Vandalism Detection in Wikidata

CS 145 Winter 2018 Group Project

Anbo Wei	William Shao	Sabrina Chiang
UID: 604480880	UID: 004651115	UID: 204411921

ABSTRACT

Here, we present a solution to the WSDM Cup 2017's Vandalism Detection challenge, in which contestants are to design a machine learning model that can accurately predict whether a particular revision to WikiMedia's Wikidata public fact database constitutes vandalism. Said models are constructed based on provided feature sets such as the ID of the user responsible for the revision, location data related to the user, and tags relevant to each revision.

After testing various mechanisms of encoding various categorical features, we present findings of various models such a Random Forest Classifier at various hyperparameters, Linear SVC, and XGBoost. We conclude with a Random Forest Classifier set with 500 trees and a max_depth of 100 chosen from hyperparameter tuning. This model obtained an ROC-AUC score of 0.6379 during training and a ROC-AUC score of 0.6498 during testing, suggesting that our model had little to no predictive power. We then suggest further improvements that could be made to produce a better result. We attach links to our source code and instructions for execution.

Code related to this project is available at our GitHub link[0].

Table of Contents

INTRODUCTION

Wikidata is the enormous community knowledge base created by the Wikimedia Foundation, which is also responsible for the popular online site Wikipedia. The benefit to Wikidata being publicly accessible and editable by any user is the large scale at which information becomes quickly updated and available to anyone. However, the main risk of this structure of a public knowledge base is its susceptibility to vandalism, in which malicious users abuse this freedom to attempt to quickly spread misinformation on the internet. This not only has consequences for those directly seeking information from Wikidata, but also for other popular applications such as question-answering systems that obtain their results from knowledge bases such as Wikidata. With the growing amounts of data online that are impossible to manage manually, this could mean that information vandalism has the potential to affect a high volume of unknowing users without having to be manually filtered by a single actual middleman.

This presents the task of vandalism detection, a challenge offered at the WSDM Cup 2017 to be solved by machine learning and data mining. As part of the challenge, large amounts of data detailing Wikidata revisions from past years were provided, as well as information on whether those revisions were classified as vandalism or legitimate. The goal of the challenge was to design and create a machine learning classifier that could manage incoming Wikidata revisions, and in real time, determine whether each revision should be classified as vandalism. The success of such a classifier could have a direct impact on the integrity of Wikidata as a whole, as its application to the actual database in production would do a service to the online community by protecting truthful information on a scalable, automated platform.

PROBLEM DEFINITION AND FORMALIZATION

In this section we formalize the inputs and outputs of our problem by describing the format of the given data and the goal output of our work.

Data Inputs

We are given a training, validation, and test corpus consisting of Wikidata revisions, their associated metadata, and their classification of whether they are vandalism or not. The training set spans revisions from October 2012 through February 2016. The validation dataset spans revisions in March and April 2016. The test dataset consists of revisions from May 2016 through June 2016. The corpus is constructed using community feedback, i.e. revisions that are rolled back via Wikidata's rollback facility are labeled as vandalism. The creation of the corpus is described in detail in [1].

Data is separated into three types of files:

1. Raw revision data is given in XML files. A typical revision is in the following format:

```
<page>
    <title>Q6306627</title>
    <ns>0</ns>
    <id>6113077</id>
    <revision>
      <id>97494994</id>
      <parentid>97494764</parentid>
      <timestamp>2014-01-01T00:00:15Z</timestamp>
      <contributor>
        <username>Chrumps</username>
        <id>50541</id>
      </contributor>
      <comment>/* wbsetclaim-create:1||1 */[[Property:P22]]: [[Q1433355]]</comment>
      <model>wikibase-item</model>
      <format>application/json</format>
      <text> ... </text>
      <sha1>h9i4u65x1sgzx5tv3qnt0reeluc67hi</sha1>
    </revision>
</page>
```

This data includes page title and ID, and a plethora of revision information: ID, timestamp, contributor username and ID, comment, model, format, sha1, and text, which is typically a long string and is redacted for the purpose of this example. It is up to us to parse this XML into a more manageable data format and find meaningful features from these attributes.

2. Metadata associated with each revision is also given in the form of CSV files. This data is not readily available from Wikidata, and is provided by the data corpus creator. It includes geolocation data of anonymous edits as well as Wikidata revision tags. The columns and their meanings are provided in [2]:

Name	Types	Description
REVISION_ID	Integer	The Wikidata revision id
REVISION_SESSION_ID	Integer	The Wikidata revision id of the first revision in this session
USER_COUNTRY_CODE	String	Country code for IP address (only available for unregistered users)
USER_CONTINENT_CODE	String	Continent code for IP address (only available for unregistered users)
USER_TIME_ZONE	String	Time zone for IP address (only available for unregistered users)
USER_REGION_CODE	String	Region code for IP address (only available for unregistered users)
USER_CITY_NAME	String	City name for IP address (only available for unregistered users)
USER_COUNTY_NAME	String	County name for IP address (only available for unregistered users)
REVISION_TAGS	List<String>	The Wikidata revision tags

