

Stacking Classifiers for Higher Predictive Performance

 towardsdatascience.com/stacking-classifiers-for-higher-predictive-performance-566f963e4840

September 14, 2019

Responses



Using the Wisdom of Multiple Classifiers to Boost Performance

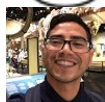




Photo by [Erwan Hesry](#) on [Unsplash](#)

Purpose: The purpose of this article is to provide the reader with the necessary tools to implement the ensemble learning technique known as stacking.

Materials and methods: Using [Scikit-learn](#), we generate a [Madelon-like data set](#) for a classification task. Then a Support Vector classifier (SVC), Nu-Support Vector classifier (NuSVC), a Multi-layer perceptron (MLP), and a Random Forest classifier will be individually trained. The performance of each classifier will be measured using the area under the receiver operating curve (AUC). Finally, we stack the predictions of these classifiers using the [StackingCVClassifier](#) object and compare the results.

Hardware: We train and evaluate our models on a workstation equipped with Inter(R)Core(TM) i7-8700 with 12 CPU @ 3.70 Ghz and NVIDIA GeForce RTX 2080.

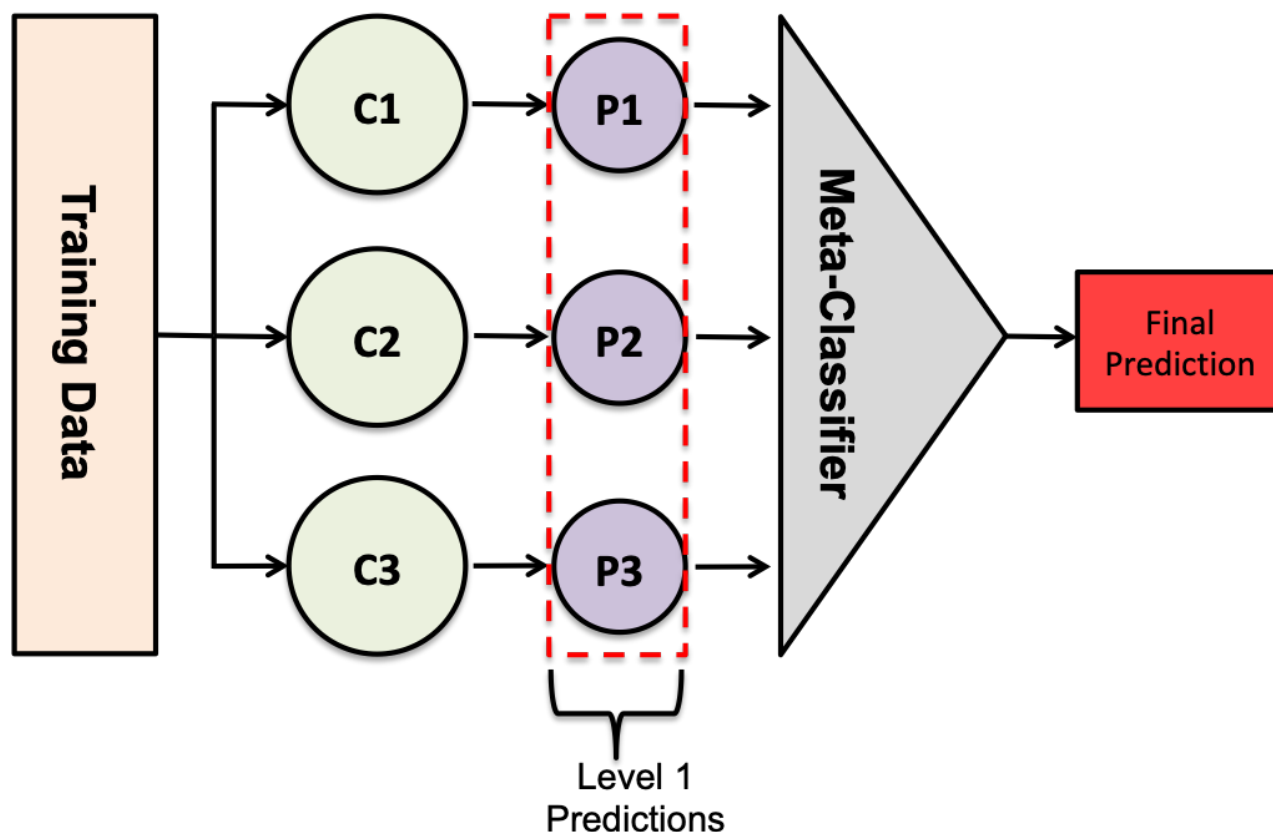
Note: In the case you're starting from scratch, I advise you follow this [article](#) to install all the necessary libraries. Finally, it will be assumed that the reader is familiar with Python, [Pandas](#), Scikit-learn, and ensemble methods. The whole contents of this article can be found on [my GitHub](#). You're welcomed to fork it.

