Chapter 7: Ensemble Learning and Random Forests

Aly Khaled

August 16, 2021

1 What is Ensemble Learning?

- Ensemble Learning in using more than one predictor then it aggregate the prediction of each predictor and predict the class that gets most votes.
- **Ensemble Learning** works best when the predictors are **independent** form each other as possible and we can achieve this by **training them using very different algorithms**.
- **Soft voting** is to predict the class with the highest class probability averaged over all the individual **but this need all classifier to be able to estimate class probabilities which means that they all have a *predict_proba()* method**
- **Soft voting** is often achieve higher performance than **hard voting** because it gives more weight to highly confident votes
- When we train all the predictors we can make prediction for a new instance by simply aggregating the predictions of all predictors using:
  - **Statistical mode** for classification
  - **Average** for regression
- Each individual predictor has a higher bias than if it were trained on the original training set but **aggregation** reduces both bias and variance and the **net result** is that the ensemble has a similar bias but a lower variance than a single predictors trained on the original training set

2 Bagging VS Pasting

- There are two approaches for ensemble learning.
  - **First:** Use different training algorithms.
  - **Second:** Use the same training algorithm for every predictor and train them on different random subsets of the training set.
- There are two types of sampling:
  - **Bagging:** When sampling is performed with replacement.
  - **Pasting:** When sampling is performed without replacement.
- Only **Bagging** allows training instances to be sampled several times for the same predictor .

- In **Scikit–Learn** the **BaggingClassifier** class automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities which means that it has a *predict_proba()* method.

- **Bagging** ends up with a slightly **higher bias** than pasting; but the extra diversity also means that the predictors end up being **less correlated** so the ensemble's variance is reduced.

- **Bagging** often results in better models.

- Its recommended if you have spare time and CPU power to use cross validation to evaluate both of bagging and pasting and select the one that works best.

- In **Scikit–Learn** the **BaggingClassifier** class by default it samples m training instances with replacement (*bootstrap=True*) where m is the size of the training set. This means only about 63% of the training set are sampled and the remaining 37% are called *(out–of–bag)* instances, so we can evaluate the predictor on these instances without making validation set

  - To evaluate the predictor on these instances add (*obb_score=True*) parameter and access it through (*oob_score_*) variable.

- In **Scikit–Learn** the **BaggingClassifier** class can sample features too and sampling is controlled by two hyperparameters: *max_features* and *bootstrap_features* and sampling features is useful when dealing with high-dimensional inputs (such as images).

- Sampling both training instances and features is called **Random Patches Method** but keeping all training instances and sampling features is called **Random Subspaces Method**

3 Random Forests

- **Random Forests** is an ensembles of **Decision Trees** and it trained using **Bagging** method but sometimes by **Pasting**.

- You can make **Random Forests** by using **BaggingClassifier** class but it recommended to make it using **RandomForestClassifier** class (or by **RandomForestRegressor** class for regression) because it more convenient and optimized for **Decision Trees**.

- **Random Forests** algorithm introduces extra randomness when growing tree.

- Instead of searching for the very best features when splitting a node. it searches for the best features among a random subset of features.

- The algorithm results in greater tree diversity and making overall better model.

- It is possible to make trees even more random by using random threshold for each feature rather than searching for best threshold and this tree called **Extremely Randomized Trees**.

- **Extremely Randomized Trees** trades more bias for a lower variance and is faster to train because searching for the best threshold is time-consuming.

- We can't tell which tree forest is better between **Extremely Randomized Trees** and **Random Forests** so its recommended to evaluate both of them and compare them using cross-validation.

4 Boosting

- **Boosting** is an ensemble method that combine several weak learners to a strong learner.
- There are many boosting algorithms:
  - AdaBoost.
  - Gradient Tree Boosting.
  - XGBoost.
- The general idea of boosting is to train predictors sequentially and each trying to correct its predecessor.

4.1 AdaBoosting

- **AdaBoost** technique is to pay more attention to the training instances that the predecessor underfitted
- **AdaBoost** steps are:
  1. Train a base classifier
  2. Increase the weight of the misclassified training instances.
  3. Train a second classifier using the updated weights.
- Once all predictors are trained, the ensemble makes predictions very much like bagging and pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.
- One important drawback to this sequential learning technique that it cannot be **parallelized** because each predictor depends on the previous one.
- **AdaBoost** algorithm is:
  1. Each instance weight $w^{(i)}$ is initially set to $\frac{1}{m}$, then we compute the weighted error rate for every predictor using this equation:

     $$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

  2. Then we compute the predictor's weight α using this equation

     $$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

  3. Then update the instance weight using this equation:

     $$w^{(i)} = \begin{cases} w^{(i)} & \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} exp(\alpha_j) & \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

  4. Then we normalize all the instance weights by dividing by $\sum_{i=1}^{m} w^{(i)}$

5. Then the whole process is repeated till the number of desired predictors is reached or when a perfect predictor is found.

- To make prediction the **AdaBoost** simply compute the predictions of all the predictors and weighs them using the predictors weights α_j, then the predicted class is the one that receives the majority of weighted votes

$$\hat{y}(x) = \underset{k}{argmax} \sum_{\substack{j=1 \\ \hat{y}_j = k}}^{N} \alpha_j$$

where N is the number of predictors.

- **Scikit–Learn** uses multiclass version of **AdaBoost** called **SAMME**.

- When there are only two classes **SAMME** is equivalent to **AdaBoost**.

- If the predictors can estimate their probabilities we can use a variant of **SAMME** called **SAMME.R** which relies on class probabilities rather than predictions and generally performs better.

- **Decision Stumps** is a tree composed of as single decision node plus two leaf nodes.

4.2 Gradient Boosting

- **Gradient Boosting** is similar to **AdaBoost** but instead of tweaking the instance weights at every iteration like **AdaBoost** does, this method tried to fit the new predictor to the residual errors made by the previous predictor.

- Using Gradient Boosting in regression is called **Gradient Tree Boosting** or **Gradient Boosted Regression Trees**.

- In **Scikit–Learn** you can use *(GradientBoostingRegressor)* class to train **GBRT** ensembles.

- The *learning_rate* hyperparameter scales the contribution of each tree. If you set it to a low value such as 0.1, you will need more trees but the prediction will generalize better and this generalization technique called *shrinkage*.

5 Stacking

- Its idea is instead of using trivial functions like hard voting to aggregate the predictions, we train a model to perform this aggregation.

- In stacking we take the prediction of all predictors as inputs to final predictor called **Blender**

- To train a **Blender**, a common approach is to use a hold-out set.

- To train a **Blender** first we split the training set to two subsets, then we use the first subset to train the predictors. Then the predictors are used to make predictions on the second set. We can create a new training set using these predicted values as input features and keeping the target value. The blender is trained on this new training set.