3. The remaining data is the truth data, also provided in CSV format. These files associate revision IDs with their true classification of vandalism or not. The format is provided in [2]:

Name	Types	Description
REVISION_ID	Integer	The Wikidata revision id
ROLLBACK_REVERTED	Boolean	Whether this revision was reverted via the rollback feature
UNDO_RESTORE_REVERTED	Boolean	Whether this revision was reverted via the undo/restore feature

There are a total of over 82 million revisions in the data corpus, of which about 200,000 are labeled as vandalism. We observe that the data is extremely skewed, which may prove to be a challenge during the training phase of our project, as we have relatively few positive instances from which to diagnose the characteristics that are associated with vandalism revisions.

Project Outputs

The formal objective of our project is to create software that, given revision data in the format specified above, outputs a vandalism score in the range [0,1]. We use ROC-AUC as the primary measure of evaluating our model.

RELATED WORK

In this section, we discuss the strategies of several approaches created prior to and during the WSDM competition and what makes them effective. We conclude with a comparison of these approaches with each other.

Overview of Alternate Approaches

Prior to the WDSM competition, several approaches to vandalism detection on Wikidata were developed independently, namely WDVD (Wikidata Vandalism Detection), FILTER, and ORES. Of these approaches, WDVD is the most effective. In fact, WDVD performs better than three of the five approaches submitted by participants in the WDSM competition. Performance of each of the classifiers is illustrated in a figure from [1]:

Approach	Overall performance					
	Acc	P	R	F	PR_{AUC}	ROC_{AUC}
META	0.9991	0.668	0.339	0.450	0.475	0.950
Buffaloberry	0.9991	0.682	0.264	0.380	0.458	0.947
Conkerberry	0.9990	0.675	0.099	0.173	0.352	0.937
WDVD (baseline)	0.9991	0.779	0.147	0.248	0.486	0.932
Honeyberry	0.7778	0.004	0.854	0.008	0.206	0.928
Loganberry	0.9285	0.011	0.767	0.022	0.337	0.920
Riberry	0.9950	0.103	0.483	0.170	0.174	0.894
ORES (baseline)	0.9990	0.577	0.199	0.296	0.347	0.884
FILTER (baseline)	0.9990	0.664	0.073	0.131	0.227	0.869

META is a classifier that takes the mean classification output of each of the other approaches. It performs better than any individual classifier. We observe that Buffaloberry and Conkerberry are the only submissions that perform better than the WDVD baseline. We will spend the rest of this section discussing the WDVD, Buffaloberry, and Conkerberry approaches.

WDVD

WDVD uses 47 features, of which 27 encode edit content and 20 encode edit context. The classification model consists of 16 random forest that are trained utilizing multiple-instance learning to capture consecutive edits on a page by the same user. Each random forest is built on 1/16 of the data, and each forest consists of 8 trees each with a max depth of 32 [1].

Buffaloberry

The Buffaloberry approach uses quite extensive feature engineering, based on WDVD's feature engineering, to get useful features for classification. These features are inspired by WDVD and so are also separated into content and context features.

Content features:

Revision comments are mined for properties at the character, word, and sentence level. Character level features capture the ratio of character types, with features such as upperCaseRatio and digitRatio. Word level features attempt to find meaning in the words in the revision comment. Features include badWordRatio, lowerCaseWordRatio. Sentence level features build on top of WDVD features by using fuzzy string matching.

Context features:

Context features encode user data such as isRegUser, isPrivUser, continent, country, etc. It also encodes revision features such as langLocale, affectedProperty.

The Buffaloberry approach discards any instances before May 2015. The final model selected by Buffaloberry is a single XGBoost with a maximum depth of 7200 boosting rounds.

The final ROC-AUC achieved by the classifier was 0.94702, with an online runtime of 17:11:16 to classify the test dataset.

Conkerberry

Conkerberry[4] appears to be unique to the other approaches submitted in the competition. In particular, it uses a LinearSVC and ignores the content features. To deal with the large size of the dataset, this approach discards all data from before January 2015.

Unlike Buffaloberry, the feature engineering in this approach does not base off of any prior approach. Notably, all content-based features are discarded due to RAM constraints. This means that the classifier works only using user-based and page-based features. User-based features include the metadata (country, continent, etc.) as well as a new feature that labels anonymous users and a feature called IP path which is a string consisting of each DNS subnet of the IP address associated with the revision. Comment features are mined as well.

Conkerberry represents each of these features as strings, then combines them all into one single large string feature and encodes the resulting string feature using HashingVectorizer from scikit. One single SVM is trained on the resulting sparse matrix.

Conkerberry achieved an ROC-AUC of 0.93708 with an online runtime of 02:47:50 for the test dataset. This runtime is significant faster than the other models without sacrificing much ROC-AUC.

