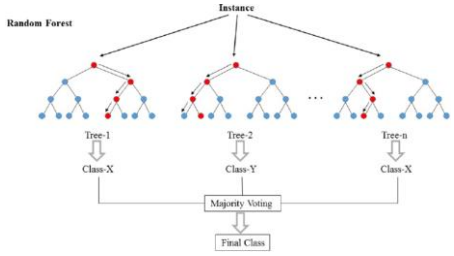

ENSEMBLE LEARNING

MACHINE LEARNING SERIES: ENSEMBLE LEARNING

This document focuses on understanding and creating Ensemble Models. This report will then cover Bagging, Pasting and Random Forests

Viraj Vaitha

Contents

Introduction

On the television show "who wants to be a millionaire" the candidate has an option to ask the audience for help. The audience vote which answer they believe is correct and the candidate will use the answers to help select his answer. The *wisdom of the crowd* can often lead to better success. To thought, immediately we know that they are not always right. This emphasizes that the individuals making up the audience must be knowledgeable in that area to increase the probability of success. In our case, the individual machine learning models are the audience. The ensemble of machine learning models (audience) will undergo a majority voting to determine what the correct value or class.

When we take the majority vote it is called **hard voting.** Even if the individual models are *weak learners (less than or equal to 50/50),* it can still produce a strong ensemble. Law of large number numbers takes effect here. The average individual machine learning models are in fact quite strong causing the ensemble to be strong too.

Build an Ensemble model (Hard Voting)

This data was from the moons dataset, please check the python file in GitHub for the full notebook. Hard Voting means we will take a majority vote between all of the models. It makes sense that aslong as each model has atleast more than 51% accuracy, we can expect a higher performance overall. A human analogy is that if we ask 5 people that are unsure and all guess the same answer there is a high chance that the chosen guess will be correct.

Import Relevant Packages

```
From sklearn.ensemble import RandomForestClassifier
From skleearn.ensemble import VotingClassifier
From sklearn.linear_model import LogisticRegression
From sklearn.svm import SVC
```

Initiate Model

```
Log_clf = LogisticRegression()
Rnd_clf = RandomForestClassifier()
Svm_clf = SVC()
```

Ensemble Models

```
Voting_clf = VotingClassifier(                    #Framework created to help with ensembles
        Estimators=[('lf', 'log_clf') , ('rf',rnd_clf), ('svc', svc_clf)], #acronyms for the models such as lf
        Voting ='hard')
```

Fit models to the data

```
Voting_clf.fit(X_train, y_train)
```

Viewing Metric of Individual Models

If we would like to view metrics of individual modes, we must loop through the models individually. As the voting classifier statistics would give you metrics for the overall new model.

```
From sklearn.metrics import accuracy_score

For clf in (log_clf, rnd_clf, svm_clf, voting_clf):

        Clf.fit(X_train, y_train)

        Y_pred = clf.predit(X_train, y_train)

        Print(clf.__class__.__name__, accuracy_score(y_test,y_pred))
```

Results

LogisticRegression = 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896

Soft Voting

The voting parameter was set to "hard". The alternative is "soft".

Soft Voting is when we take the average probability of all models to make our decision. Aslong as each model are able to estimate the probability of their prediction (they have the dict_proba method), we can use soft voting. In our above example SVC by default does not estimate probabilities therefore we must enter an argument into SVC. This argument is the probability hyperparameter set to true. (Unfortunately this will use cross validation to estimate probabilities and slow down the training process). Fortunately, by changing to "soft voting" we are able to gain over 91% accuracy !! Please see below

ENTER CODE HERE

```
Import Relevant Packages


Initiate and Fit the Model
```

Bagging and Pasting

So far we have obtained diversity through using different classifier algorithms. An alternative method is to train the same algorithm on different random subsets of the training set.

Sampling performed with replacement is known as bagging (aka bootstrap aggregating) and without replacement is known as pasting.

An example of bagging would be training 5 Logistic Classifiers on subsets of the data. In bagging some samples of data may appear in each of the 5 classifiers (This is useful as it helps compensate for small datasets). The ensemble aggregates all the predictions. For classifiers the general type is soft voting if the base classifier can estimate class probabilities. Regression would have an average of the predictions of each classifier.

