Ensemble Learning

– Quality Classification on Customers' Reviews

Abstract

The best performances on the test set are carried out by:

Bagging + DecisionTree--Gini, Max Depth:2, Minimum Split:100 (ROC AUC: 0.76) and AdaBoost.M1--Itr:2 + LinearSVM--C=0.7(ROC AUC: 0.80).

Although Bagging is marked as "almost useful in all circumstances", it still highly relies on the features and sample ratios when forming a training subset. When these two ratios are set to extremely low, underfitting can happen; When both are set high and the iteration amount is large, the learning results can be easily overfitted.

Boosting algorithms can easily lead to overfitting too. Take the SVM for example, the SVM already carries out a high accuracy on our data model, and as the iteration amount of boosting increases, most of the time, the ROC AUC scores start to follow a downward trend.

This is also an issue for most of the base estimators we select for ensembles. They carry out a good and stable-enough prediction before the ensemble. In that way, how Bagging and Boosting works with estimators can be clouded.

The experiment can hardly be qualified as complete in our case. To improve the structure and make the data more comprehensive, we could do:

1. Bagging and Boosting Parameter Study on validation set (now on test set)
2. More unstable classifiers to see how Bagging and Boosting work with them
3. More testing with different data preprocessing, for example: more/less features extracted from vectorization, other ways to map textual contents to numeric data, ablation study on features(especially on those sentiment analysis features)

1 Introduction

Online shopping plays an important role in our daily life. When selecting online products, customers' reviews are an aspect to pay attention to. Current quality of reviews are uneven. Short or meaningless reviews, and even sabotage reviews can happen, which seriously affects customers' shopping experience and their choices of products.

Our goal is to use ensemble learning to predict the quality of reviews in real-life scenarios. The data is provided by Amazon.

Both Bagging and Adaboost.M1 frameworks are used for ensembles. The prediction results are evaluated by AUC scores.

Ensemble learning helps improve machine learning results by combining several models. These two decrease the variance of a single estimate as they combine several estimates from different models. So the result may be a model with higher stability[1].

2 Programming Modules

Numpy and Pandas are used for data processing. Matplotlib is used for analyzing extracted features and training result visualization. We invoke NLTK and TextBlob for textual content processing and sentiment analysis.

In ablation study, we also experimented on the combination of Naive Bayes, Support Vector Machine and Decision Tree with Bagging and Adaboost.M1 frameworks.

The running time and calculation weight is heavy for Support Vector Machine, that's the reason sklearnex.patch_sklearn is called. This is an Intel extension for Scikit-learn so that we can accelerate Scikit-learn applications and still have full conformance with all Scikit-Learn APIs and algorithms[2].

3 Data Analyze and Pre-processing

The train set contains more than 57K sample points. Each sample point contains a textual feature that could be mapped to a high-dimensional vector. Under these circumstances, it is important to limit the amount of the features in the pre-processing stage. We could achieve this by extracting limited terms in the vectorization stage or do dimensionality reduction after vectorization.

3-1. Drop features an Invalid Samples

In the train set, features "votes_up" and "votes_all" are used to measure the quality. If "votes_up/votes_all" is smaller than 0.9, the review will be marked as "good quality". These two features are also missing from the test set.

First, we will use this rule to check if any invalid data exists. Then, we need to come up with a way to deal with these two features.

One common way is to deal with missing features as such would be to drop them. We could also use other features to interpolate missing values, but this could potentially introduce over-fitting.

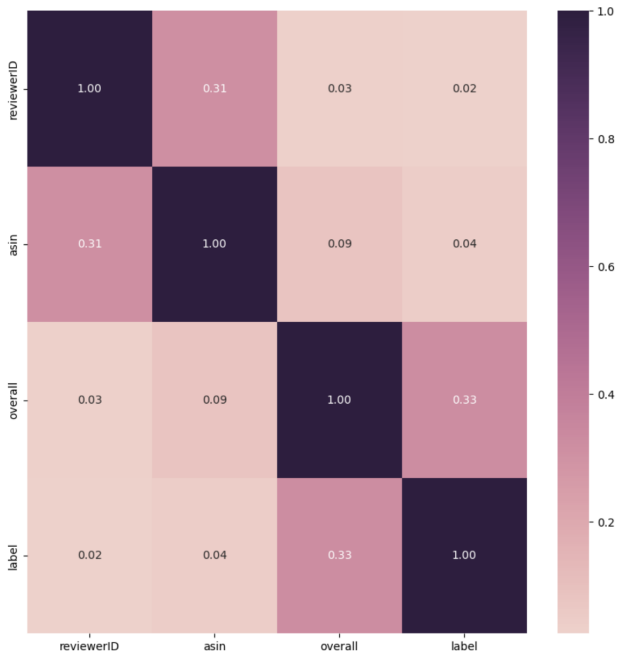
Besides, the labels, that is the goal of our task, are directly set by the return values from votes_up/votes_all. The learning on other features would be comparatively meaningless, as the relationship between labels and votes is clearly known.