A Comparative Conclusion

WDVD and Buffaloberry appear to be closely related. The feature engineering done in the Buffaloberry approach is an improved version of the feature engineering done in WDVD. They differ only in model training. WDVD uses a random forest and Buffaloberry uses XGBoost, which is a specific way of training a tree ensemble. So these approaches are actually extremely similar, with Buffaloberry being an iterative improvement of WDVD.

Conkerberry, on the other hand, is wildly different from these other two approaches. Its feature engineering discards the majority of the feature engineering work done by WDVD, choosing to ignore content features entirely. It shows that a powerful classifier can be built considering only the page title and user features. It also does not use the forest/tree approach, opting for a support vector machine.

Buffaloberry and Conkerberry have comparable performance, with Buffaloberry having a better ROC-AUC at the cost of higher prediction time cost. Buffaloberry uses around twice as many features as Conkerberry. We can attribute the ROC-AUC difference largely to Buffaloberry performing much more feature engineering, and thus capturing more of the attributes associated with vandalism.

METHODS DESCRIPTION

This section describes the process through which we approached the task, including our setup and an overview of the steps taken in building our classifier.

Environment

Since the task involved processing several hundred gigabytes of revision data from Wikidata history, we set up a Google Cloud instance to download and expand the compressed training and validation files provided by WSDM Cup 2017. We chose a 2 terabyte HDD to hold our data and selected the n1-highmem-8 machine type, which has 8 virtual CPUs and 52 GB of memory to accommodate loading our data into RAM. We ran Ubuntu 16.04. This cloud instance was used for all experimentation and data processing, and we connected it to our GitHub project where we used Python 2.7 for our development stack.

Tools we use in our coding of our model include `numpy` for executing calculations and `feather` for parsing data files into formats for simpler processing. We make use of `scikit-learn` for the bulk of machine learning used by our model, which is used for easily building and manipulating classifiers for our data. Additionally, we use `pandas` for the DataFrame type used to represent our data in-memory during processing and classification. We use the `cPickle` module to store binary representations of Python objects so that we can use them across Python runtimes.

Data Processing Pipeline

This section describes the steps taken to transform the starting XML and CSV files into the final format that we use for training and classification. It is based off of the data parsing pipeline provided by the Conkerberry source code[5].

1. Starting with the files given by WDSM, we first run convert_xmls_to_csv.py. This parses the XML files and encodes the information in each tag under its own column for each reivsion, and saves the resulting CSV files.
2. We run merge_csvs.py, which combines the CSV files resulting from step 1 with the information we want from the meta CSV files and the truth CSV files provided. We save

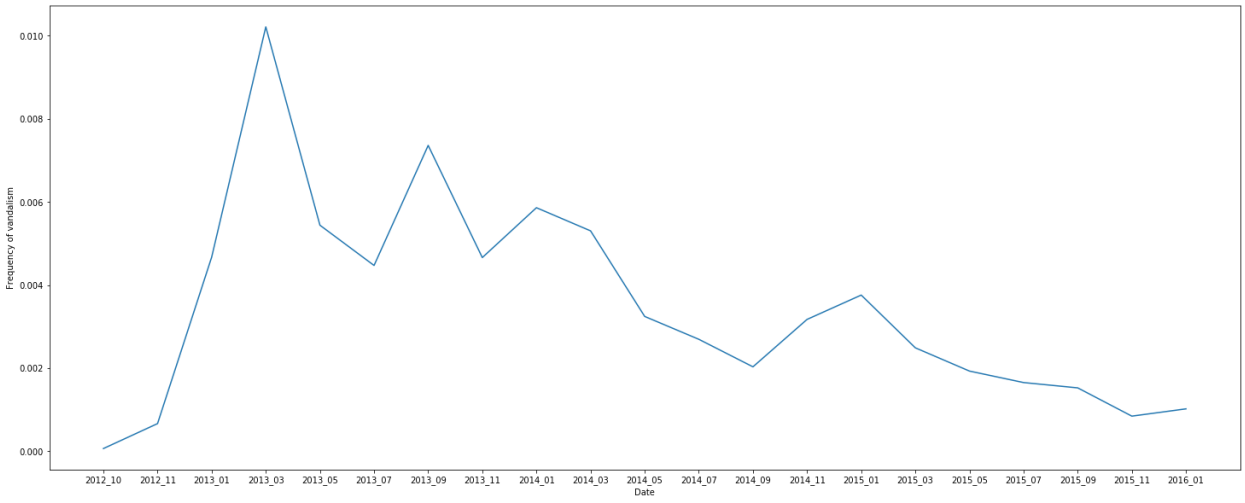
these in pandas DataFrames and export in .feather files, which allow us to load them up as DataFrames again in future Python runtimes.

3. We run create_final_features.py which takes the resulting .feather files and adds on features that will be described in our feature engineering section. It combines the separate .feather files from the previous step and merges them into one large .feather file containing all data. It also splits the data into training, validation, and test set by adding a feature called fold.
4. At this point, the .feather files hold the features in their original representation. Many of them are strings, but we need to run them through classifiers that only accept numerical data. We run create_encoded_features.py which applies `sklearn`'s LabelEncoder to map each of the string features to numbers.
5. Up to this point none of the test data has been converted. A single script, process_training_csvs.py, takes the test data from the CSV files produced by step 1 to the same .feather format as the training data in step 4.

