Old Bailey Decisions

Michael Young

Project Overview

For this project, we were asked to predict court decisions (guilty or not guilty) based on the transcribed dialogue of court cases from the *Old Bailey* during the period from 1674-1913. The transcribed dialogue was provided to us in the form of 3 preprocessed feature sets:

1. **Bag of words**: A 10,000 element vector in which each dimension corresponds to a word and the value stored corresponds to how many times that word appeared in the court dialogue. The 10,000 words represented were the 10,000 most frequently used across all the cases.
2. **TF-IDF**: A remix of bag-of-words where instead of the counts being stored for each word, a value that seeks to capture how relevant a word is to a document is stored. Once again, the top 10,000 words were considered.
3. **GLOVE**: The average of each word's "embedding" in the document, weighted by the word's tf-idf score. Each example was a 300 dimensional vector.

Additionally, we were provided with a metadata set consisting of 6 facts about the trial (such as age and gender of the defendant). This data was not preprocessed.

The data was provided to us in a training set consisting of 17,500 examples, a test set consisting of 2,250 examples, and an evaluation set consisting of 5,250 examples. We were tasked to make 6 submissions of predictions on the evaluation set to Kaggle. Across these 6 submissions, we were required to use at least 4 different learning algorithms with at least 2 distinct feature sets. Additionally, 5 of the 6 submissions needed to be made without the use of any machine learning libraries.

The main ideas explored in my approach to this task were data wrangling/ feature engineering, and machine learning (ML).

Data Wrangling

Data wrangling is the unsexy secret that lies behind the glossy facade of any successful machine learning pipeline. We've developed many impressive algorithms that can perform unbelievably well across a range of tasks, but no matter the algorithm's strength, a lackluster input to a model will usually result in a lackluster output. Thus, developing a strong initial feature set to feed into an algorithm is one of the best things to do to improve a model's performance.

My approach to the data for this task was first to test each of the provided feature sets with my respective ML algorithms, and use the one that performed best. I found that across each algorithm, the TF-IDF feature set produced the best results.

I next wanted to explore the metadata, as it contained many features which on the surface would seem relevant to the court decision. I processed the set by converting each feature to a one hot encoding. For the numeric data (age + number of victims), I binned using ranges that I thought would lead to compelling categories. Later, in an effort to improve my results, I binned these variables using quantile binning. I also combined many of the less common subcategory offenses into one category and even created a few new features. I'll refer to the first metadata set processed as *meta1* and the second (with the adaptive binning strategy) as *meta2.* I found that this metadata performed even better in most algorithms than the court dialogue data alone.

Following processing the metadata, I decided it would be interesting to explore what results I could get by combining the metadata with the other feature sets. I simply tacked on the metadata to the end of the dialogue data, and found that the TF-IDF + metadata delivered the best performance. Specifically the TF-IDF + *meta2* yielded some of the highest evaluation accuracies.

My last attempt at data wrangling was to let a neural network do the work for me. Using pytorch, I created a two layer neural net with 1000 neurons in the first layer and 100 in the second. I used batch normalization, dropout for regularization, and ReLU as my activation function for both of the layers. I used cross entropy as my loss function and trained the model using my TF-IDF + *meta1* data (at the point of training the network, I had not yet created *meta2*). I extracted the output of the second layer (a 100-dim vector) for each example and used this as a newly transformed feature set.

To summarize, here are the datasets I used across my 6 chosen kaggle submissions (as well as my additional submissions):

- TF-IDF + *meta1* (10084 dimensions)
- TF-IDF + *meta2* (10051 dimensions)
- TF-IDF (10000 dimensions)
- *meta1* (84 dimensions)
- Neural Net Feature Vector (100 dimensions)

Machine Learning algorithms

The next major component to this project was which ML algorithms to use for my submissions. Over the course of the semester, we discussed several learning algorithms spanning everything from decision trees to Adaboost. Of these, I used the following in my submissions:

1. **Decision Tree**: This learning algorithm involves constructing a tree based on the training set. The tree splits based on the features in the data that yield the highest information gain.
2. **Perceptron:** The seminal linear classifier. An online, mistake-driven algorithm, the perceptron learns a line that separates the data by updating the linear weights as mistakes happen. Several variants exist, and in my implementation I use the averaging, margin and weight decay variants combined.
3. **SVM:** or a "Support vector machine", this is another linear classifier, very similar in spirit to the perceptron, except that it is framed as a different problem:

maximizing the margin between the data classes. This framing introducing regularization and gradient descent into the learning process.