	reviewerID	asin	reviewText	overall	votes_up	votes_all	label
0	7885	3901	First off, allow me to correct a common mistak...	5.0	6	7	0
1	52087	47978	I am really troubled by this Story and Enterta...	3.0	99	134	0
2	5701	3667	A near-perfect film version of a downright glo...	4.0	14	14	1
3	47191	40892	Keep your expectations low. Really really low...	1.0	4	7	0
4	40957	15367	"they dont make em like this no more..."well.....	5.0	3	6	0
...	...	...	...	...	...	...	...
57034	58315	29374	If you like beautifully shot, well acted films...	2.0	12	21	0
57035	23328	45548	This is a great set of films Wayne did Fox and...	5.0	15	18	0
57036	27203	42453	It's what's known as a comedy of manners. It's...	3.0	4	5	0
57037	33992	44891	Ellen can do no wrong as far a creating wonder...	5.0	4	5	0
57038	27478	19198	I agree with everyone else that this is a grea...	2.0	5	5	1

3-2. Numeric Features Analysis

There are two ID-like features in the train set. Theoretically, it's a good idea to try to account for all possible features in the training stage. However, the ID numbers can be random.

Before jumping into any conclusions, we will use correlation scores to see if there exists any hidden pattern or connection between the labels and IDs. By saying hidden patterns we are more talking about a reflection of real-life. For example, if a product(same ID) has good quality, then the reviewer might give good-quality feedback and others vote up for it more.



From the diagram, the features "reviewerID" and "asin" have extremely low correlation with the label we are going to predict. This suggests that we could remove these two sets of data in the training stage, as they have low impact on prediction.

Another reason we move ID-related features is: "One of the general rules of building usable machine learning models is that you never include features that won't be available at prediction time, account ID is one of such features"[3].

Technically, an ID can be encoded using something like feature hashing, but this would only mean that for an unknown ID you would be using another, random ID from the training set.

3-3. Language Processing: Sentiment

Based on real-life cases, when a product is of low quality, reviews can be subjective and emotional. Thus, in our task, we should not only focus on extracting terms from textual contents, but also to analyze its sentiment.

Sentiment Analysis can help us decipher the mood and emotions of the general public and gather insightful information regarding the context. Sentiment Analysis is a process of analyzing data and classifying it based on the need of the research[4].

Inspired by “Sentiment Analysis using TextBlob”[4], module “TextBlob” is used. It is a simple library which supports complex analysis and operations on textual data. We will extract “Polarity” and “Subjectivity” as features for learning.

Polarity lies between [-1,1], -1 defines a negative sentiment and 1 defines a positive sentiment. Subjectivity lies between [0,1]. Subjectivity quantifies the amount of personal opinion and factual information contained in the text. The higher subjectivity means that the text contains personal opinion rather than factual information.

One thing needs extra care is the polarity, since it can have negative values. Our experiment will use Naive Bayes as base models later, and negative values do not work with Naive Bayes, as a probability value cannot be under 0. Thus, after extracting polarity values, an extra normalization to map values between 0-1 range is required.

3-4. Textual Contents Clean Up

Before vectorization, we need to clean up the textual contents. This happens after sentiment analysis otherwise, sentiment conveyed by punctuations and so on can be affected.

This section contains: punctuation removal, lowering texts, tokenization, and stop word removal. Stopwords are the commonly used words and are removed from the text as they do not add any value to the analysis. These words carry less or no meaning[5]. Some of them are: i, me, myself, you etc. These words will always be in the extracted terms after vectorization if we do not remove them first.

3-5. Vectorization

TF-IDF is first chosen for text vectorization. Our data amount is huge, which could result in large values if we use CountVectorizer, since this algorithm only cares about how many times a term appears. Another advantage is that TF-IDF has embedded normalization concept in, and returns a sparse matrix that could fasten the SVM running.

If running TF-IDF on top of an unbalanced data set, with a restricted amount of features to extract, there is a high chance that the majority of words chosen will be from the majority class (negative words/features). This makes sense since TF-IDF is selecting features based on term frequency alone and negative words are present in most of the samples. As a result, the minority class gets under-represented[6].

The Images on the right visualize the top-frequency terms extracted by TF-IDF.

Two diagrams below show the top 100 terms extracted from text contents that are associated with label 0 or 1 only. Top two images are top 100 terms extracted from all text contexts in the train set. Top left is vectorized by non-weighted TF-IDF, and top right is with class-weighted TF-IDF.

