Project Report

February 9, 2016 by Leiya Ma & Luciana Hartmann Paolillo Cendon

1. Overview

The objective of this CS155 Kaggle project is to predict the sentiment, either opposition or support, expressed by a speech. We are given data represented as a bag of words in each speech. The words are the 1000 most common words in the speeches, stemmed, and with stop words filtered out. The training dataset consists of 4189 speeches with their target labels while the 1355 speeches in testing dataset were unlabeled. Our objective is to train the model using the preprocessed data which consists of 1000 features and using the learned model to predict labels for the testing data.

Within our group Luciana and Leiya first implemented decision tree algorithm together, then Luciana focused on Naïve Bayes and SVM algorithms. Leiya implemented bagged and Adaboost Decision Tree, random forest algorithms, parameters adjustment (cross validation) and ensemble selection. The model library we used is Scikit-Learn. And the language we used is python.

2. Data Manipulation

We have tried several methods to preprocess the data, which include TF-IDF normalization, feature selection.

- **TF-IDF**: Term frequency-inverse document frequency, is a feature representation of a collection of documents, or a corpus. By using a bag of words as the feature space, we can project a document in the corpus onto a numerical vector. The main thought is to find the feature words that appears very frequently in a specific document while not so common it other documents in the corpus. Thus we think this feature word represent this specific document well. An example is a simple word count normalized by the total number of words, which suffers from inherent bias towards lesser meaningful words such as 'the' and 'is'. TF-IDF attempts to overcome this by weighting the term frequency within a document by the overall term frequency across the entire corpus.

  For this project, we used both word count and TF-IDF representations of the data, which the later one we deemed to be a more sensible representation for relative importance of keywords.

- **Feature Selection**: The training data set contains more than 4000 points, where each point is represented by a point in a 1000-dimensional feature space. But such a big number of features may result in learning model overfitting. Thus we tried reducing the number of features using $L1$-based feature selection introduced in class.

  $L1$-based feature selection uses the fact that linear models regularized by the $L1$-norm results in sparse solutions, which will remove less important features. The amount of reduction of feature space is control by the regularization parameter C in the objective function. For certain learners, we applied $L1$-based feature reduction using various values of C.

3. Learning Algorithm

As most of the learning algorithms used in this project were discussed during class lectures, this section mostly aims to describe our approach to exploring the parameter space grid of each learner.

- Logistic regression (using SGD)
- Support Vector Machines - SVCs
- Naïve Bayes
- Random Forests
- Bagging
- Gradient-boosted trees
- Adaptive boosting

- **Logistic Regression**: Logistic regression is a linear classification model that seeks to maximize the likelihood, or equivalently minimize the log loss. Using a regularization parameter $\lambda = 4 \times 10^3$, we made around 20% sparseness which is not satisfying.

- **Support Vector Machine**: Support vector machine is a max-margin classifier, where the model tries to linearly separate the data while maximizing the distance between the separating hyperplanes and the closest data points, which is the margin. In most cases, the data is not linearly separable due to noise, thus classifiers called soft-margin SVM's allow violation of the margin. One formulation of soft-margin SVM is:

$$\mathrm{argmin}_{\mathrm{w}} \frac{1}{2}\|w\|^2 + C\sum_i \xi_i$$

$$\text{subject to } \mathrm{y_i}(w^T\phi(x_i)+b) \geq 1-\xi_i \textit{ and } \xi_i \geq 0\ \forall i,$$

where ξ_i is the parameter representing the amount of violation allowed for i^{th} data point and $\phi^T\phi$ is the kernel function which transforms the data into a higher dimensional representation.

We implemented the SVM using SVC model in Scikit-Learn library with no change in default parameters. Using the 5-fold cross-validation we got the score of 0.61243. This was not bad, and we continued to normalize our data using $L1$-based regularization but turned out that this would destroy the structure of data and diminish the accuracy. Our score for this attempt reduced to 0.59615.

Therefore, we decided to abandon linear model and change to non-linear ones like decision tree and random forest. Another point is that when using TF-IDF transformed data, it really worked well and boosted from 0.6143 to around 0.64.

- **Naïve Bayes:** Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. Given a class variable y and a dependent feature vector $\mathrm{x_1}$ through $\mathrm{x_n}$, the algorithm uses the following classification rule:

$$\hat{y} = argmax_y P(y)\prod_{i=1}^{n} P(x_i|y)$$

where Maximum A Posteriori (MAP) is used to estimate P(y) and $P(x_i|y)$.