4. **Logistic Regression:** An algorithm similar to the above two linear classifiers with the following twist: instead of outputting a discrete label (-1 or 1) it predicts the probability of a certain input belonging to a class or not. Even though this is a much different approach to say SVM, the end implementation ends up looking very similar - it's minimizing some loss with gradient descent.
5. **Ensembling:** A classifier can be made by using other classifiers as building blocks. Boosting and Bagging are effective uses of ensembles. For my ensemble implementation, I simply combined perceptron, SVM and the decision tree and used the majority vote as my prediction.

Additionally, I wanted to try out gradient boosting. Gradient boosting is related to the Adaboost algorithm which we discussed in class. The key insight to gradient boosting is that it recognizes that the additive modeling used in Adaboost can be represented as a cost function and thus optimized using gradient descent. This is the one algorithm I used that I didn't implement myself.

For all algorithms, I used 5 fold cross validation on the training set to determine the optimal hyper-parameters.

Results

Submission 1- Decision tree w/ *meta1* (dec_tree_v1_evals.csv on kaggle)

For this submission, I used the decision tree algorithm that I implemented for homework 1 on the miscellaneous attributes feature set that I processed (*meta1*). Interestingly, the root attribute for this feature was the NaN category for defendant age data. This underscores the wisdom of sometimes not filling in missing values for data, as the very fact that it's missing may be useful information. I used a depth of 5 for my final tree.

Performance: Train accuracy: 78.8%; Test accuracy: 79.7%; Evaluation accuracy (public): 80.3%

Submission 2 - Perceptron w/ TF-IDF + *meta2* (perceptron_v2_misc2_evals.csv on kaggle)

For this submission, I used the perceptron algorithm that I implemented for homework 2 with the TF-IDF + *meta2* data. (I also submitted with the TF-IDF + *meta1* data, but the *meta2* data yielded a slightly higher evaluation accuracy). I used a learning rate of 1, a margin of 5, and no weight decay.

Performance: Train accuracy: 89.9%; Test accuracy: 84.8%; Evaluation accuracy (public): 84.2%

Submission 3 - Perceptron w/ NN feature vector (perceptron_v2_feat_vec_evals.csv on kaggle)

For this submission I used the perceptron algorithm with my neural-net transformed features. I used a learning rate of 0.001, no margin, and weight decay.
Performance: Train accuracy: ~90%; Test accuracy: ~84%; Evaluation accuracy (public): 83.5%

Submission 4 - SVM w/ TF-IDF + *meta1* (SVM_evals.csv on kaggle)

For this submission I used the SVM algorithm I implemented for homework 6. I used a learning rate of 1e-5 and a regularization strength of 1e5.
Performance: Train accuracy: 90%; Test accuracy: ~83.4%; Evaluation accuracy (public): 82.6%

Submission 5 - Ensemble (Perceptron + Logistic Regression + SVM) w/ TF-IDF + *meta1* (ensemble_evals.csv on kaggle)

For this submission, I ensembled Perceptron, SVM, and Logistic regression models trained on the TF-IDF + *meta1* data. This was a simple ensemble where for each example I chose the majority vote across the 3 models. I implemented all 3 of these models using the same implementations I used in the corresponding homeworks. For Perceptron and SVM, I used the same hyperparameters discussed above in their respective submissions. For logistic regression I used a learning rate of 0.1, and a regularization strength of 1e4.
Performance: Train accuracy: ~90%; Test accuracy: 84.8%; Evaluation accuracy (public): 84.0%

Submission 6 (Used external library) - Gradient Boosting w/ TF-IDF + *meta2* (xgb_tf.csv on kaggle):

For my one submission using an external library, I tried out gradient boosted decision trees using the XGBoost library. These algorithm paired with my TF-IDF + *meta2* data yielded the best evaluation results of the bunch.
Performance: Train accuracy:84.5%; Test accuracy: 83.9%; Evaluation accuracy (public): 84.8%

Additional submissions:

1. Perceptron w/ TF-IDF+ meta1 - (perceptron_v2_evals.csv on kaggle)
2. Perceptron w/ TF-IDF - (perceptron_v1_evals.csv on kaggle)
3. 5 neural net ensemble w/ TF-IDF + meta 1 - (nn_ensemble_evals.csv on kaggle)
4. Best performing neural net w/ TF-IDF + meta1 - (nn_evals.csv on kaggle)
5. Gradient boosting w/ NN feature vecs - (xgb_vecs.csv on kaggle)
6. Random Forest w/ NN feature vecs - (rf_evals.csv on kaggle)