We could see that terms representing classes can be quite different. Non-weighted TF-IDF could easily lead to low-interpretability.

From the top images, we could also notice that weighted and non-weighted have main differences on mid-high frequency terms. We will stick with class-weighted TF-IDF for now. Next step is to use the same vocabulary to vectorize the test set, and construct the final train and test sets by concatenating TF-IDF vectors with other non-textual features.

4 Ensemble Learning FrameWork

In this section, we will implement Bagging and AdaBoost.M1 from scratch. For frameworks, we want them to return both the predicted probability and the predicted result.

The main causes of error in learning are due to noise, bias and variance. Ensemble helps to minimize these factors. These methods are designed to improve the stability and the accuracy of Machine Learning algorithms. Combinations of multiple classifiers decrease variance, especially in the case of unstable classifiers, and may produce a more reliable classification than a single classifier[7].

The final prediction is made by the voting results from all the estimators' results in an ensemble framework. We can vote by mean, majority or weights. The weights usually are formed from the performance of each estimator. A main difference between Bagging and Boosting is that, in Bagging, samples have uniform weights but Boosting samples will be changed based on the estimators' prediction results. Samples that have been misclassified will have increasing weights, and correct-classified samples' weights will be reduced.

One of the advantages of AdaBoost.M1, as suggested by some AI forums, is that it is best used to boost the performance of decision trees on binary classification problems. It is also proved to be sufficiently functional with weak classifiers, which basically means the hypothesis can be quite simple and light weighted[8].

Another main difference between these learning methods is the way in which they are trained. In bagging, weak learners can be trained in parallel, but in boosting, they learn sequentially[9], since the sample weights have to be updated by the learning result from the last-run estimator.

4-1. Bagging

Bagging, also known as bootstrap aggregation, is the ensemble learning method that is commonly used to reduce variance within a noisy dataset. In bagging, a random sample of data in a training set is selected with replacement—meaning that the individual data points can be chosen more than once.

Below are the codes to achieve this subset generation. Also, inspired by Sklearn modules, we will not only have subsets for samples, but also on features.

```
def get_subset_index(rand_range, ratio=0.1):
    """
    Take random samples from the main set with replacement
    oob: whether to retun oob indices
    """
    amount = round(rand_range*ratio)
    indices = random.sample(range(rand_range), amount)
    return indices
```

To add up probability from estimators for final probability and prediction, we provide two methods for voting. One is taking the average, and the other is by weighted sum average.

If we turn on the “weight” option for our Bagging model, we will store all samples that haven’t been taken to form the subset as out-of-bag. After each estimator is trained, we will use the associated out-of-bag samples to predict. The accuracy scores for each estimator are stored in a list and normalized to [0,1] range for ensemble.

```
    # train each estimator
    trained_models = []
    for i, model in enumerate(self.estimators):
        # fit the estimator on asscociated x and y subset
        trained_model = model.fit(x_subsets[i], y_subsets[i])
        trained_models.append(trained_model)

        #print("fit estimator {}".format(i))

        # if weight set to "acc", we need to run trained model on oob
        # the accuracy score will be the weight for the estimator
        if self.ensemble_weight == "acc":
            oob_y_hat = trained_model.predict(x_oob_sets[i])
            # get the accuracy score
            estimate_weight.append(accuracy_score(y_oob_sets[i], oob_y_hat))
    #print(estimate_weight)

    # now update the Bagging regressor
    self.estimators = trained_models
    self.sub_features = feature_subset_index

    if not estimate_weight:
        self.weights = []
    # normalize weight
    else:
        estimate_weight = np.array(estimate_weight)
        normalized_weight = estimate_weight*(1/sum(estimate_weight))
        self.weights = list(normalized_weight)
```

4-2. Adaboost.M1

Implementation for this framework basically follows the “AdaBoost from Scratch”[8].

Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Source: The Elements of Statistical Learning, ch. 10

A test condition for the error is added. If error for an estimator is above 0.5, the base estimator would be weak enough and the Adaboost training should be aborted. Under this case, users should choose another base model to re-run the initialization of Adaboost for training and predicting.

5 Parameter Tuning and Ablation Study On Base Estimators

Before jumping into ensemble learning, we will check how base estimators perform on the data model. Then compare and contrast the performance with ensembling.

We will use Gaussian Naive Bayes, Complement Naive Bayes, Multinomial Naive Bayes, Decision Tree and SVM for testing. Cross-validation are run on the train set. To measure the training quality, mean accuracy scores are used.