Data Exploration

After running step 2, we open up the training data and observe some of its properties to try to get an idea of what kind of features will be important in classifying vandalism. This proves to be a challenge due to the extreme skew of the data and the large data size. Regardless, we include and discuss some visualizations generated from this step.

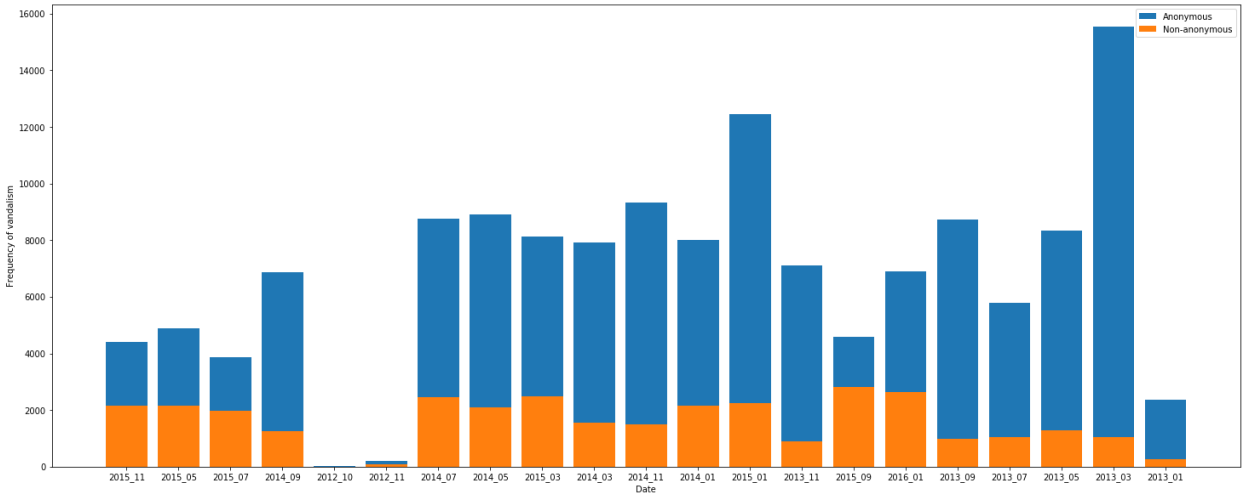
To try to see if there is a time pattern of vandalism, we plot the vandalism frequency in each file.



There is a large spike of vandalism in 2013 but the rate gradually decreases after that with small spikes. January 2015 is the last large spike, after which it decreases. This leads us to believe that

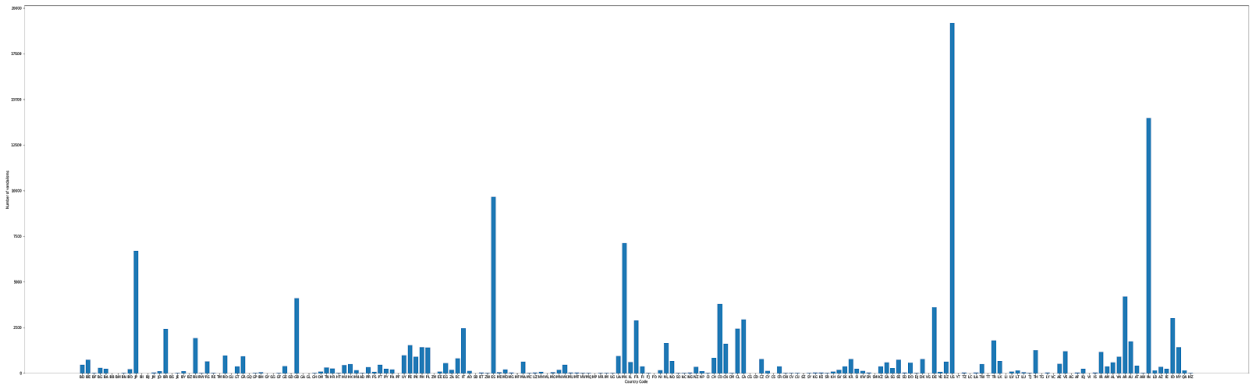
the vandalism occuring after this time may have some different properties than the vandalism that occured before. As a result, we prune our training data to March 2015 onwards.

Next we observe the ratio of anonymous to non-anonymous vandalisms over time.



There is no apparent pattern over time, but the ratio of non-anonymous and anonymous ratio shows that it is mostly anonymous users that commit vandalism. This indicates that the anonymity of a user may be a useful feature in predicting vandalism.

We plot the country code vs frequency of vandalism.



Certain countries have frequency in this graph but these are not relative frequencies, so this information is not particularly useful.

Some other graphs are generated but not shown due to their lack of pertinent information.

