How to Implement Stacked Generalization (Stacking) From Scratch With Python

Code a Stacking Ensemble From Scratch in Python, Step-by-Step.

Ensemble methods are an excellent way to improve predictive performance on your machine learning problems.

Stacked Generalization or stacking is an ensemble technique that uses a new model to learn how to best combine the predictions from two or more models trained on your dataset.

In this tutorial, you will discover how to implement stacking from scratch in Python.

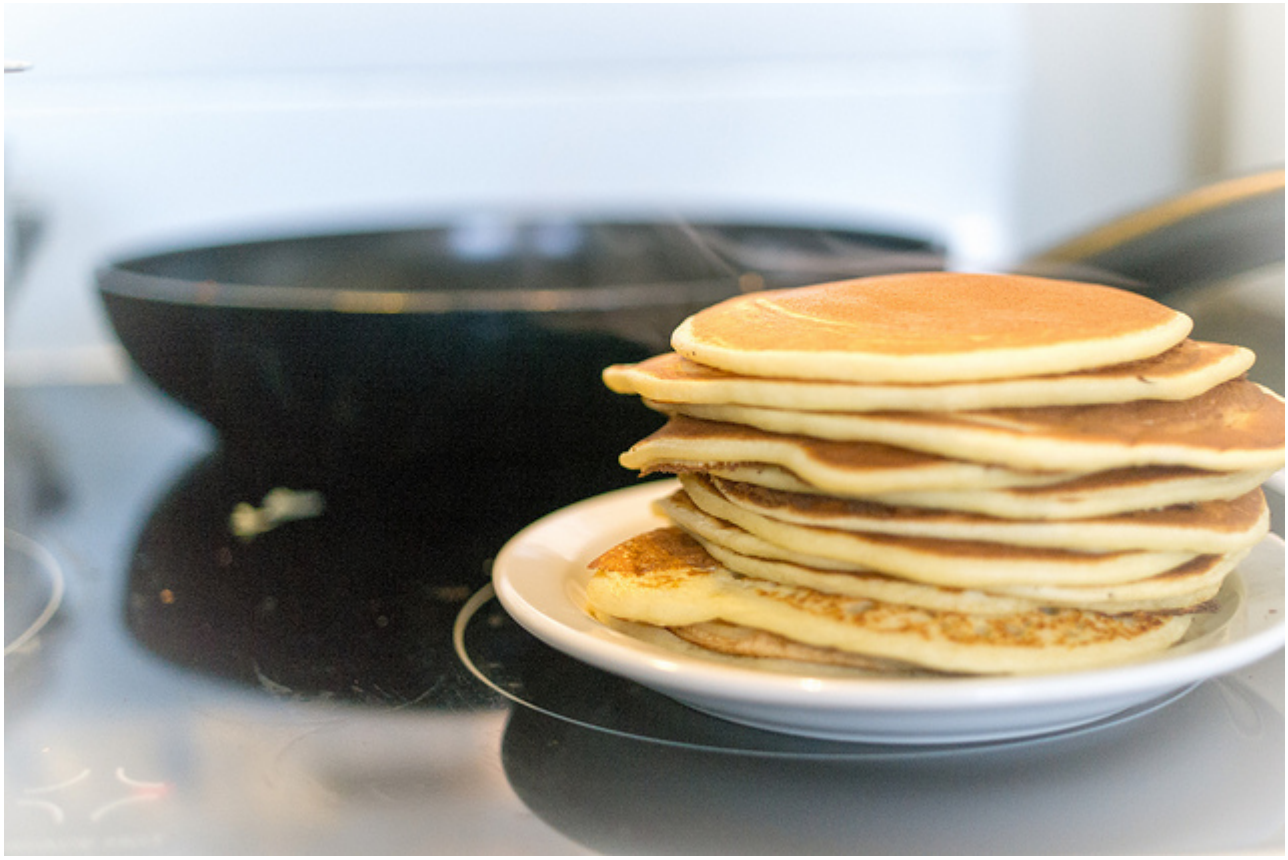
After completing this tutorial, you will know:

- How to learn to combine the predictions from multiple models on a dataset.

- How to apply stacked generalization to a real-world predictive modeling problem.

Let's get started.

- **Update Jan/2017**: Changed the calculation of fold_size in cross_validation_split() to always be an integer. Fixes issues with Python 3.
- **Update Aug/2018**: Tested and updated to work with Python 3.6.



Description

This section provides a brief overview of the Stacked Generalization algorithm and the Sonar dataset used in this tutorial.

Stacked Generalization Algorithm

Stacked Generalization or stacking is an ensemble algorithm where a new model is trained to combine the predictions from two or more models already trained or your dataset.

The predictions from the existing models or submodels are combined using a new model, and as such stacking is often referred to as blending, as the predictions from sub-models are blended together.

It is typical to use a simple linear method to combine the predictions for submodels such as simple averaging or voting, to a weighted sum using linear regression or logistic regression.

Models that have their predictions combined must have skill on the problem, but do not need to be the best possible models. This means that you do not need to tune the submodels intently, as long as the model shows some advantage over a baseline prediction.