5-1. Naive Bayes

The mean accuracy scores for different Naive Bayes models with cross-validation are:

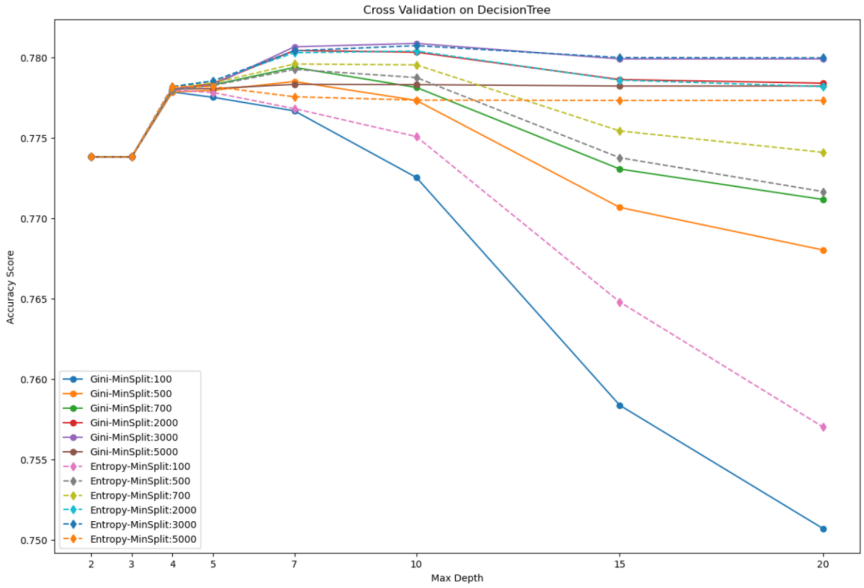
Multinomial-accuracy: 0.7762232959978439

Complement-accuracy: 0.6693138446521175

Gaussian-accuracy: 0.6779570358669801

We will choose Multinomial Naive Bayes and Complement Naive Bayes for ensemble. They are the strongest and weakest classifiers among the Naive Bayes models we tested.

5-2. Decision Tree



Tree depths, minimum split amounts and the algorithms for choosing nodes are all tested.

Most of the accuracy curves follow an upward trend then drop down as the max depth grows.

The first anchor we will choose is:

Gini-max depth:2 and minimum split sample amount:100. The accuracy score is 0.7738 for all DecisionTree models with max depth at two. So we will randomly choose one as the base estimator in the ensemble.

Best performance among them would be: Gini-depth 10 and minimum split: 3000. The accuracy score for this occasion is 0.7809.

Note when Gini-depth 20 and minimum split: 100, it has the lowest accuracy. This is potentially due to overfitting. We want to check how the ensemble behaves when the base classifier is already overfitting. An assumption is that the accuracy will drop further with Adaboost, but improves with Bagging, as the bootstrap can reduce overfitting to some extent.

5-3. Support Vector Machine

We will test linear SVM only, due to the calculation time and the limited scope of the experiment. The value of the regularization parameter needs to be tuned in study. The larger the regularization parameter is, the more punishment is made on misclassification. This usually means we will have high accuracy on the train set, but weak generalization ability[9].

Here listed the regularization parameter values we test: 0.1, 0.5, 0.7, 1.0. And the associated mean accuracy scores are [0.7762583775543941, 0.7764161631025265, 0.7765213555171642, 0.7765564186307686].

We will choose C=0.7(regularization parameter) for later testing. There are no huge improvements when we increase the regularization parameter from 0.7 to 1.0. This suggests that when C=0.7, the hyperplane already has close to the lowest possible misclassification error.

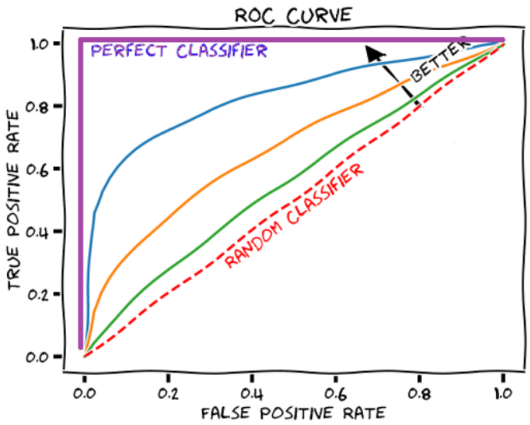
6 Parameter Tuning and Ablation Study On Ensemble Frameworks

In this section, we will focus on studying how ensemble frames' parameters can affect the learning quality. The estimators will be the models we choose from Section 5.

Accuracy scores and ROC-AUC scores are used to measure the learning quality.

Sensitivity and specificity are two common-used matrices for accessing how well a model fits a dataset.

- Sensitivity: The probability that the model predicts a positive outcome for an observation when indeed the outcome is positive. This is also called the “true positive rate.”
- Specificity: The probability that the model predicts a negative outcome for an observation when indeed the outcome is negative. This is also called the “true negative rate.”



