News Text Classification

Definition

Overview

Text data is ever increasing on the internet. Everyday more and more data is added to the internet in the form of text, whether it be in the form of product reviews on Amazon, a news story uploaded to your local news station's website, an email sent to your inbox, or even this report itself.

In this project, we created a text classifier capable of categorizing news articles into one of twenty categories. This project is a proof of concept that with enough data, it is possible to categorize it into given topics.

Problem Statement

The goal of this project is to classify bodies of text into their relevant categories. To do so we must:

1) Download the 20 NewsGroup dataset from Scikit-Learn's `datasets` submodule.
2) Vectorize the text in both the testing and training data.
3) Encode the features in both the testing and training data.
4) Train several classifiers
5) Identify the best classifier through Grid Search Cross Validation
6) Produce graphs of the results of the testing dataset
7) Examine the effectiveness of this model with a Chrome Extension built specifically to classify the text on a webpage.

The final classifier should be capable of providing an accurate article prediction based on the given data.

Metrics

Four different metrics were used to describe the effectiveness of each classifier. Accuracy, precision, recall, and f1-score. **Accuracy** was used in determining the overall best classifier when using Grid Search Cross Validation. Accuracy is described as the sum of true positives and true negatives divided by the size of the dataset.

$$Accuracy = (True\ Positives + True\ Negatives) / Dataset\ Size$$

Precision and **recall** are used in the calculations of f1-scores for each category. Precision is described as the number of true positives divided by the sum of the true positives and false

positives. Recall is described as the number of true positives divided by the sum of the true positives and false negatives.

$$Precision = True\ Positives / (True\ Positives + False\ Positives)$$
$$Recall = True\ Positives / (True\ Positives + False\ Negatives)$$

F1-score is the *harmonic mean* of precision and recall. It can be effectively used as an accuracy score for each category. F1-score is given by the following formula:

$$F1 - score = 2 * ((Precision * Recall) / (Precision + Recall)$$

Overall, accuracy of the model as a whole is an important metric on its own, however, it's also important to look at the f1-score to see where the classifier is failing. The accuracy of the model can be high, but completely miss the mark for some categories.

Analysis and Methodology

Data

The 20NewsGroup dataset can be downloaded through Scikit-Learn's `datasets` submodule, with the `fetch_20newsgroups`[1] class. The data is already split into testing and training set and can be loaded separately. The data contains twenty classes, or categories, and it contains almost 20,000 samples. The samples are spread relatively evenly across each class, as can be seen in **Figure 1**. The data is in the form of texts, with the categories being integers. The shape of our data can be found with `trainingData.filenames.shape` and `testData.filenames.shape`, the shapes are found to be `(11314,)` and `(7532,)` respectively. To get information from either set, we need to turn the text to numbers and the integer categories to text form.