Feature Engineering

In this section we explore the methods by which we extracted features from the original data, and enumerate and justify the features we select for our classification model.

The initial features that we scrape from the XML files are:

Feature	Description
page_title	The title of the page being revised. This information is encoded in page_id, so we later remove this.
page_ns	Page namespace. This entry is uniformly 0 for all pages, so we later remove this.
page_title	The title of the page being revised. This information is encoded in page_id, so we later remove this.
page_ns	Page namespace. This entry is uniformly 0 for all pages, so we later remove this.
page_id	Unique ID associated with each page. Later discarded because from the Conkerberry paper we believe page features to not be important.
revision_id	Unique ID associated with each revision. We later remove this because it is unique for each instance and therefore not useful in classifying vandalism.
revision_timestamp	Timestamp of the revision. This is not considered for a feature since we don't believe the date to be a useful indicator of vandalism (from our data exploration). Also the test dataset comes from all later timestamps than the training data, so this will not be useful in training our classifier.
revision_comment	A string that indicates some automatically generated properties of the revision as well as some user input. We save the entire string initially and later parse some attributes d remove the original full string. We do this because we want to apply an encoder that maps categorical values to numbers, and revision_comment has too many unique entries to do is efficiently.
revision_model	Model of the revision. Uniform for the entire dataset, so removed.
revision_format	Format of the revision. Uniformly 'application/json' for the entire dataset, so we remove it.
revision_count	Number of consecutive revisions to the page. We remove this later because our data exploration shows that it is not very useful.

username	Username of the contributor. This information is encoded in user_id, so we later remove it.
user_id	Unique ID number associated with each contributor.
ip_address	IP address of the contributor. Later removed in favor of ip_prefix.

Due to the size of storing the information in the <text> tags and the choice of Conkerberry to discard this while still being effective, we also choose to discard the <text> tags. In effect, this means we also do not consider the content features described by WDVD and Buffaloberry.

After we merge the revision CSV with the meta and truth CSVs, we add the features stored in the meta CSV file. The resulting additional features are:

Feature	Description
USER_CONTINENT_CODE	A two character representation of the user's continent.
USER_TIME_ZONE	Time zone of the user.
USER_REGION_CODE	Region of the user.
USER_CITY_NAME	City of the user.
USER_COUNTY_NAME	County of the user.
USER_COUNTRY_CODE	Two character representation of the user's country.
REVISION_TAGS	Wikidata revision tag in its original string form.

We generate two new features on our own:

Feature	Description
ip_prefix	The first two numbers in the ip_address feature, e.g. an ip_address of 99.98.0.0 results in an ip_prefix of 99.98. We use this and discard the ip_address feature because we believe the first two digits may encode some information shared by multiple instances, but the last two digits make the IP addresses unique, and so are less useful as a feature.
anon	Has value 1 for anonymous users and -1 for logged in users. This feature will further encode whether users are logged in (in addition to meta features) since our data exploration led us to believe that this is an important feature in predicting vandalism.

In lieu of full revision comments, we remove the revision_comment feature and instead create two new features from it:

Feature	Description
revision_comment_category	Action type of the revision. We believe the action in the property is related to whether a comment is vandalism.
revision_comment_property	Part of the comment string of the form Property:(.+) that encodes something about the revision we believe may be useful.

The final set of 12 features is:
'USER_COUNTY_NAME','USER_COUNTRY_CODE','USER_CONTINENT_CODE', 'USER_TIME_ZONE','USER_REGION_CODE','USER_CITY_NAME', 'REVISION_TAGS', 'ip_prefix', 'revision_comment_category', 'revision_comment_property', ‘anon’, and ‘user_id’.

Each of these can be considered categorical data, but the inputs to our training algorithms require numerical data. As a result, we use LabelEncoder from `sklearn.preprocessing`, which maps a list of strings to ordinal integer values. Thus our final .feather dataframe consists of each of the above features containing purely numerical data.

Training

We used available `scikit-learn` libraries and XGBoost as a base for creating a classifier to process the training data, since it is a useful tool for handling major classifier algorithms and makes it easy to tweak the parameters as needed to fit our data. The training algorithms of high interest include Linear SVM, and Random Forests, particularly because they were included among the methods tried by high-performing participants at the WSDM Cup 2017. The Linear SVM and Random Forest approaches were available through `scikit-learn`, and XGBoost provides a library of its own, so these classifying methods were relatively simple to implement for our data.

We heavily used the Conkerberry paper[4] as a point of reference throughout this project, which primarily classifies using Linear SVM. Other sources of methods from high performing submissions to WSDM Cup 2017 include Buffaloberry[3] which makes use of XGBoost as well as Multiple-Instance. Several other submissions including WDVD implemented Random Forest, which uses a network of decision trees for thoroughness while avoiding overfitting. Our goal was to implement a simpler, yet effective version of the methods from these reference projects by training the processed data and evaluating various methods and parameters against each other to obtain the optimal value.