One way to visualize these two metrics is by creating a ROC curve, which stands for “receiver operating characteristic” curve. In general, an AUC of 0.5 suggests no discrimination (i.e., ability to diagnose patients with and without the disease or condition based on the test), 0.7 to 0.8 is considered acceptable, 0.8 to 0.9 is considered excellent, and more than 0.9 is considered outstanding[11].

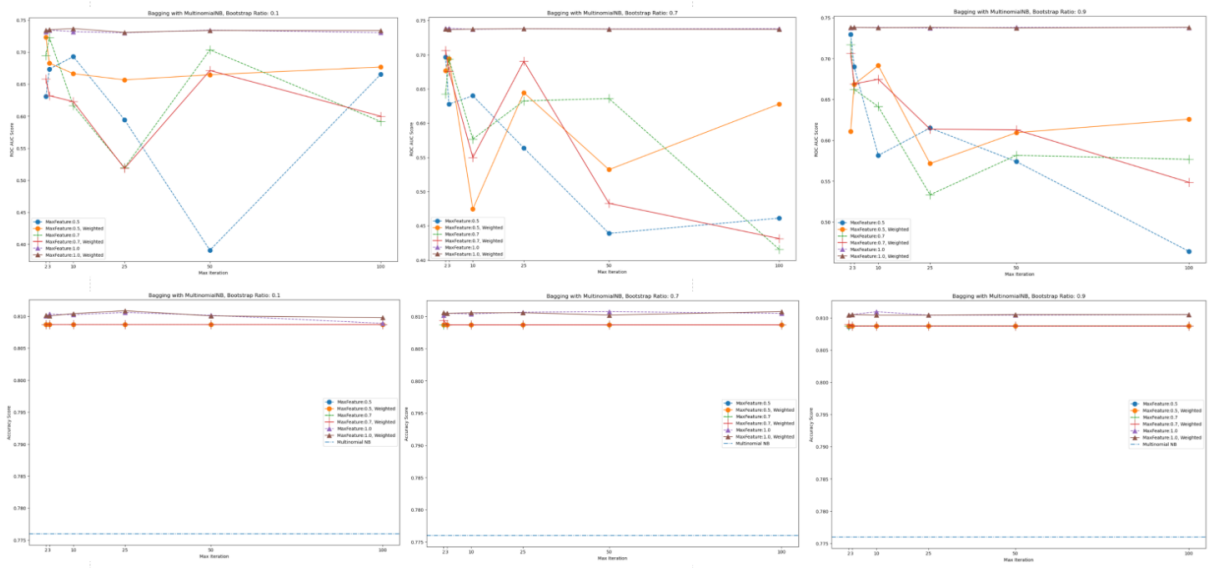
6-1. Estimators for Ensemble

```
multinomial_nb =  MultinomialNB()
complement_nb = ComplementNB()

dicisiontree_1 = DecisionTreeClassifier(criterion="gini", max_depth=2, min_samples_split=100)
dicisiontree_2 = DecisionTreeClassifier(criterion="gini", max_depth=10, min_samples_split=3000)

svm = SVC(kernel="linear", C=0.7, probability=True)
```

6-2. Ensemble (No SVM)

One way to visualize these two metrics is by creating a ROC curve, which stands for “receiver operating characteristic” curve. In general, an AUC of 0.5 suggests no discrimination (i.e., ability to diagnose patients with and without the disease or condition based on the test), 0.7 to 0.8 is considered acceptable, 0.8 to 0.9 is considered excellent, and more than 0.9 is considered outstanding[11].

MultinomialNB



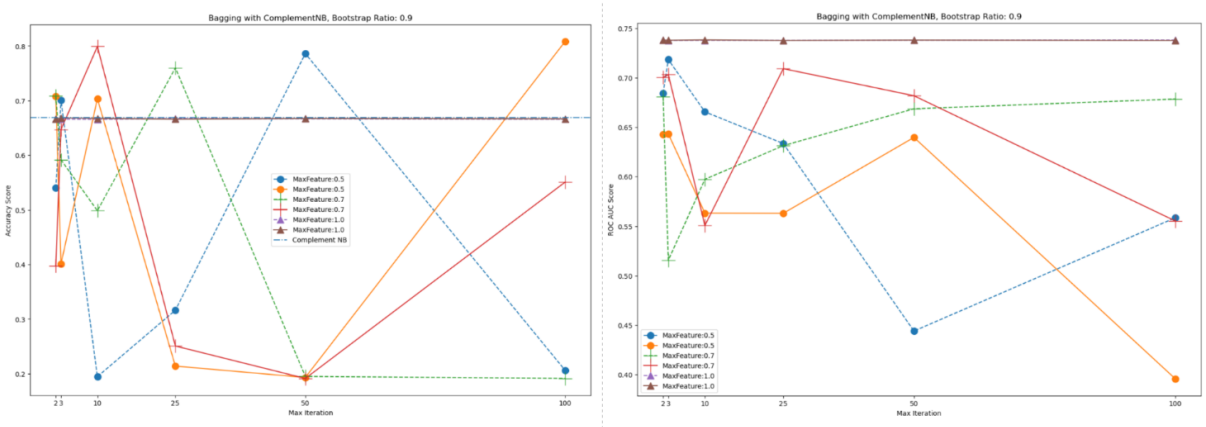
Accuracy for Multinomial Bayes sufficiently improved in the Bagging framework. But the Roc AUC scores are not as good as without ensemble learning, and fluctuates a lot. Generally the best performances are when the bootstrap ratio is set to 0.9 and take all features into each bootstrap subsets.