Notation: Bold text will represent either a list, dictionary, tuple, a [Pandas DataFrame](#) object, or will be referring to a figure or script. **This notation will be used to represent parameters in an object or command lines to run in the terminal.**



What is Stacking?

The simplest form of stacking can be described as an ensemble learning technique where the predictions of multiple classifiers (referred as level-one classifiers) are used as new features to train a meta-classifier. The meta-classifier can be any classifier of your choice. **Figure 1** shows how three different classifiers get trained. Their predictions get stacked and are used as features to train the meta-classifier which makes the final prediction.



* C1, C2, and C3 are considered level 1 classifiers.

Figure 1 — Schematic of a stacking classifier framework. Here, three classifiers are used in the stack and are individually trained. Then, their predictions get stacked and are used to train the meta-classifier.

To prevent information from leaking into the training from the target (the thing you're trying to predict), the following rule should be followed when stacking classifiers:

1. The level one predictions should come from a subset of the training data that was not used to train the level one classifiers.

A simple way to achieve this is to split your training set in half. Use the first half of your training data to train the level one classifiers. Then use the trained level one classifiers to make predictions on the second half of the training data. These predictions should then be used to train meta-classifier.

A more robust way to do this, is to use k-fold cross validation to generate the level one predictions. Here, the training data is split into k-folds. Then the first k-1 folds are used to train the level one classifiers. The validation fold is then used to generate a subset of

the level one predictions. The process is repeated for each unique group. **Figure 2** illustrates this process.

