Combining different machine learning algorithms into an ensemble model

Model ensembling is a class of techniques for aggregating together multiple different predictive algorithm into a sort of mega-algorithm, which can often increase the accuracy and reduce the overfitting of your model. Ensembling approaches often work surprisingly well. Many winners of competitive data science competitions [use](http://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf) [model](https://www.kaggle.com/c/amazon-employee-access-challenge/forums/t/5283/winning-solution-code-and-methodology) [ensembling](http://arxiv.org/pdf/0911.0460.pdf) in one form or another. In this tutorial, we will take you through the steps of building your own ensemble of a random forest, support vector machine, and neural network for doing a classification problem. We’ll be working on the famous spam dataset and trying to predict whether a certain email is spam or not, and using the standard Python machine learning stack (scikit/numpy/pandas/matplotlib).

You have probably already encountered several uses of model ensembling. Random forests are a type of ensemble algorithm that aggregates together many individual tree **base learners**. If you’re interested in deep learning, one common technique for improving classification accuracies is training different networks and getting them to vote on classifications for test instances (look at dropout for a related but wacky take on ensembling). If you’re familiar with bagging or boosting algorithms, these are very explicit examples of ensembling.

Regardless of the specifics, the general idea behind ensembling is this: different classes of algorithms (or differently parameterized versions of the same type of algorithm) might be good at picking up on different signals in the dataset. Combining them means that you can model the data better, leading to better predictions. Furthermore, different algorithms might be overfitting to the data in various ways, but by combining them, you can effectively average away some of this overfitting.

Let’s get started!

1. Loading up the data

Load dataset. We often want our input data to be a matrix (X) and the vector of instance labels as a separate vector (y).

We won’t do fancy visualizations of the dataset here. Check out this tutorial or our bootcamp to learn Plotly and matplotlib.

**Conclusion**

There are plenty of ways to do model ensembling. Simple **majority voting**. We can also do **weighted majority voting**, where models with higher accuracy get more of a vote. If your output is numerical, you could average. These relatively simple techniques do a great job, but there is more! Stacking (also called blending) is when the predictions from different algorithms are used as input into another algorithm (often good old linear and logistic regression) which then outputs your final predictions. For example, you might train a linear model on the predictions. Blending.

It is best to ensemble together models which are less correlated (see an excellent explanation [here](http://mlwave.com/kaggle-ensembling-guide/)).

See an excellent explanation of ensembling [here](http://mlwave.com/kaggle-ensembling-guide/).

What happens when your dataset isn’t as nice as this? What if there are many more instances of one class versus the other, or if you have a lot of missing values, or a mixture of categorical and numerical variables? Stay tuned for the next blog post where we write up guidance on tackling these types of sticky situations.

“The perceptron is a decent linear classifier which is guaranteed to find a separation if the data is linearly separable. This is a welcome property to have, but you have to realize a perceptron stops learning once this separation is reached. It does not necessarily find the best separation for new data.

So what would happen if we initialize 5 perceptrons with random weights and combine their predictions through an average? Why, we get an improvement on the test set!

“

2) Random forests

Next up, let’s build a random forest. A great explanation of random forests can be found here. Briefly, random forests build a collection of classification trees, which try to predict classes by recursively splitting the data on features that split classes best.

Let’s build a random forest with 500 decision trees.