Taken together with accuracy scores and Roc AUC scores, we can see that when we take a limited amount of features to train subset models, the accuracy scores are almost unchanged.

Ensemble with MultinomialNB performs the worst with sample ratio is set to 0.1 and feature ratio=0.5. This is because we take limited information for training, leaving the model underfitting.

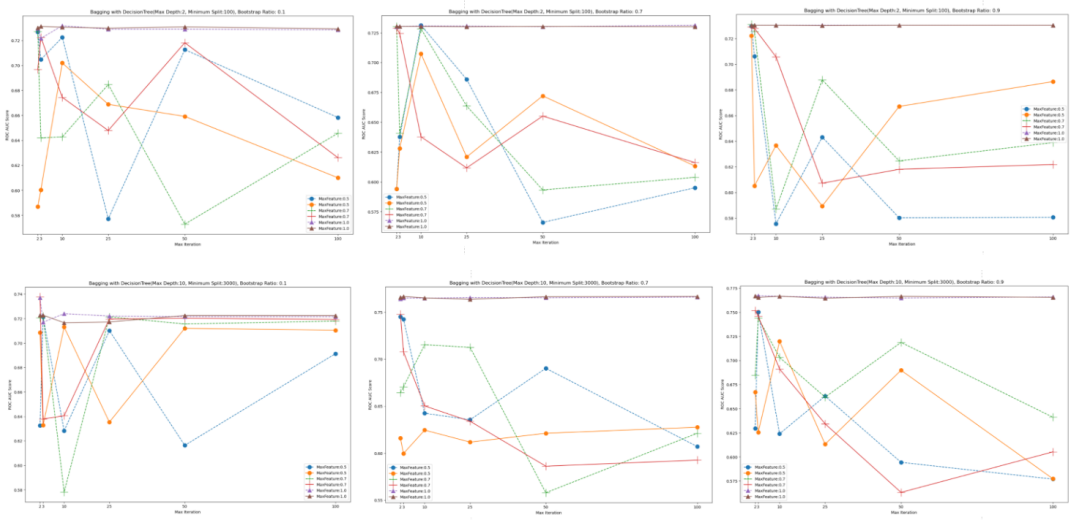
As the iteration amount grows, an ensemble with weights gets better than non-weighted. The weight we use is based on how accurate the estimator can be. Thus, estimators with better accuracy will weigh more in predicting probability calculation. When the iteration amount, also known as the estimated amount, is limited, the weights would not have such a big impact.

ComplementNB



We only take sample_ratio=0.9 as an example, bagging behaviors with Complement Naive Bayes are like this, unstable, and worse than without using Complement Naive Bayes only.

One thing needed to mention is that Complement Naive Bayes already has the worst performance (Section 3.1.1). Using it as the base model for Bagging framework is basically repeating bad learning and adding all the bad results up.

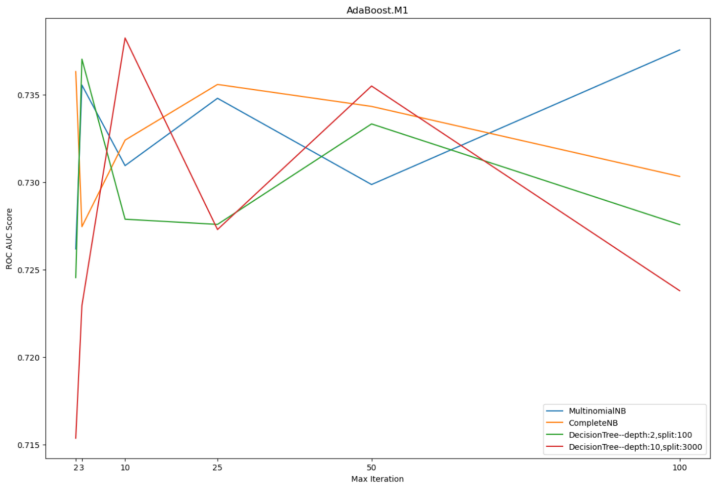


As the iteration(bootstrap bag amount) increases to 100, the ROC AUC scores are still low, due to potential overfitting.