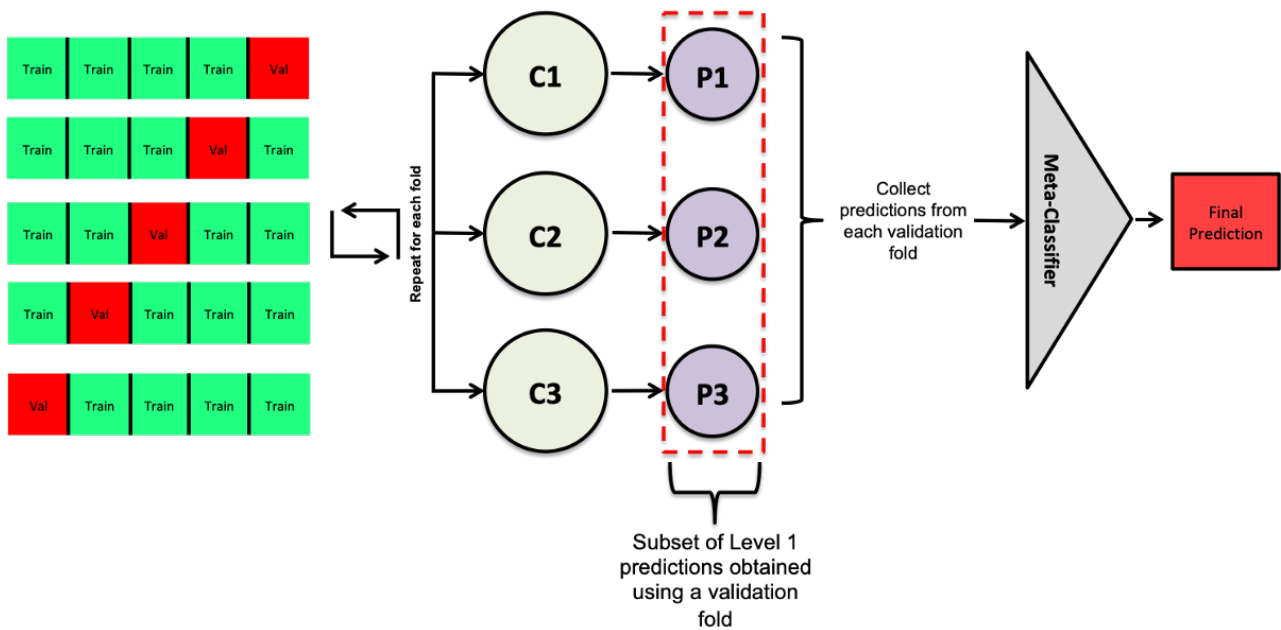


Figure 2 — Schematic of k-fold cross validation used with a Stacking classifier framework.

The avid reader might be wondering, why not stack another layer of classifiers? Well you can but this would add more complexity to your stack. For example consider the following architecture:

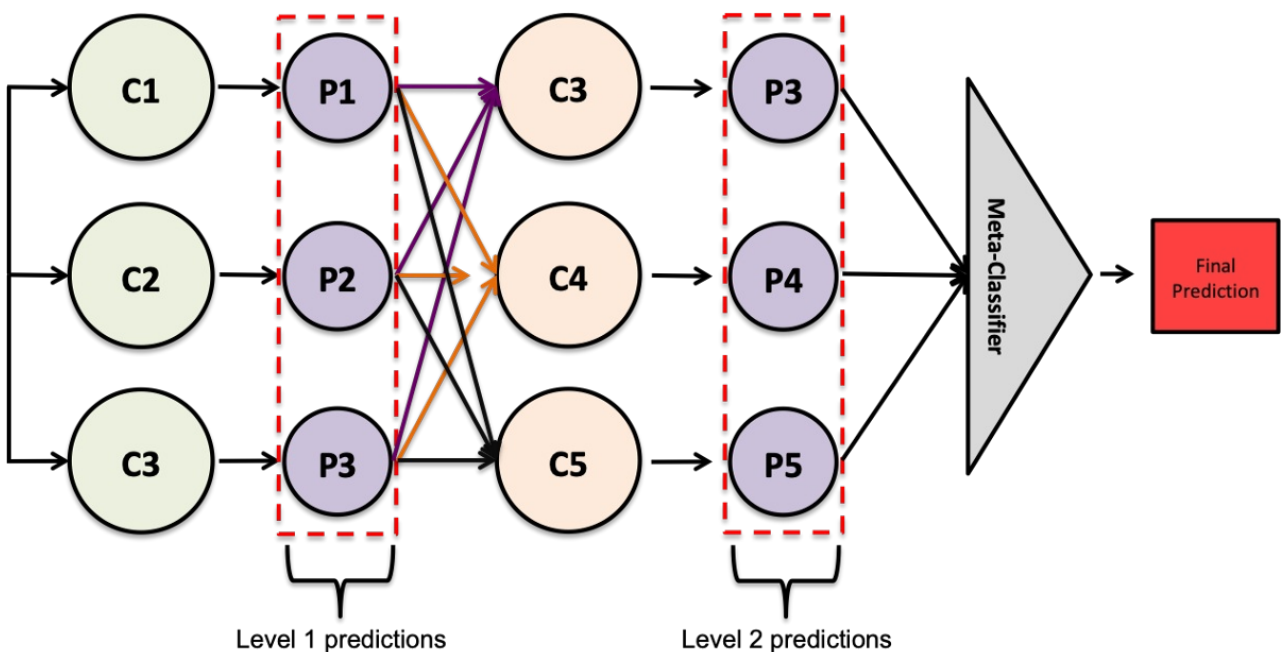


Figure 3 — Schematic of a Stacking classifier with two layers of classifiers.

This is starting to look like a neural network where each neuron is a classifier. The number and type of classifiers used in level two don't necessary need to be the same than the ones used in level one — see how things are starting to get out of hand real quick. For example, the level one classifiers can be an Extra Trees classifier, a Decision

Tree classifier, and a Support Vector Classifier, and the level two classifiers can be an artificial neural network, a Random Forest, and a Support Vector Classifier. In this article, we will implement the Stacking classifier architecture shown in **Figure 2**.