	Train Accuracy (%)	Test Accuracy (%)	Eval Accuracy (%)
Decision tree w/ meta1	78.8	79.7	80.3
Perceptron w/ TF-IDF+meta2	89.9	84.8	84.2

	Train Accuracy (%)	Test Accuracy (%)	Eval Accuracy (%)
Perceptron w/ NN features	~90	~84	83.5
SVM w/ TF-IDF+meta1	90	83.4	82.6
Ensemble (Perceptron, SVM, Logistic Regression) w/ TF-IDF + meta1	~90	84.8	84.0
Gradient Boosting w/ TD-IDF + meta2	84.5	83.9	84.8

Discussion

There were several surprises that occurred over the course of this project. As the results show, perceptron performed extremely well compared with every other algorithm. The only algorithm that achieved a higher evaluation accuracy was the gradient boosting, and by less than 1%. I was definitely not expecting this from an algorithm famously unable to handle the XOR function. This is yet another example of the strange gap in ML between theory and practice.

Related to this surprise was the failure of my neural net to produce features that improved much on the unaltered TF-IDF and fairly naïvely processed metadata. I chalk this up to my ineptitude at training neural networks. I wasn't as scrupulous as I could have been in making sure that I wasn't overfitting. Additionally, I think it may have been wise to use the output of the first hidden layer instead of the second. My hunch is that using later layers for feature extraction essentially limits other algorithms in how many different ways they can deal with the features. By taking features from earlier layers we may allow the feature space to stay more expressive.

Another result that felt fairly counterintuitive was the uninspiring performance of both my "voting" ensemble attempts (both the ensemble of algorithms I implemented and the ensemble of my 5 best neural networks). The voting based ensembles failed to rise above the performance of the best performing member of the ensemble. A potential reason could be that there weren't enough members of the ensembles to make a real difference. Another could be that the members of the ensembles weren't diverse enough. They were all trained on the same features and were all linear classifiers, thus, if they were making mistakes, they were likely to be the same mistakes. Ensembles with different types of models and models trained on different feature spaces may yield more interesting results. "Bagging" seems like a way to address this problem as well.

Overall, this project taught me a few important lessons when it comes to approaching an ML task. The first is that it is essential to guard against the scourge of overfitting! For most of my algorithms I was very vigilant in performing cross validation and holding out my test set, but for my neural network, I got lazy and used my test set as the validation set. I suspect that this may have poisoned my neural networks and the subsequent features I gleaned from it from being too fitted to the test set.

Another, and perhaps the biggest take away from this project is the absolute imperative to perform thoughtful feature engineering. From my exploits in deep learning and computer vision I was almost hubristic in my belief that without doing much to the features, a neural net would easily outperform all other algorithms in the task. In this I was sorely mistaken. If in real estate the saying is “location, location, location”, then in ML it should be “feature engineering, feature engineering, feature engineering.” The bare minimum of binning and one-hot-encoding my miscellaneous features improved my model accuracy by several percent. I suspect that I could have bumped this up even further with some strategic feature pruning. There seemed to be this accuracy wall of around 83-85% that I could not break no matter how sophisticated my algorithm was, and I suspect that the only way to get beyond this wall would be to create better input features.

Future Directions (What I would do if I had more time)

As discussed above, I think the greatest area for improvement for my pipeline would be to thoughtfully re-engineer my features. My first thought on how to do this would be to examine which features correlate with the labels, and then perform some pruning based on this. I could also remove features that don’t appear much in my dataset. An additional way would be to use my decision tree to identify which features hold the highest information gain, and remove features that don’t pass a certain threshold. In one of my submissions, I used a decision tree of depth 5 - maybe I could simply use those 5 features as input to other models. I would also like to take another crack at using a neural network to identify relevant features for me.

Another direction I could go would be to use some recent deep learning approaches to structured and tabular data such as “TabNet.”

Code

In accordance with the following post by a TA on the discussion board:

For Jupyter Notebooks, we would like the following:

1. *the ipynb file*
2. *a pdf/html export of the file with the outputs as they see it (for confirmation)*
3. *The report as usual*
4. *You can't use libraries that are not on CADE , and you can't use any ML library (like sklearn).*

I’ve submitted my code as a series of Jupyter notebooks w/ pdfs showing their outputs.