The peak AUC score is 0.7666653321817656, with Decision Tree -- Max Depth:10, Minimum Split:3000 and Bagging with sample_ratio=0.7 and feature_ratio=1.0, with 3 iterations.

When sample ratio is set to 0.1 or 0.9, weighted ensembles have weaker performances then non-weighted ones. The reason is underfitting and overfitting.

Accuracy scores and Roc AUC scores on Adaboost.M1 are generally lower than Bagging, and higher than without ensemble.



Bagging is best for data that has high variance, low bias, and low noise, as it can reduce overfitting and increase the model's stability. Boosting is more suitable for data with low variance, high bias, and high noise, as it can reduce underfitting and increase accuracy[12]. We can assume by this that our data model has low noise and low bias. We will verify this again with the ensemble on SVM.

6-3. Ensemble (With SVM)

We will only try max iteration with 2, 3, 5, due to the limit on computility.

Right images are parts of the accuracy scores and ROC AUC scores from the Bagging + LinearSVM combination.

Without ensembling, the accuracy score for SVM is around 0.777. With ensembling, the accuracy scores are mostly more than 0.8. It is fair to say that Bagging has improved the accuracy at a limited rate.

However, regardless of the combination of Bagging parameters, the ROC AUC scores mainly fall into the 0.6-0.7 range.

Based on the analysis in Section 6 header, we could conclude that Bagging + LinearSVM does not fit on our data model that well.

As for Adaboost.M1, the accuracy scores and ROC AUC scores for iteration at 2, 3, 5 are as following:

```
Bagging ratio: 0.1 max_feature: 0.5 weight: uniform
accuracy: [0.8087080656673804, 0.8087080656673804, 0.8087080656673804]
roc auc: [0.668047481178617, 0.5439311743357352, 0.3868854748488362]
Bagging ratio: 0.1 max_feature: 0.5 weight: acc
accuracy: [0.8088865096359743, 0.8087080656673804, 0.8087080656673804]
roc auc: [0.6421303607184728, 0.5501616922283991, 0.5879692789911871]
Bagging ratio: 0.1 max_feature: 0.7 weight: uniform
accuracy: [0.8087080656673804, 0.8089757316202713, 0.8087080656673804]
roc auc: [0.6968831612842671, 0.5978105734017468, 0.61847946320362]
Bagging ratio: 0.1 max_feature: 0.7 weight: acc
accuracy: [0.7983583154889364, 0.8087080656673804, 0.8087080656673804]
roc auc: [0.6814161382243681, 0.5219675425827613, 0.46420525557890163]
Bagging ratio: 0.1 max_feature: 0.9 weight: uniform
accuracy: [0.7930942184154176, 0.8096002855103498, 0.8087080656673804]
roc auc: [0.6090184455316093, 0.6818686109391261, 0.5503889320223683]
Bagging ratio: 0.1 max_feature: 0.9 weight: acc
accuracy: [0.7505353319057816, 0.8086188436830836, 0.8077266238401142]
roc auc: [0.6044134949150979, 0.6901153674204002, 0.5601523700451845]
Bagging ratio: 0.7 max_feature: 0.5 weight: uniform
accuracy: [0.808083511777302, 0.808083511777302, 0.8081727337615988]
roc auc: [0.533461316953406, 0.6126128068560551, 0.6315862490284675]
Bagging ratio: 0.7 max_feature: 0.5 weight: acc
accuracy: [0.8087080656673804, 0.8087080656673804, 0.8087080656673804]
roc auc: [0.5755256361067567, 0.6523548649899225, 0.5647449192146066]
Bagging ratio: 0.7 max_feature: 0.7 weight: uniform
accuracy: [0.8087080656673804, 0.8087080656673804, 0.8087080656673804]
roc auc: [0.6519706774215858, 0.6711844812510704, 0.5459687166550302]
Bagging ratio: 0.7 max_feature: 0.7 weight: acc
accuracy: [0.8084403997144897, 0.8087080656673804, 0.8087080656673804]
roc auc: [0.670575678261385, 0.6609085701512298, 0.6289382055960271]
Bagging ratio: 0.7 max_feature: 0.9 weight: uniform
accuracy: [0.8096002855103498, 0.8087080656673804, 0.8087080656673804]
roc auc: [0.6712181349705576, 0.6387494998254536, 0.5832860603206387]
```

Accuracy: [0.8075481798715204, 0.5729835831548894, 0.5745895788722342]

ROC AUC: [0.7790969338271133, 0.7542650171747177, 0.7535119766074745]

6-4. Session Summary

Bagging:

- Linear Support Vector Machine has a comparatively low ROC AUC scores when combining with Bagging framework. This type of combination has the highest ROC AUC score at around 0.67 and the associated accuracy score at 0.81.
- With the combination of Decision Tree, the best performance has ROC AUC score at around 0.77 and the associated accuracy score also at around 0.81.