To Stack, or Not Stack, That is the Question.

Before you start stacking classifiers, it's advisable that you first consider all your other options to boost your predictive performance. As a guide, consider the following questions to determine if you should start stacking:

- Is it possible for you to get a hold of more data? In machine learning, data is king (that is the biggest boost in performance can be done by gathering more data); therefore, if it this is a possibility you should focus your efforts in putting together a bigger more diverse data set.
- Is it possible to add more features to your data set? For example, you can use the pixel intensity values in an image as features. You could also extract texture features from these images, which considers how pixel intensity values are spatially distributed. Is important that you collect as many features as possible from the problem at hand. This should be done simply because you do not know which of these features are good predictors. Remember, the predictive power of any model should mainly stem from the features used to train it and not from the algorithm itself.
- Have you correctly preprocess all of your data? If you push trash into a model, trash should come out. Make sure that you visually explore all of your features to gain a better understanding of them. Take care of missing values, outliers, and ensure that your data set is not grotesquely imbalanced. Scale your features, remove all redundant features, apply feature selection methods if your number of features is large, and consider feature engineering.
- Have you explored the predictive performance of a large set of classifiers with your data set? Have you carefully tuned these classifiers?

If your answer to all of this questions was a yes and you're willing to do anything for a possible boost in performance consider stacking.



Installing the Necessary Modules

The easiest way to implement the stacking architecture shown in **Figure 2** is to use the MLXTEND Python library. To install it read their GitHub ReadMe file found [here](#). If you have Anaconda on Windows, launch Anaconda prompt, navigate to the conda environment you want to install this module, and run the following command:

```
conda install -c conda-forge mlxtend
```


or you can use pip:

```
pip install mlxtend
```

Disclaimer: I'm almost certain this won't break your Python environment but if it does don't blame me. Here is a guide in [How to Install A Python Based Machine Learning Environment in Windows using Anaconda](#) for the interested reader.



Data Set

We will generate a Madelon-like synthetic data set using Scikit-learn for a classification task. This is the same data set I used to explore the performance of various models in a [previous article](#) I published in Medium. The Madelon data set is an artificial data set that contains 32 clusters placed on the vertices of a five-dimensional hyper-cube with sides of length 1. The clusters are randomly labeled 0 or 1 (2 classes).

The data set that we will generate will contain 30 features, where 5 of them will be informative, 15 will be redundant (but informative), 5 of them will be repeated, and the last 5 will be useless since they will be filled with random noise. The columns of the data set will be ordered as follows:

1. **Informative features — Columns 1–5:** These features are the only features you really need to build your model. Hence, a five-dimensional hyper-cube.
2. **Redundant features — Columns 6–20.** These features are made by linearly combining the informative features with different random weights. You can think of these as engineered features.
3. **Repeated features — Columns 21–25:** These features are drawn randomly from either the informative or redundant features.
4. **Useless features — Columns 26–30.** These features are filled with random noise.

Let's start by importing the libraries.

Script 1 — Importing all libraries

Now we can generate the data.

Script 2 —Generates the data set. The random_state parameter is fixed so that we can work with the same data.



Splitting and Preprocessing Data

Once we have our data, we can proceed with creating a training and test set.

Script 3— Create train and test set. Notice we set the random_state parameter.

Now we can scale our data set. We will do so via Z-score normalization.

Script 4 — Scaling the data.

We are done preprocessing the data set. In this example, we won't go through the trouble of removing highly correlated, redundant, and noisy features. However, it is important that you take care of this in any project you might be tackling.



Classifiers

For the purpose of illustration, we will train a Support Vector Classifier (SVC), Multi-layer Perceptron (MLP) classifier, Nu-Support Vector classifier (NuSVC), and a Random Forest (RF) classifier— classifiers available in Scikit-learn. The hyper-parameters for these classifiers will be set — we will assume that we did our best tuning them. To put it bluntly, if your classifiers are trash, a stack of them would probably be trash too. Additionally, to obtain reproducible results, the `random_state` on each of these classifiers has been fixed.