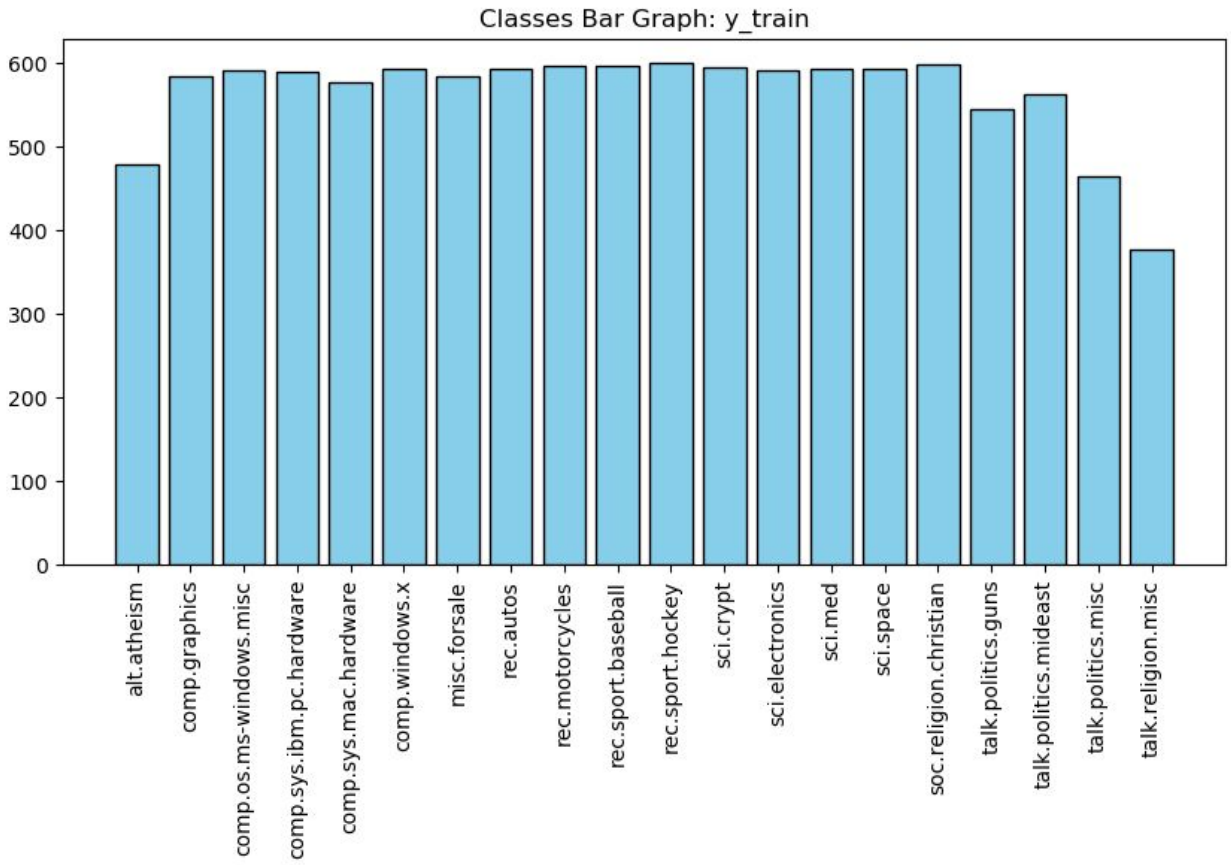
Preprocessing

One problem with the current data being given by `fetch_20newsgroups` is that the data is a block of text. Text cannot be fed to a classifier, so it needs to be vectorized. Vectorization of text in this project was done through Term Frequency, Inverse Document Frequency Vectorization or **TF-IDF Vectorization**. TF-IDF is calculated by taking the term's frequency in a document and multiplying it by the log of the number of documents divided by the number of documents that actually contain the term. The formula for TF-IDF is given by the following formula:

[1] scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_20newsgroups

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

Fig. 1 A basic bar graph displaying a relatively balanced distribution of the classes across the samples in the `y_train` subset.



TF-IDF Vectorization can easily be done through Sci-Kit Learn, by importing the `TFIDFVectorizer`[2] class from the `feature_extraction.text` submodule. Once imported, it can be fitted with the texts from the samples and can be used to transform text for predictions.

Another problem, although not initially apparent, is that the classes given by the data are integers rather than strings, which is good for feeding into the classifier, but is not good, however, for displaying tangible predictions. To understand what the classifier is actually classifying a block of text as, we must convert the integer to its respective "string" format. For this project, this was done through the `LabelEncoder`[3] class from the `preprocessing` submodule. Once fitted with the classes in the dataset, the

[2] scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer
[3] scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder

`LabelEncoder` can be used to change the integer values of the predictions to their respective class through the `inverse_transform` method.

Techniques and Algorithms

This project is a classification problem, so it makes sense to use models built for classification. For this project, four classifiers were chosen: `Bernoulli Naive Bayes`[4], `Multinomial Naive Bayes`[5], `Linear Support Vector Machine Classifier`[6], and `Stochastic Gradient Descent Classifier`[7]. All classifiers were imported from Scikit-Learn.

