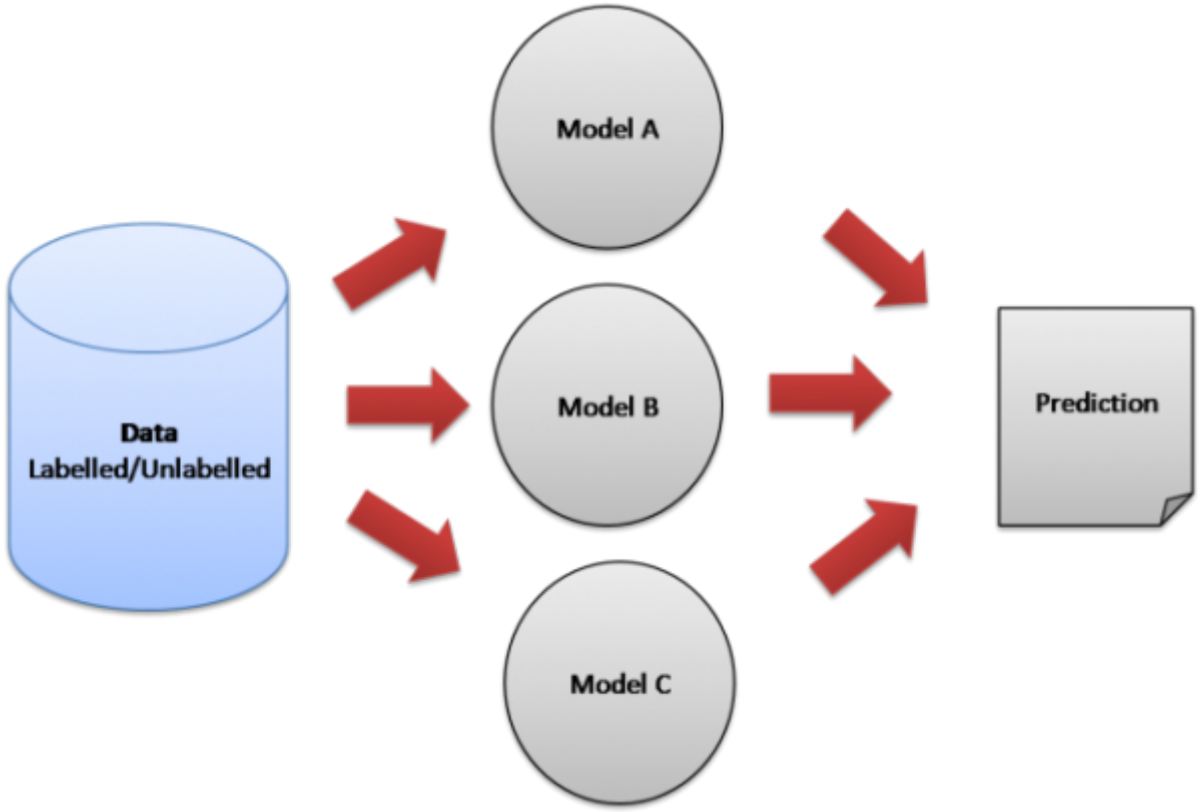
Building an Ensemble Learning Model Using Scikit-learn

Eijaz Allibhai
Nov 18, 2018 · 8 min read ★



Ensemble Learning (image credit)

Ensemble learning uses multiple machine learning models to try to make better predictions on a dataset. An ensemble model works by training different models on a dataset and having each model make predictions individually. The predictions of these models are then combined in the ensemble model to make a final prediction.

In this tutorial, we will be using a Voting Classifier in which the ensemble model makes the prediction by majority vote. For example, if we use three models and they predict [1, 0, 1] for the target variable, the final prediction that the ensemble model would make would be 1, since two out of the three models predicted 1.

We will use three different models to put into our Voting Classifier: k-Nearest Neighbors, Random Forest, and Logistic Regression. We will use the Scikit-learn library in Python to implement these methods and use the diabetes dataset in our example.

Note: Ensemble models can also be used for regression problems, where the ensemble model will use either the mean output of the different models or weighted averages for its final prediction.

Reading in the training data

The first step is to read in the data we will use as input. For this example, we are using the diabetes dataset. To start, we will use the Pandas library to read in the data.

```
import pandas as pd

#read in the dataset
df = pd.read_csv('data/diabetes_data.csv')

#take a look at the data
df.head()
```

the number of patients and the columns indicate the number of features (age, weight, etc.) in the dataset for each patient.

```
#check dataset size
df.shape
```


Split up the dataset into inputs and targets

Now let's split up our dataset into inputs (X) and our target (y). Our input will be every column except 'diabetes' because 'diabetes' is what we will be attempting to predict. Therefore, 'diabetes' will be our target.

We will use the pandas 'drop' function to drop the column 'diabetes' from our dataframe and store it in the variable 'X'.

```
#split data into inputs and targets
X = df.drop(columns = ['diabetes'])
y = df['diabetes']
```

Split the dataset into train and test data

Now we will split the dataset into into training data and testing data. The training data is the data that the model will learn from. The testing data is the data we will use to see how well the model performs on unseen data.