While the bias of the individual models may be higher, the overall classifier is known to reduce both bias and variance. These models scale very well as they can be used on multiple CPUs.

Since the difference between Bagging and Pasting is quite vague here, I have found an answer of stackexchange to further clarify the differences.

ENTER ANSWER HERE

```
from sklearn.tree import export_graphviz

export_graphviz(tree_clf,
                out_file = "iris_tree.dot",
                feature_names = iris.feature_names[2:],
                class_names = iris.target_names,
                rounded = True,
                filled  = True)
#Find the file (When using Jupyter Notebooks, this was automatically saved in my repository notes

#Copy and Paste the contents of that file into the relevant section of this website.

http://www.webgraphviz.com/
```

Creating a Bagging Model

Import Relevant Packages

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
```

Initiate Model

```
Bag_clf = BaggingClassifier(
        DecisionTreeClassifier(), n_estimators =500,
        Max_samples=100, bootstrap =True, n_jobs=-1)
```

Fit the model to your data

```
Bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

If you would like to use pasting you set bootstrap = False. Pasting would cause each sample to be unique increasing the bias in the datasets (The reference states this but I think it depends on each case. I highly believe this could be used to reduce bias in certain cases). While bias is supposedly increased, the variance is reduced even more as each model is less correlated.

The graph below compares a single decision tree vs bagging ensemble of 500 trees. This shows how the model has comparable bias but a smaller variance as the decision boundaries are less irregular.

Out-of-Bag Evaluation

When bagging is used, it is common for some instances to be sampled several times for any given predictor while others may not be sampled at all. By default a BaggingClassifier samples m training instances with replacement (bootstrap=True), where m is the size of the training set.

This means that only 63% of the training instances are sampled on average for each predictor. The remaining 37% of the training instances that not sampled are called out-of-bag(oob) instances. Note that they are not the same 37% for all predictors.

Since a predictor never sees oob instances during training it can be evaluated on these instances without the need for a separate validation set or cross validation. Evaluate the model by averaging out oob instances of each predictor

Summary so far on OOB

- Data that has not been sampled is known as out of bag
- Oob instances can be used to validate your model (No need for CV ect..)
- Evaluate through an average on each predictor

What is Gini?

This is a measure of impurity

Gini = 0 = Pure

Gini = 1 = Impure

Gini is the composite of the leaf node. Let's say for versicolor, the Gini will account for how dominant the dominating class is.

$$G_i = 1 - \sum_{k=1}^{n} p_{i_k}^2$$

Where:

p_i is the ratio of class k instances in the training instances in the i^{th} node.

Therefore,

Gini for the Versicolor node = 1 - $(\frac{0}{54} + \frac{49}{54} + \frac{5}{54})$ = 0.168

Gini or Entropy?

Entropy, a term in thermodynamics as a measure of molecular disorder has been taken into different fields such as Shannon's information theory. In machine learning it is frequently used an impurity measure.

- They both tend to produce fairly similar results.
- Gini is faster to compute
- If they are different, entropy tends to create balanced trees whereas Gini isolates the most frequent class into its own leaf node.

Regularization

This is possible through limiting the maximum depth of the decision tree. They're other hyperparameters such as min_samples_split, min_samples_leaf and min_weight_fraction_leaf (same as min samples leaf but expressed as a fraction of the total number of weighted instances),max_leaf_nodes and max_features.

If we do not regularize the model for decision trees, it is known as nonparametric. While there still are parameters, the parameters are not predefined therefore training and the model is free and able to use as much depth ect..

This generally tends to overfitting therefore you should set some of the hyperparameters to restrict the model.

PCA Required

Decision trees are sensitive to rotation in the data. The orientation is extremely important since decision trees use orthogonal decision boundaries (all splits are perpendicular to the axis). PCA can be used to better orientate the data.