We chose one classifier that we believed would do rather poorly, for the sake of confirming that it would not do as well as the other models we chose. Bernoulli Naive Bayes is a classifier that works well with binary choices, either a 0 or a 1, however, since there are twenty labels, Bernoulli should perform poorly. Bernoulli Naive Bayes was chosen for this reason, as we believed that it would get closely related topics confused, but would do reasonably well with categories that are vastly different. Multinomial Naive Bayes was chosen as a better alternative to the Bernoulli Naive Bayes. Multinomial Naive Bayes works better with our data since it treats the features as event probabilities, so it should perform much better than the Bernoulli alternative.

The other two classifiers we chose, Linear Support Vector Machine (**LinearSVC**) and Stochastic Gradient Descent (**SGDClassifier**) should perform very similarly, as they both operate under the same premise. Both algorithms use hyperplanes and loss based functions to separate the classes. Both should be relatively similar in speed as well, although speed is not being tested in this project.

Refinement

Both Naive Bayes models were left at their default state, as their default values were suitable for our data and would not benefit from tuning. However, both our Linear Support Vector and Stochastic Gradient Descent classifiers were tuned using 3-fold `GridSearch Cross Validation`[8]. For LinearSVC, tuning was done to the *loss, C, dual,* and *penalty* parameters. For SGDClassifier, tuning was done to the *loss, alpha,* and *epsilon* parameters.

---

[4] scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB
[5] scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB
[6] scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC
[7] scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier
[8] scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV

Figures 2 and 3 display the hyperparameters used in the Grid Search. Every *loss* function was tested, with many performing similarly, and some performing terribly. The parameter grid was created based on parameters that we thought would affect the classifier the most, whether it be a positive or negative effect.

```
paramGridSVC = [
    {
        'loss' : ['hinge', 'squared_hinge'],
        'C': [0.01, 0.1, 1, 10, 100]
    },
    {
        'dual': [False],
        'penalty': ['l1'],
        'C': [0.1, 1, 10, 100]
    }
]
```

```
paramGridSGDClassifier = [
    {
        'loss': ['hinge', 'log', 'modified_huber',
            'squared_hinge', 'perceptron', 'squared_loss'],
        'alpha': [0.000001, 0.00001, 0.0001, 0.001]
    },
    {
        'loss' : ['epsilon_insensitive'],
        'epsilon' : [0.001, 0.0001, 0.00001],
        'alpha': [0.000001, 0.00001, 0.0001, 0.001]
    }
]
```

Fig. 2 and 3 Parameter grids for the LinearSVC and SGDClassifier.

The final parameters were chosen by the `best_param_` attribute of the Grid Search and were fed into our classifiers. For LinearSVC, the best parameters were determined to be *loss: squared_hinge* and *C: 1.* For SGDClassifier, we found *loss: epsilon_insensitive, alpha: 0.0001,* and *epsilon: 0.00001* to be the best parameters.

Results

Models

Our models performed about how we expected, with Bernoulli Naive Bayes performing the worst, with extreme confusion in some labels as seen in **Figure 4**, while the other three classifiers performed reasonably well, with Linear Support Vector Machine and Stochastic Gradient Descent performing the best.

In **Figures 5**, **6**, and **7**, the best models are shown. Although each perform rather well, there are still some areas with medium to high confusion, specifically in the labels related to hardware or religion. For example, talk.religion.misc and soc.religion.christian are similar, same with talk.politics.misc and talk.politics.guns. **Figure 8** displays the precision, recall, and f1-scores for the bottom right corner of **Figure 6**, or the LinearSVC.

