such as letting only users post can greatly decrease the amount of vandalism that occurs if a majority of vandalism is done by anonymous users. The problem with this approach is that anyone can also create an account with just a little effort. It may be possible to introduce more and more restrictions on who can edit, but that is in contention with the goal of sites such as Wikipedia, which have a goal of allowing anyone and everyone to edit their pages.

5. Conclusion

5.1. Problems Encountered

One problem we encountered right away is that of the data provided. The datasets are very large, and therefore it was quite hard to run code on them. To deal with this problem, we took only parts of certain datasets and tested our code on them first before expanding the code we tested on to larger datasets. Unfortunately, it was unavoidable to run at least once on the large dataset, and doing so took quite a long time to run.

Furthermore, because there is so much data, we had to choose which data and which parts of the data to use. Not only is the data very large, some of the categories are also not useful in our classification and therefore completely unnecessary.

After we decided which parts of the data we wanted to use, we still had to carefully choose what data structures to use. Even though we cut down on the amount of data used, there was still so much data that it is unreasonable to store all of the data one-to-one.

5.2. Final Remarks

Many problems were encountered while attempting the project, but we found a fitting solution to each of the problems encountered. Our classifier was quite weak at first, but by the end, our classifier could produce a ROC_AUC score of .854 which, although not perfect, shows that our classifier is at least competent in classifying vandalism. If our code were to be implemented to Wikidata, we are confident it would be able to classify at least a majority of actual vandalism as vandalism and reject them accordingly.

The project taught us a lot about machine learning and classification in general. If we were to continue improving on the project, we would refine our features in order to make our classifier stronger. In addition, we could experiment with different classifiers and see which one provides the best results.

Since we would have more time, we could also run our code on all of the data instead of parts of the data and just let the machine run for hours on end. This may provide a more comprehensive set to the data and could lead to a better classifier.

6. References

"TensorFlow." *TensorFlow*, www.tensorflow.org/.

"Scikit-Learn." *Scikit-Learn: Machine Learning in Python - Scikit-Learn 0.19.1 Documentation*, scikit-learn.org/stable/index.html.

Grigorev, Alexey. "Large-Scale Vandalism Detection with Linear Classifiers." arxiv.org/pdf/1712.06920.

7. Task and Workload Distribution

Feature extraction - Andy, Matt
Tensorflow model - Matt
SKLearn - Shuhao
Project Report - All