Script 5— Initializing classifiers.



Stacking Classifier

To stack them, we will use the `StackingCVClassifier` from `MLXTEND`. I will recommend that you take a look at the [official documentation](#) since it goes in detail over useful examples of how to implement the `StackingCVClassifier`. For example, it shows you how to stack classifiers that operate on different feature subsets — that's so cool. Finally, we will set the `use_probab` parameter in `StackingCVClassifier` to `True`. This means that the classifiers in the first level will pass predictions that are probabilities and not a binary output (0 or 1). However, this does not mean you should use this setting, it might be more beneficial for your work if you set `use_probab` to `False`; however, the only way to know that is to actually do it. As the `meta_classifier` we select a SVC and use it's default parameters.

Script 6— Initializing the stacking classifier.

Finally, for convenience, we put all the classifiers in a dictionary.

Script 7— Saving classifiers in a dictionary.



Training Classifiers

We train each classifier and save it into dictionary labeled **classifiers**. After a classifier is trained, it will be saved into the same dictionary.

Making Predictions

Now we are ready to make predictions on the test set. Here we first create **results**, a pandas DataFrame object. Then using a for loop, we iterate through the trained classifiers and store the predictions in **results**. Finally, in line 14, we add a new column labeled “Target” with the actual targets, either

Script 9— Making predictions on test set.

Visualizing Results

Now is time to visualize the results. We will do so by creating a figure with 5 subplots. On each subplot, we will show the probability distribution obtained on the test set from each classifier. We will also report on the area under the receiver operating curve (AUC).

Script 10 — Visualizing results.

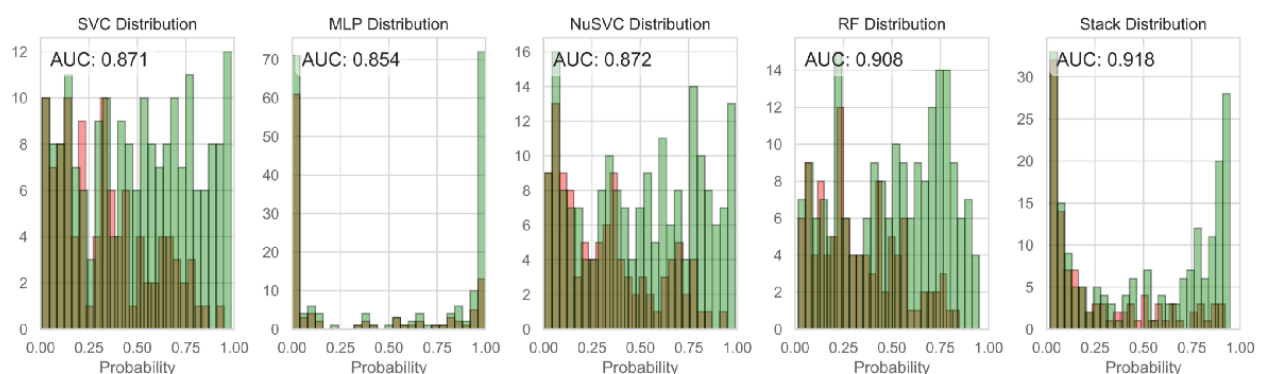


Figure 4 — Probability distribution for each classifier. The AUC is also reported.

Nice! Each classifier individually obtained an AUC less than 0.909; however, by stacking them we obtained an AUC = 0.918. That’s a pretty nice boost don’t you think?

Tuning the Stacking Classifier

The StackingCVClassifier offers you the option to tune the parameters of each of the classifiers as well as the meta-estimators. However, I do not recommend to do a full blown search with the StackingCVClassifier. For example, suppose that you wanted to use four classifiers each with a parameter grid of 100 points — a fairly small grid to be honest. How long would it take to explore this space? Let’s do a simple calculation:

$100 * 100 * 100 * 100 = 100$ million models you have to fit

Assuming that each model takes 0.5 second to fit, that would take 50 million seconds or more simply put 1.6 years. Nobody has time for that. This calculation did not consider the fact that the meta-classifier is not fixed and contains its own parameters that need to be tuned. That's why I would recommend that you first tune your classifiers individually. After that, you can stack them and only tune the meta-classifier. If you do decide to tune the stack of classifiers, make sure you tune about the optimal point in parameter space for each one of them.