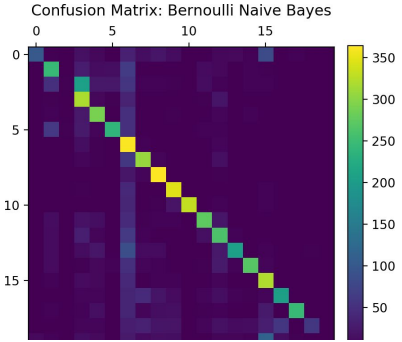
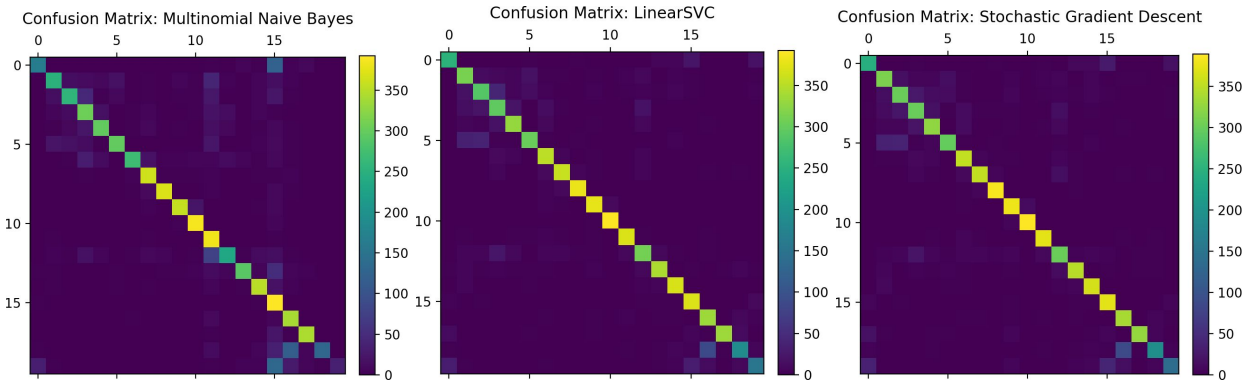


Fig. 4 A confusion matrix for `Bernoulli Naive Bayes`, displaying a high rate of confusion in related labels.



(Left to Right) **Fig. 5** depicts the confusion matrix for the Multinomial Naive Bayes classifier. There is some confusion in the bottom right with the political/religious labels but otherwise it performed rather well. **Fig. 6 and 7** display the confusion matrix for LinearSVC and SGDClassifier respectively. Both these figures are almost identical in their results

soc.religion.christian	0.53	0.80	0.64
talk.politics.guns	0.74	0.57	0.64
talk.politics.mideast	0.96	0.65	0.78
talk.politics.misc	0.89	0.19	0.31
talk.religion.misc	1.00	0.00	0.01

Fig. 8 The label, precision, recall, and f1-scores for the last five rows of LinearSVC, depicting common areas of confusion across all the classifiers tested.

Furthermore, **Table I.** can be used to quickly see where each classifier fails and where it succeeds. The large amount of misclassifications brings the recall scores of both Naive Bayes Classifiers down by quite a margin.

Weighted Average	*Precision*	*Recall*	*f1-score*
Bernoulli Naive Bayes	0.71	0.63	0.61
Multinomial Naive Bayes	0.82	0.77	0.77
Linear SVC	0.85	0.85	0.85

Stochastic Gradient Descent	0.85	0.85	0.85

Table I. displays the weighted averages of the scorings for each classifier, given by the `classification_report`[9] of Scikit-Learn.

Chrome Extension

There's multiple ways to connect a desktop application to the web. A browser testing tool such as Selenium is one way to connect to it. However, Selenium suffers with the limitation that it imitates how a human would use the browser. Another way would be to write a bash script that uses cURL to fetch the page, remove all but the text, and feed it to our text classifier, and certainly it's a good solution though a bit tedious on traditional desktop systems, but efficient for those who use a Windows manager for a mouse-less experience.

The other way would be to tie down the script to a certain browser family, by creating a browser extension. Generally, this will be the most accessible option, however, power users will not be able to appreciate it, if they're using more 'input efficient' browsers such as qutebrowser, or anything that doesn't follow a certain extension standard we have followed. Due to limited time, we only decided to make browser applets, although they were commonly called as Chrome extensions. With time, major browsers including Firefox, have decided to follow the same standards as Chrome for developing extensions. However, as no testing was done on any other browser, we'll refer to this as the Chrome extension.