EXPERIMENT DESIGN AND EVALUATION

Experiment Design

To create a basis of comparison, we first wrote a rough classifier whose output was always negative - meaning, the result of every revision would always output as not-vandalism. Our justification for this was to create a “low bar” for result comparison, and since revisions labeled vandalism were a much smaller fraction of the training data than those not labeled vandalism (approximately 0.2% of training data was labeled vandalism), an all-negative classifier would in theory be the preferred baseline in that it would be more rudimentarily accurate than an all-positive classifier. Since ROC-AUC calculations are based on percentages of false positives and false negatives, the ROC-AUC result for the all-negative classifier was 0.5. To clarify, this score was calculated while in development of the classifier, before applying to the separated test data and only using training and validation sets. This helped establish that although the true goal was to achieve an ROC-AUC result of as close to 1 as possible, that scoring below 0.5 would indicate something had truly gone wrong.

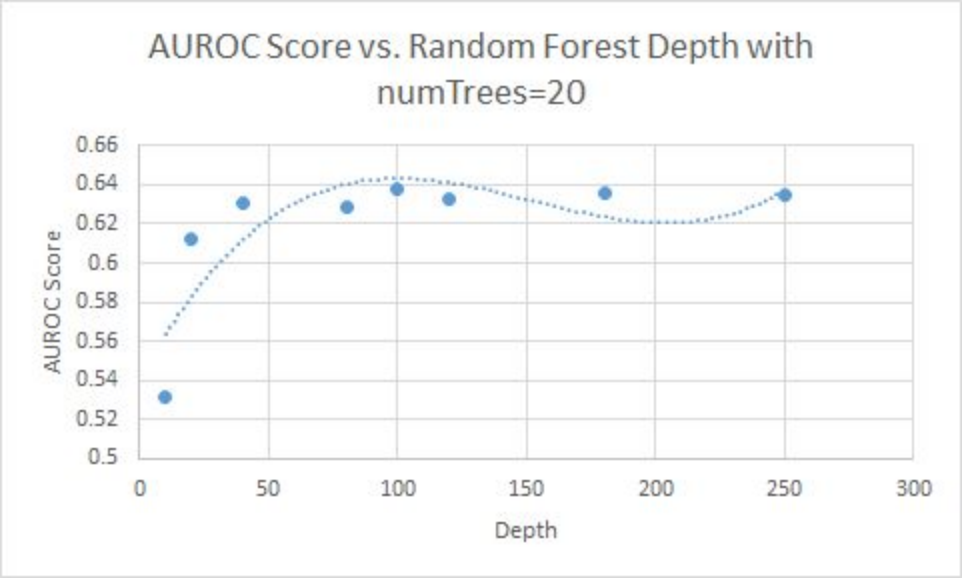
Luckily, classifiers we tested did not score below 0.5, but still had lots of room to improve upon each other. Initially, we designed a LinearSVC classifier using `scikit-learn`, based on the Conkerberry approach. However, after implementing and testing a LinearSVC classifier, we

could not achieve an ROC-AUC score of any significant amount over 0.5. This led us to conclude that the the user-based featureset we had chosen and parsed was not easily linearly separable.

Our next step was to implement a simple Random Forest classifier, also using `scikit-learn`. To begin, we tried a smaller classifier that only used 10 decision trees and a depth of 20. This small classifier resulted in an ROC-AUC of approximately 0.60489, improving upon the LinearSVC classifier. Increasing only parameter for number of trees in the random forest to 500 improved the ROC-AUC score again to 0.61309.

Before exploring that method further, we compared these results to an XGBoost trainer, using a hash matrix of 500,000 features and leaving most features as typical defaults. However, with a low ROC-AUC result of 0.5, we decided that even though there were parameters possible to tweak in XGBoost and LinearSVM, that the highest effectiveness we could have with parameter tuning would be by pursuing Random Forest classifiers. Many of these shortcomings are due to the way that we encoded our data, which will be further discussed in the evaluation section.

Increasing the number of trees in the forest proved to be more reliable, but to tune the optimal tree depth, we recorded results for random forests of 20 trees each and varying depths. Results of this tuning can be seen in the following plot:



Based on these numbers, we selected a tree depth of 100 and then proceeded to train our final classifier of forest size 500.

Evaluation

Our final Random Forest classifier of depth 100 and size 500 was applied to the test data and resulted in an ROC-AUC score of 0.649836579347, giving our highest performing score yet. This result was generated on our cloud instance running 7 cores and took 239.2 minutes to train.

As an aside, it is worth noting that the baseline ROC-AUC scores were computed on a TIRA test platform while ours were computed offline. Submissions to the WDSM Cup had to create a client that would receive the data in a moving window of 16 revisions at a type, so as to not allow classifiers to see the entire future before returning a classification. This was done to emulate a real deployed vandalism detector. For our evaluation, however, we evaluate on the entirety of the test data offline. This likely does not make a difference for our resulting metric, though, because we do not use any time data or pairwise relationships between the instances. We consider each instance in isolation and perform a prediction one by one.