CART Algorithm

Classification and Regression Tree, a greedy algorithm which produces only binary trees (2 splits, true or false). The algorithm is trying to minimise the impurity, to obtain the purest subsets. The algorithm first splits the training set into two subsets using a single feature k and threshold t_k.

For example threshold would be less than 2.45 cm..

Once it has split the subset, it will repeat the process recursively.

$$J(k, t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{Right}$$

Where:

$G_{left/right}$ is a measure of the impurity of the left/right subset

$m_{left/right}$ is the number of instances in the left/right subset

$$J(k, t_k) = \frac{m_{left}}{m} MSE_{left} + \frac{m_{right}}{m} MSE_{right}$$

Where:

$MSE_{node} = \sum(y_{predicted} - y^{(i)})^2$

$Y_{predicted} = 1/m_{node} * y^i$

Comments

We can limit a lot of the instability related issues by using random forest as it takes a average prediction over many trees.

We must remember to initiate a random state.

Computational complexity is roughly $O(nxmlog_2(m))$. The NP-Complete problem is that we can't find the optimal tree since the CART algorithm searches for an optimum split at the top level and doesn't check whether or not the split will lead to the lowest possible impurity several level downs. The computational complexity makes it unfeasible for the alternative solution.

From	Import	Example	Notes
sklearn.linear_model	SGDClassifier	• Sgd_clf= SGDClassifier(random_state =42) • Sgd_clf.fit(X_train, y_train_5) • Sgd_clf.predict([Some_Digit])	This is a linear model. Random_state for reproducibility.(SGD operates on randomness) .fit arguments are training_set, labels .predict arguments is a matrix/vector [ ]
Sklearn.model_selection	StartifiedKfold	• skfolds = StratifiedKFold(n_splits=3,random_state=42) for train_index, test_index in skfolds.split(X_train, y_train_5): • See code in document to finish off CV with Stratified sampling	Shuffles the dataset in a way that makes each set contain a diverse range of possible values in each training set.
	cross_val_score	cross_val_score(sgd_clf, X_train,y_train_5, cv=3, scoring="accuracy")	Scoring has many possible arguments such as precision. Search online to see more. Arguments is the model, train, testset and cv folds.
	cross_val_predict	y_train_pred = cross_val_predict(sgd, X_train, y_train_5, cv=3) additional argument is method = “decision_function”.	Same as above. However this will return the predictions as opposed to the scoring. Decision_function will produce predictions that are a score in

			order calculate certain metrics while altering the threshold.
sklearn.metrics	precision_score,recall_score	precision_score(y_train_5, y_train_pred) recall_score(y_train_5, y_train_pred)	Arguments to pass are the training labels and the predictions
	f1_score	f1_score(y_train_5, y_train_pred)	
	precision_recall_curve	precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores) Graph def plt_precision_recall_vs_threshold(precisions,recalls,thresholds): plt.plot(thresholds,precisions[:-1],"b--",label="Precision") plt.plot(thresholds, recalls[:-1],"g-", label= "Recall") plt.xlabel("Threshold") plt.legend(loc="center left") plt.ylim([0,1]) plt_precision_recall_vs_threshold(precisions,recalls,thresholds) plt.show()	When you want to assess the impact of the threshold use this function. The first section of code will obtain the parameters you require for you. (p, r and t) The decision_function of cross_val_predict must be used to produce y_scores to use this.
	roc_curve	From sklearn.metrics import roc_curve Fpr,tpr,thresholds = roc_curve(y_train_5, y_scores)	

		Plotting the Graph def plot_roc_curve(fpr,tpr, label=none): plt.plot(fpr,tpr,linewidth = 2, label=label) plt.plot([0,1],[0,1], 'k - -') plt.axis([0,1,0,1]) plt.xlabel('False Positive Rate) plt.ylabel('True Positive Rate') plot_roc_curve(fpr,tpr) plt.show	
	Roc_auc_score	Roc_auc_score(y_train_5, y_scores)	Arguments to pass are the labels and score predictions from decision_function
sklearn			

Task	Do it	Don't do it
Shuffle Trainset	Image Classification,	Stock market prices, weather conditions
SGD Classifier	Large Datasets, online learning	Small Datasets