There's multiple ways to get a machine learning model work with a Chrome extension. One needs to remember that browsers are a sandbox, and the only way to interact with this sandbox is through Javascript. Our project, however, uses Python with Scikit-Learn, thus the challenge was to somehow make our program interact with the browser. After looking around, we found multiple ways to get this to work.

One way would be to transpile our extension from Python to Javascript. There's many free and open source transpilers out there that claim to work. However, they present many problems. First, Scikit-Learn is a complicated machine learning library and these transpilers might not be able to transpile it properly. A big worry we had while working on this was silent errors. The transpiler might have transpiled the code, but a transpiler cannot guarantee that the transpiled code is correct. Coupled with Javascript's nature as a weakly typed dynamic language, we don't even have quality static analysis to do even a

[9] scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report

simple check. Even if the transpiler gave us errors regarding the presence of sections of code that was not transpiled properly, we're not confident in converting code in such a complicated library. There's also the issue of it significantly slowing down the browser, as the transpiled program would consume a large amount of resources. On weak systems, such as a Chromebook, we would most likely cause a kernel-lock, which would be disastrous.

Another way would be to transpile Python to WebAssembly, which would avoid the issue of a slow browser, but it would have its own share of security vulnerabilities as a new technology, and we also lack the skill and time to work on it.

The easiest and most effective way would be to make our text classifier as a server side program. While, in languages such as C++ where this would be quite an annoying task, even with boost library collection, Python doesn't face the same problem. Python has two major libraries/web framework, namely, Flask and Django, that is easily able to solve our problem. We decided to use Flask over Django due to it's lightweight nature, plus ease of use.

This works by starting a Flask app, while our model is trained, and wrapping a query around the predict function. The Chrome extension can now talk to this endpoint we have exposed by sending a GET request. Chrome extensions are quite easy to develop, although Chrome has its own share of weird functions that are only available to being used inside Chrome and Chromium respectively. While, this is not a big problem, since we are developing a chrome extension, we still decided to focus on portability, and used no chrome-only Javascript function. Our extension works by grabbing the selection text in the webpage we are currently at, storing it as a string, and then sending a GET request to the Flask web app. **Figure 9** depicts a working example of our extension, wherein we analyze a page on Craigslist and return the predicted label for the page.