Compared to the baselines WDVD, Buffaloberry, and Conkerberry, our ROC-AUC is much worse. Here we will explain the reason for the discrepancy.

First, the feature engineering we perform is weaker than the feature engineering performed by the other submissions and WDVD. We have only 12 features in the end, while the others can have upwards of 50 features. Due to the enormous size of the data and our time/processing power constraints, it is difficult to mine and store a huge number of features, especially those encoding large string attributes. We also relied too heavily on Conkerberry's result to justify discarding many features. As a result, we train and base our predictions on much less information than the baselines and so have weaker discrimination between instances.

Continuing, our selection of random forest model should not be problematic, as WDVD and Buffaloberry are also built on forest-based approaches. However, our particular use of the random forest is not effective because we encode our data with `scikit`'s LabelEncoder. This maps each label in a string feature to a numerical value counting from 0 upward. The classifiers implemented by `sklearn` assumes the features are all continuous, but our numbers really represent categories. The RandomForestClassifier makes splits in its tree based on the continuous value assumption, so each split picks an attribute and values at which to split the data. This is not useful for the way our information is encoded, as having a value greater or less than any value for a feature does not indicate any information--all values actually represent categories. Thus the RandomForestClassifier is trained in a suboptimal way.

The LabelEncoder encoding of our data also explains why the LinearSVC is ineffective for our features. Because our categorical values are used in training as if they are continuous, this makes the LinearSVC quite useless, as support vector machines try to find a boundary between the positive and negative instances that linearly separates them. In the feature space generated by our

LabelEncoder encoding, it is likely that positive and negative instances are highly interspersed, and no decision boundary can be found. This explains why the classifier is unable to get an ROC-AUC higher than 0.5.

The poor performance of the XGBoost model can also be explained along the same lines of reasoning. Because while inducing its internal CART trees, it assumes values are continuous, it is unable to make meaningful splits on our actually categorical data.

In the conclusion, we discuss improvements and proposals for future work on the task.

CONCLUSION

This paper details the purpose of the Wikidata Vandalism Detection task, the models from the WSDM Cup 2017 that we researched to determine a basis for our model, and the classifier construction process for our results. The need for machine learning models to handle vandalism detection is derived from the scale at which information in Wikidata grows, and the unfortunate reality that with high volumes, manual inspection becomes more impossible as a standard. Under a simplified model that primarily analyzes user features, we attempted classifiers based on approaches like the Conkerberry paper, including Linear SVM, Random Forest, and XGBoost. After careful parameter tuning, our final classifier was a 500 tree, max depth-100 Random Forest model that resulted in an ROC-AUC score of about 0.6498.

Though our model was merely a simplified version of the high-performing WSDM Cup 2017 participant models, throughout this classifier construction process we were able to note on potential improvements to make on this task for the future. Due to the fact that we were only able to work at a limited scale and limited processing power, we opted to ignore content features entirely and focus on features surrounding the revision user. Though this proved to be effective in the Conkerberry approach, experience with manual detection speaks to how effective looking directly at revision content can give clues to vandalism. Being able to effectively analyze content features, though too complex for our model, would likely be a benefit to the model's success rate.

With more processing power, it would also be possible to focus on relationships between the different instances of revisions rather than their individual vandalism probability. This would be considered a deeper context feature than those we did get to analyze; for example, it would be useful to know whether a preceding revision was written by the same user in the same session, and was also categorized as vandalism. Deeper context clues could contribute to higher model accuracy, but would have to be processed by a more complicated classifier.

On a data parsing level, one idea that we were unable to implement was using one-hot encoding on the feature matrix. This could alleviate the issue we had with LinearSVC, where

interpretations of categorical data potentially caused the resulting dataset to not be linearly separable. If one-hot encoding was applied to the feature matrix, this discrepancy may be more easily clarified and it would be more possible to apply a linear model.

REFERENCES

Github to Project

[0] Final project code: http://github.com/microsizeMe/cs145project/

Paper References

[1] Overview of the Wikidata Vandalism Detection Task at WSDM Cup 2017: https://arxiv.org/pdf/1712.05956.pdf

[2] WSDM Cup 2017 - Vandalism Detection Task: http://www.wsdm-cup-2017.org/vandalism-detection.html

[3] Buffaloberry Paper: https://arxiv.org/pdf/1712.06919.pdf

[4] Conkerberry Paper: https://arxiv.org/ftp/arxiv/papers/1712/1712.06920.pdf

[5] Conkerberry source code: https://github.com/wsdm-cup-2017/conkerberry/

Listed Dependencies

Package	Version	Link
Backports-Abc	0.5	https://pypi.python.org/pypi/backports-abc
Backports.Functools-Lru-Cache	1.5	https://pypi.python.org/pypi/backports.functools-lru-cache
Backports.Shutil-Get-Terminal-Size	1.0.0	https://pypi.python.org/pypi/backports.shutil-get-terminal-size
Bleach	2.1.3	https://pypi.python.org/pypi/bleach
Boto	2.38.0	https://pypi.python.org/pypi/boto
Chardet	2.3.0	https://pypi.python.org/pypi/chardet
Configparser	3.5.0	https://pypi.python.org/pypi/configparser
Crcmod	1.7	https://pypi.python.org/pypi/crcmod