In this example, we first create a parameter grid for the meta-classifier — a Support Vector Classifier. Then we conduct an exhaustive search using [GridSearchCV](#) to determine the point in the parameter grid that yields the highest performance. The results are printed out to the console.

Script 11 — Tuning the meta-classifier.

The AUC of the tuned Stacking classifier is 0.923

Nice, we manage to push the Stacking classifier AUC from 0.918 to 0.923 by tuning the meta-classifier. However, we are not done here. Let's see if we can get more out of the Stacking classifier. To do so, we will create a set of Stacking classifiers each with a unique combination/stack of classifiers in the first layer. For each Stacking classifier, the meta-classifier will be tuned. The results will be printed out to the console.

Script 12— Stacking different classifiers and exploring their performance.

```
AUC of stack ('SVC', 'MLP'): 0.876
AUC of stack ('SVC', 'NuSVC'): 0.877
AUC of stack ('SVC', 'RF'): 0.877
AUC of stack ('MLP', 'NuSVC'): 0.876
AUC of stack ('MLP', 'RF'): 0.877
AUC of stack ('NuSVC', 'RF'): 0.877
AUC of stack ('SVC', 'MLP', 'NuSVC'): 0.875
AUC of stack ('SVC', 'MLP', 'RF'): 0.876
AUC of stack ('SVC', 'NuSVC', 'RF'): 0.879
AUC of stack ('MLP', 'NuSVC', 'RF'): 0.875
AUC of stack ('SVC', 'MLP', 'NuSVC', 'RF'): 0.923
```

Well it turns out that we obtain the highest AUC if we stack all of the classifiers together. However, this will not always be the case. Sometimes stacking less classifiers will yield a higher performance. It actually took me a whole day to figure out what classifiers to stack together in order to get this boost. Additionally, I noticed that sometimes by stacking I obtained lower performances. The trick is to stack classifiers that make up for the errors of each other and not work against each other.



Closing Remarks


I started this article to introduce the reader to technique of stacking. It was suppose to be short and quick but it turned to be quite long. If you made it this far, I appreciate your time. I hope this is of help in your endeavors. I would like to point out that stacking classifiers might not always work and it can be a tedious process for a small boost; however, if done correctly and luck is on your side, you will be able to gain a boost in performance. I recommend that you first explore other options to boost your performance. The process of stacking should be left to the end since it can be time consuming and the rewards might be minimal. Finally, if you stack trash, is very likely that trash will come out. The whole contents of this article can be found [here](#). All the best and keep learning!

You can find me in [LinkedIn](#).

Sign up for The Daily Pick

By Towards Data Science

Hands-on real-world examples, research, tutorials, and cutting-edge techniques delivered Monday to Thursday. Make learning your daily ritual. [Take a look](#)

 Get this newsletter

By signing up, you will create a Medium account if you don't already have one. Review our [Privacy Policy](#) for more information about our privacy practices.



359



2



in



More From Medium

[10 Cool Python Project Ideas for Python Developers](#)

Claire D. Costa in [Towards Data Science](#)



Develop and sell a Python API—from start to end tutorial

Daniel Deutsch in [Towards Data Science](#)



Farewell RNNs, Welcome TCNs

Bryan Tan in [Towards Data Science](#)



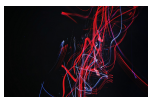
Building a Face Recognizer in Python

Behic Guven in [Towards Data Science](#)



Machine Learning With SQL—It's Easier Than You Think

Dario Radečić in [Towards Data Science](#)



Stop One-Hot Encoding Your Categorical Variables.

Andre Ye in [Towards Data Science](#)



5 Reasons why you should Switch from Jupyter Notebook to Scripts

Khuyen Tran in [Towards Data Science](#)



VSCode Jupyter Notebooks are Getting an Upgrade

Richard So in [Towards Data Science](#)