We used Multinomial Naïve Bayes from Scikit-Learn library since it’s the most suitable for classification with discrete features such as word counts from speech. However, the resulting scores from these attempts on public leaderboard were quite low around 0.57 – 0.59.

- *Providing weight to the data*: when analyzing the data labels, some words seemed to be more relevant than others in determining sentiment. For instance, words like “death”, “fail” or “good” would seem to be *more likely* to provide information about sentiment than others like “third”, “approach” or “three”. Therefore, we attempted to create a “weight vector” of dimension 1,000 and assign it to each word by multiplying the training data by this weight vector. We found that to significantly improve cross-validation step score to around 0.66 – 0.67 even for simple models, but the score in the public leaderboard remained low (around 0.59)

- **Boosted Decision Trees**: Adaptive Boosting refers to iteratively learning a data set using a collection of "weak" learners, where a weak learner is a learned model with low predictive power. This method takes advantage of the low variance of weak learners, while the inherently high bias of the weak learners is reduced by the ensemble nature of boosting.

- **Gradient Tree Boosting**: Gradient tree boosting is another method of boosting trees similar to AdaBoost. Whereas AdaBoost is used only for classification, gradient boosting can be used for both regression and classification. Although the concepts are similar, the differences in the algorithm might result is different generalization, thus we decided to include some models traine using gradient tree boosting.

  We generate different models by varying the number of rounds of boosting, the maximum depth of the base tree classifier, and the learning rate in a similar way as we did for the boosted decision trees. The accuracy could rise to higher than Adaboost Decision Tree algorithm (say about 0.6654) but would take almost 2 times as long. Therefore, it is not so efficient in operation.

- **Random Forest**: In contrast to the boosting methods, random forest seeks to take advantage of the low bias and high predictive power of deeper and more complex decision trees. As complex trees are prone to overfitting, random forest creates an ensemble of such trees where each tree is trained using different subsets of the data (bagging) and features. Bagging, or bootstrap aggregation, reduces the variance of the base classifiers and is not limited only to decision trees.

  We implemented the default random forest tree and searching the parameter space for better combinations (enabled by Python and Scikit library). Parameters includes:

  1) the total number of trees in the forest (n_estimater)
  2) the maximum number of features to consider when looking for the best split (max features)
  3) impurity function (Entropy or Geni)
  4) the maximum depth of the tree (max depth)
  5) the minimum number of leaf nodes

  For optimal n_estimater is at about 120 trees while we tested the set {100, 110, 120, 130} because we would like to take benefit of improved prediction accuracy of more trees.

  For max features we found that best prediction came when choosing 32 features. We used Gini as our impurity function and tested max depth in a set {50, 60, 70, 80, 90}.

It worked well in 70~80 period.

Another important parameter is the bootstrap, we had better prediction when setting the bootstrap to be true, which means that the samples of the dataset were no drawn with replacement. This makes sense because the tree-specific parameters like minimum number of leaf nodes varied highly and was so random. We just leaved the Grid-SearchCV /RandomGridSearch over parameters searching over a wide-enough range for these parameters.

4. *Model Selection*

Using Cross-validation and Ensemble selection we found that the SVM works well at first, but its accuracy drops down afterward and even lower when applied $L1$-based regularization. For Random Forest algorithm, the accuracy was quite consistent at around 66%.

5. *Conclusion*

This project gave us the opportunity to see how those learning algorithms really worked with the real-world data and how did the parameters tuning affected the learning results. Since we can utilize established machine learning libraries such as scikit-learn and etc., we can pay more attention on understanding their parameters rather than their implementations details and save much time.

There are several interesting things we found during the project implementation:

Normalization was a technique that did not yield all positive results. It may be useful in some linear model, but useless in most of our implementation. What could explain this phenomenon is that it may be just not suitable in this dataset because the features were just counting integers, which is quite trivial work though.

increasing a particular model's complexity. For example, a random forest's number of trees, can significantly lowers the training error at the initial time. However, after some optimal point, things could get worsen which might attribute to overfitting problems. At the same time, when we consider the time complexity problem, the improvement made by the more complex model was much smaller compared with the increasing complexity itself. Therefore, it would be unworthy to do this improvement. And it seems that the best alternate solution was to use a fairly complex algorithm with relatively low hyperparameters.

Cross-validation was a really useful in finding out best sets of hyperparameters. It was particularly convenient because it is applicable to all sorts of different algorithms, so we could utilize it with every new method we tried.