Cycler	0.10.0	https://pypi.python.org/pypi/cycler
Decorator	4.2.1	https://pypi.python.org/pypi/decorator
Entrypoints	0.2.3	https://pypi.python.org/pypi/entrypoints
Enum34	1.1.6	https://pypi.python.org/pypi/enum34
Feather-Format	0.4.0	https://pypi.python.org/pypi/feather-format
Functools32	3.2.3.post2	https://pypi.python.org/pypi/functools32
Futures	3.2.0	https://pypi.python.org/pypi/futures
Google-Compute-Engine	2.7.5	https://pypi.python.org/pypi/google-compute-engine
Gyp	0.1	https://pypi.python.org/pypi/gyp
Html5lib	1.0.1	https://pypi.python.org/pypi/html5lib
Ipykernel	4.8.2	https://pypi.python.org/pypi/ipykernel
Ipython	5.5.0	https://pypi.python.org/pypi/ipython
Ipython-Genutils	0.2.0	https://pypi.python.org/pypi/ipython-genutils
Ipywidgets	7.1.2	https://pypi.python.org/pypi/ipywidgets
Jinja2	2.1	https://pypi.python.org/pypi/Jinja2
Jsonschema	2.6.0	https://pypi.python.org/pypi/jsonschema
Jupyter	1.0.0	https://pypi.python.org/pypi/jupyter
Jupyter-Client	5.2.3	https://pypi.python.org/pypi/jupyter-client
Jupyter-Console	5.2.0	https://pypi.python.org/pypi/jupyter-console
Jupyter-Core	4.4.0	https://pypi.python.org/pypi/jupyter-core
Kiwisolver	1.0.1	https://pypi.python.org/pypi/kiwisolver
Lxml	4.2.0	https://pypi.python.org/pypi/lxml
Markupsafe	1	https://pypi.python.org/pypi/MarkupSafe
Matplotlib	2.2.0	https://pypi.python.org/pypi/matplotlib

Mistune	0.8.3	https://pypi.python.org/pypi/mistune
Nbconvert	5.3.1	https://pypi.python.org/pypi/nbconvert
Nbformat	4.4.0	https://pypi.python.org/pypi/nbformat
Notebook	5.4.0	https://pypi.python.org/pypi/notebook
Numpy	1.14.2	https://pypi.python.org/pypi/numpy
Pandas	0.22.0	https://pypi.python.org/pypi/pandas
Pandocfilters	1.4.2	https://pypi.python.org/pypi/pandocfilters
Pathlib2	2.3.0	https://pypi.python.org/pypi/pathlib2
Pexpect	4.4.0	https://pypi.python.org/pypi/pexpect
Pickleshare	0.7.4	https://pypi.python.org/pypi/pickleshare
Prompt-Toolkit	1.0.15	https://pypi.python.org/pypi/prompt-toolkit
Psutil	5.4.3	https://pypi.python.org/pypi/psutil
Ptyprocess	0.5.2	https://pypi.python.org/pypi/ptyprocess
Pyarrow	0.8.0	https://pypi.python.org/pypi/pyarrow
Pygments	2.2.0	https://pypi.python.org/pypi/Pygments
Pyparsing	2.2.0	https://pypi.python.org/pypi/pyparsing
Python-Dateutil	2.7.0	https://pypi.python.org/pypi/python-dateutil
Pytz	2018.3	https://pypi.python.org/pypi/pytz
Pyzmq	17.0.0	https://pypi.python.org/pypi/pyzmq
Qtconsole	4.3.1	https://pypi.python.org/pypi/qtconsole
Requests	2.9.1	https://pypi.python.org/pypi/requests
Scandir	1.7	https://pypi.python.org/pypi/scandir
Scikit-Learn	0.19.1	https://pypi.python.org/pypi/scikit-learn
Scipy	1.0.0	https://pypi.python.org/pypi/scipy
Send2trash	1.5.0	https://pypi.python.org/pypi/Send2Trash

Simplegeneric	0.8.1	https://pypi.python.org/pypi/simplegeneric
Singledispatch	3.4.0.3	https://pypi.python.org/pypi/singledispatch
Six	1.10.0	https://pypi.python.org/pypi/six
Subprocess32	3.2.7	https://pypi.python.org/pypi/subprocess32
Terminado	0.8.1	https://pypi.python.org/pypi/terminado
Testpath	0.3.1	https://pypi.python.org/pypi/testpath
Tornado	5	https://pypi.python.org/pypi/tornado
Tqdm	4.19.7	https://pypi.python.org/pypi/tqdm
Traitlets	4.3.2	https://pypi.python.org/pypi/traitlets
Unicodecsv	0.14.1	https://pypi.python.org/pypi/unicodecsv