Scikit-learn has a function we can use called 'train_test_split' that makes it easy for us to split our dataset into training and testing data.

```
from sklearn.model_selection import train_test_split
```

'train_test_split' takes in 5 parameters. The first two parameters are the input and target data we split up earlier. Next, we will set 'test_size' to 0.3. This means that 30% of all the data will be used for testing, which leaves 70% of the data as training data for the model to learn from.

Setting 'stratify' to y makes our training split represent the proportion of each value in the y variable. For example, in our dataset, if 25% of patients have diabetes and 75% don't have diabetes, setting 'stratify' to y will ensure that the random split has 25% of patients with diabetes and 75% of patients without diabetes.

Building the models

Next, we have to build our models. Each model we build has a set of hyper parameters that we can tune. Tuning parameters is when you go through a process to find the optimal parameters for your model to improve accuracy. We will use grid search to find the optimal hyperparamters for each model.

Grid search works by training our model multiple times on a range of parameters that we specify. That way, we can test our model with each hyperparameter value and figure out the optimal values to get the best accuracy results.

k-Nearest Neighbors (k-NN)

The first model we will build is k-Nearest Neighbors (k-NN). k-NN models work by taking a data point and looking at the 'k' closest labeled data points. The data point is then assigned the label of the majority of the 'k' closest points.

For example, if k = 5, and 3 of points are 'green' and 2 are 'red', then the data point in question would be labeled 'green', since 'green' is the majority (as shown in the above graph).

Here is the code:

```
import numpy as np
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
```

```
#use gridsearch to test all values for n_neighbors
knn_gs = GridSearchCV(knn, params_knn, cv=5)

#fit model to training data
knn_gs.fit(X_train, y_train)
```

First, we will create a new k-NN classifier. Next, we need to create a dictionary to store all the values we will test for 'n_neighbors', which is the hyperparameter we need to tune. We will test 24 different values for 'n_neighbors'. Then we will create our grid search, inputing our k-NN classifier, our set of hyperparamters and our cross validation value.

Cross-validation is when the dataset is randomly split up into 'k' groups. One of the groups is used as the test set and the rest are used as the training set. The model is trained on the training set and scored on the test set. Then the process is repeated until each unique group as been used as the test set.

In our case, we are using 5-fold cross validation. The dataset is split into 5 groups, and the model is trained and tested 5 separate times so each group would get a chance to be the test set. This is how we will score our model running with each hyperparamter value to see which value for 'n_neighbors' gives us the best score.

Then we will use the 'fit' function to run our grid search.

Now we will save our best k-NN model to 'knn_best' using the 'best_estimator_' function and check what the best value was for 'n_neighbors'.

```
#save best model
knn_best = knn_gs.best_estimator_

#check best n_neigbors value
print(knn_gs.best_params_)
```

Random Forest

The next model we will build is a random forest. A random forest is considered an ensemble model in itself, since it is a collection of decision trees combined to make a more accurate model.

Here is the code:

```
from sklearn.ensemble import RandomForestClassifier

#create a new random forest classifier
rf = RandomForestClassifier()

#create a dictionary of all values we want to test for n_estimators
params_rf = {'n_estimators': [50, 100, 200]}

#use gridsearch to test all values for n_estimators
rf_gs = GridSearchCV(rf, params_rf, cv=5)

#fit model to training data
rf_gs.fit(X_train, y_train)
```

We will create a new random forest classifier and set the hyperparameters we want to tune. 'n_estimators' is the number of trees in our random forest. Then we can run our grid search to find the optimal number of trees.

Just like before, we will save our best model and print the best 'n_estimators' value.

```
#save best model
rf_best = rf_gs.best_estimator_

#check best n_estimators value
print(rf_gs.best_params_)
```

hyperparameters. We just need to create and train the model.

```
from sklearn.linear_model import LogisticRegression

#create a new logistic regression model
log_reg = LogisticRegression()

#fit the model to the training data
log_reg.fit(X_train, y_train)
```

Now let's check the accuracy scores of all three of our models on our test data.

```
#test the three models with the test data and print their accuracy
scores

print('knn: {}'.format(knn_best.score(X_test, y_test)))
print('rf: {}'.format(rf_best.score(X_test, y_test)))
print('log_reg: {}'.format(log_reg.score(X_test, y_test)))
```


As you can see from the output, logistic regression is the most accurate out of the three.

Voting Classifier

Now that we've built our three individual models, it's time we built our voting classifier.

Here is the code:

```
from sklearn.ensemble import VotingClassifier
```

```
#create our voting classifier, inputting our models
ensemble = VotingClassifier(estimators, voting='hard')
```

First, let's place our three models in an array called 'estimators'. Next, we will create our voting classifier. It takes two inputs. The first is our estimator array of our three models. We will set the voting parameter to hard, which tells our classifier to make predicitons by majority vote.

Now we can fit our ensemble model to our training data and score it on our testing data.

```
#fit model to training data
ensemble.fit(X_train, y_train)

#test our model on the test data
ensemble.score(X_test, y_test)
```

```
0.7705627705627706
```

Awesome! Our ensemble model performed better than our individual k-NN, random forest and logistic regression models!

That's it! You've now built an ensemble model to combine individual models!

Thanks for reading! The GitHub repository for this tutorial (jupyter notebook and dataset) can be found here.

If you would like to keep updated on my machine learning content, please follow me :)

Machine Learning Artificial Intelligence Data Science Python AI