Adaboost.M1:

- Linear Support Vector Machine has a comparatively higher ROC AUC score when combined with Adaboost framework. This type of combination has the highest ROC AUC score at around 0.78 and the associated accuracy score at 0.81.
- With the combination of Decision Tree, the best performance has ROC AUC score at around 0.73 and the associated accuracy score also at around 0.80.

Due to the train set sample amount and feature high dimensions, Decision Tree can be unstable and Bagging helps to stabilize through a list of issues that could cause low performance: overfitting, underfitting, etc.

As for SVM, when there are a lot of features, it is easy for models to find spurious relationships between features and target. There are two generic solutions to this problem: dimensionality reduction and regularization. Regularization penalizes the model for adding complexity. And SVM works well with our data model as features calculated by SVM are automatically regularized. By ensembling this with Adaboost, the disadvantage of Adaboost being easily affected by noise seems to be reduced.

To finish up our experiment, we will use below combinations of models on the test set:

- Bagging + DecisionTree--Gini, Max Depth:2, Minimum Split:100 (ROC AUC: 0.76)
- AdaBoost.M1--Itr:2 + LinearSVM--C=0.7(ROC AUC: 0.80)

7 Conclusion

The best performances on the test set are carried out by:

Bagging + DecisionTree--Gini, Max Depth:2, Minimum Split:100 (ROC AUC: 0.76) and AdaBoost.M1--Itr:2 + LinearSVM--C=0.7(ROC AUC: 0.80).

The decision boundary of SVM (with or without kernel) is always linear (in the kernel space or not) while the decision boundary of the decision tree is piecewise linear (non-linear). This could be a reason why these two base models work differently with Bagging and with Boosting.

Although Bagging is marked as “almost useful in all circumstances”, it still highly relies on the features and sample ratios when forming a training subset. When these two ratios are set to extremely low, underfitting can happen; When both are set high and the iteration amount is large, the learning results can be easily overfitted.

Boosting algorithms can easily lead to overfitting too. Take the SVM for example, the SVM already carries out a high accuracy on our data model, and as the iteration amount of boosting increase, most of the time, the ROC AUC scores start to follow a downward trend.

The experiment can hardly be qualified as complete in our case. To improve the structure and make the data more comprehensive, we could do:

4. Bagging and Boosting Parameter Study on validation set (now on test set)
5. More unstable classifiers to see how Bagging and Boosting work with them
6. More testing with different data preprocessing, for example: more/less features extracted from vectorization, other ways to map textual contents to numeric data, ablation study on features(especially on those sentiment analysis features)

8. References

[1] Bagging vs Boosting in Machine Learning
https://www.geeksforgeeks.org/bagging-vs-boosting-in-machine-learning/

[2] Intel® Extension for Scikit-learn https://intel.github.io/scikit-learn-intelex/latest/

[3] Encoding ID variables for machine learning
https://stats.stackexchange.com/questions/535931/encoding-id-variables-for-machine-learning

[4]Sentiment Analysis using TextBlob
https://towardsdatascience.com/my-absolute-go-to-for-sentiment-analysis-textblob-3ac3a11d524

[5] Text Preprocessing in NLP with Python Codes
https://www.analyticsvidhya.com/blog/2021/06/text-preprocessing-in-nlp-with-python-codes/

[6] How to correctly use TF-IDF with imbalanced data
https://www.deepwizai.com/projects/how-to-correctly-use-tf-idf-with-imbalanced-data

[7] what-is-the-difference-between-bagging-and-boosting
https://quantdare.com/what-is-the-difference-between-bagging-and-boosting/

[8] AdaBoost from Scratch https://towardsdatascience.com/adaboost-from-scratch-37a936da3d50

[9] What is the influence of C in SVMs with linear kernel?
https://stats.stackexchange.com/questions/31066/what-is-the-influence-of-c-in-svms-with-linear-kernel

[10]Measuring Performance: AUC (AUROC)
https://glassboxmedicine.com/2019/02/23/measuring-performance-auc-auroc/

[11]Mandrekar, J.N. (2010) 'Receiver operating characteristic curve in diagnostic test assessment', Journal of Thoracic Oncology, 5(9), pp. 1315–1316. doi:10.1097/jto.0b013e3181ec173d.

[12] How do you choose between bagging and boosting for your ML project?
https://www.linkedin.com/advice/0/how-do-you-choose-between-bagging-boosting-your#:~:text=Bagging%20is%20best%20for%20data,reduce%20underfitting%20and%20increase%20accuracy