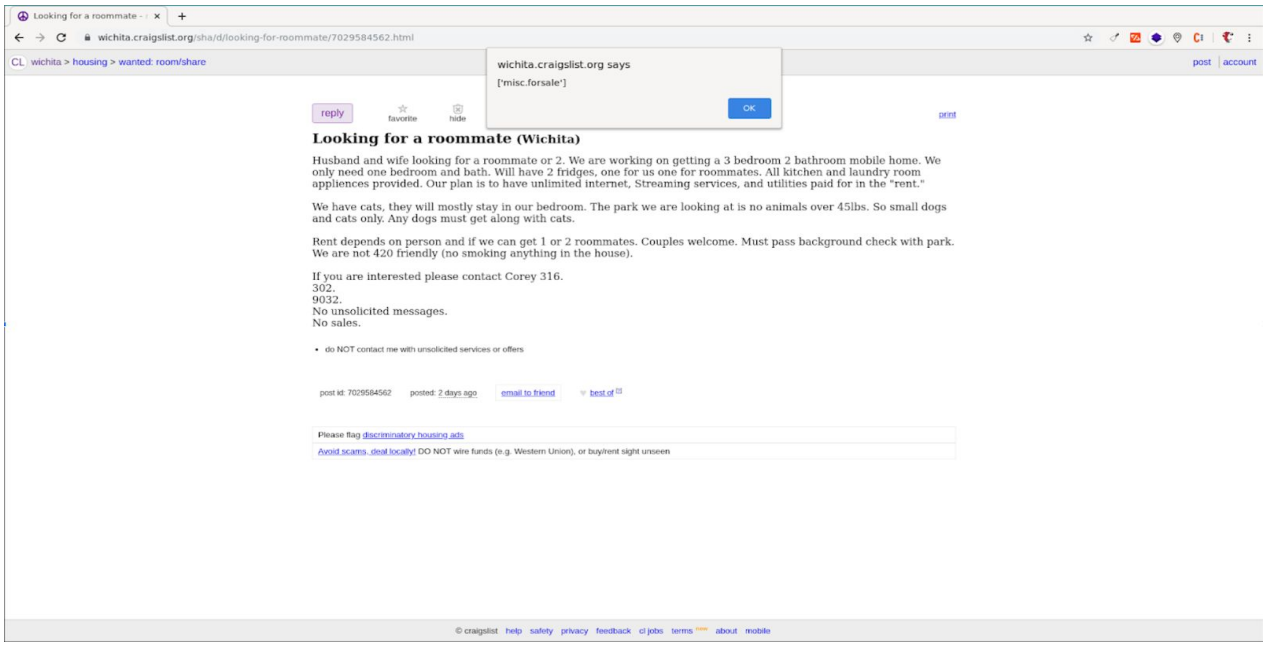


Fig. 9 The extension fetches the text from the page, and sends it as a GET query to the web server hosting the text classifier, and returns the label of the text.

There were a few weird things about Chrome's API that we thought we should mention. The extension can only show results on an alert object, and not through the extension's popup itself, since Chrome doesn't allow anything that's not a content script to have access to the browser's API. This exists as a security feature in Chrome. There are weird hacks to get around it, however, it is not recommended to use them, and we decided to not toy around with it. The extension is not available in the Chrome Web Store, and you would need to sideload it yourself. You also need to make sure the model has been trained and is running, while you use the extension, because, as we mentioned the extension is only responsible for sending requests to the Flask server. If it doesn't find a webserver to talk to, it will not be able to show any result.

Conclusion

Reflection

Overall, we consider this project to be a success. Summarized process for the project is as follows:

1) Our problem was defined and a solution was mapped out.
2) Data was downloaded and split into testing and training.
3) Our data was preprocessed.

4) A benchmark declared through confusion matrices and classification reports.
5) Each classifier was ran through the benchmark.
6) A support function was created to predict text and return a text label.
7) Lastly, a chrome extension utilizing the support function was implemented.

Generally, the project was straight-forward with very few speed bumps in the road. The most difficult part was benchmarking each classifier and finding the optimized hyperparameters, other than those things, everything was pretty easygoing.

We're glad that we did this project and we believe that the same idea could be used to implement other natural language processing solutions. The steps for preprocessing would remain the same but the data being fed into the classifiers would be different. This project serves as a good baseline for any kind of categorization problem.

Improvement

Although we're happy with the outcome of this project, there are a couple of things we believe could have been improved in either our data selection or our classifier selection. One major problem we had with the data is that it contains a lot of noise words. As most of the features come in the form of email or forum like text, there are things such as random email addresses, phone numbers, signatures, and typos that are in the data. These things likely have a minimal impact on the classifiers, but it would be nice to remove any typos, email address, phone numbers, signatures, and gibberish in general from the dataset to see if there is any improvement in the classifications. However, there are headers within the dataset that we believe could have a much larger impact on the classifiers than the other things. As detailed on the 20 News Group documentation,[10] the headers in the text can lead to overfitting, we did not remove the headers but had we taken a deeper look at the data, that may have been a solution we had come up with.

More tuning could have been done on the SGDClassifier, however, there are so many parameters that we could tune that it would take forever to actually do so. There could be some hyperparameter combination for it that would make it perform better over LinearSVC, but in general we are pretty happy with how the classifier turned out anyways.

Lastly, the Grid Search scoring was done on accuracy. It probably would have been better to Grid Search based on a different metric rather than accuracy, however, in our case, we believe the results of the Grid Search would have been about the same with maybe some slightly improved model selection.

[10] scikit-learn.org/0.19/datasets/twenty_newsgroups