It is important that sub-models produce different predictions, so-called uncorrelated predictions. Stacking works best when the predictions that are combined are all skillful, but skillful in different ways. This may be achieved by using algorithms that use very different internal representations (trees compared to instances) and/or models trained on different representations or projections of the training data.

In this tutorial, we will look at taking two very different and untuned sub-models and combining their predictions with a simple logistic regression algorithm.

Sonar Dataset

The dataset we will use in this tutorial is the Sonar dataset.

This is a dataset that describes sonar chirp returns bouncing off different surfaces. The 60 input variables are the strength of the returns at different angles. It is a binary classification problem that requires a model to differentiate rocks from metal cylinders. There are 208 observations.

It is a well-understood dataset. All of the variables are continuous and generally in the range of 0 to 1. The output variable is a string "M" for mine and "R" for rock, which will need to be converted to integers 1 and 0.

By predicting the class with the most observations in the dataset (M or mines) the Zero Rule Algorithm can achieve an accuracy of about 53%.

You can learn more about this dataset at the UCI Machine Learning repository.

Download the dataset for free and place it in your working directory with the filename **sonar.all-data.csv**.

Tutorial

This tutorial is broken down into 3 steps:

1. Sub-models and Aggregator.
2. Combining Predictions.
3. Sonar Dataset Case Study.

These steps provide the foundation that you need to understand and implement stacking on your own predictive modeling problems.

1. Sub-models and Aggregator

We are going to use two models as submodels for stacking and a linear model as the aggregator model.

This part is divided into 3 sections:

1. Sub-model #1: k-Nearest Neighbors.
2. Sub-model #2: Perceptron.
3. Aggregator Model: Logistic Regression.

Each model will be described in terms of the functions used train the model and a function used to make predictions.

1.1 Sub-model #1: k-Nearest Neighbors

The k-Nearest Neighbors algorithm or kNN uses the entire training dataset as the model.

Therefore training the model involves retaining the training dataset. Below is a function named **knn_model()** that does just this.

123	# Prepare the kNN modeldef knn_model(train): return train

Making predictions involves finding the k most similar records in the training dataset and selecting the most common class values. The Euclidean distance function is used to calculate the similarity between new rows of data and rows in the training dataset.

Below are these helper functions that involve making predictions for a kNN model. The function **euclidean_distance()** calculates the distance between two rows of data, **get_neighbors()** locates all neighbors for in the training dataset for a new row of data and **knn_predict()** makes a prediction from the neighbors for a new row of data.

12345678910111213141516171819202122232425	# Calculate the Euclidean distance between two vectorsdef euclidean_distance(row1, row2): distance = 0.0 for i in range(len(row1)-1): distance += (row1[i] - row2[i])**2 return sqrt(distance) # Locate neighbors for a new rowdef get_neighbors(train, test_row, num_neighbors): distances = list() for train_row in train: dist = euclidean_distance(test_row, train_row) distances.append((train_row, dist)) distances.sort(key=lambda tup: tup[1]) neighbors = list() for i in range(num_neighbors): neighbors.append(distances[i][0]) return neighbors # Make a prediction with kNNdef knn_predict(model, test_row, num_neighbors=2): neighbors = get_neighbors(model, test_row, num_neighbors) output_values = [row[-1] for row in neighbors] prediction = max(set(output_values), key=output_values.count) return prediction

You can see that the number of neighbors (k) is set to 2 as a default parameter on the **knn_predict()** function. This number was chosen with a little trial and error and was not tuned.

Now that we have the building blocks for a kNN model, let's look at the Perceptron algorithm.

1.2 Sub-model #2: Perceptron

The model for the Perceptron algorithm is a set of weights learned from the training data.

In order to train the weights, many predictions need to be made on the training data in order to calculate error values. Therefore, both model training and prediction require a function for prediction.

Below are the helper functions for implementing the Perceptron algorithm. The **perceptron_model()** function trains the Perceptron model on the training dataset and **perceptron_predict()** is used to make a prediction for a row of data.

123456789101112131415161718	**# Make a prediction with weightsdef perceptron_predict(model, row): activation = model[0] for i in range(len(row)-1): activation += model[i + 1] * row[i] return 1.0 if activation >= 0.0 else 0.0 # Estimate Perceptron weights using stochastic gradient descentdef perceptron_model(train, l_rate=0.01, n_epoch=5000): weights = [0.0 for i in range(len(train[0]))] for epoch in range(n_epoch): for row in train: prediction = perceptron_predict(weights, row) error = row[-1] - prediction weights[0] = weights[0] + l_rate * error for i in range(len(row)-1): weights[i + 1] = weights[i + 1] + l_rate * error * row[i] return weights**

The **perceptron_model()** model specifies both a learning rate and number of training epochs as default parameters. Again, these parameters were chosen with a little bit of trial and error, but were not tuned on the dataset.

We now have implementations for both sub-models, let's look at implementing the aggregator model.

1.3 Aggregator Model: Logistic Regression

Like the Perceptron algorithm, Logistic Regression uses a set of weights, called coefficients, as the representation of the model.

And like the Perceptron algorithm, the coefficients are learned by iteratively making predictions on the training data and updating them.

Below are the helper functions for implementing the logistic regression algorithm. The **logistic_regression_model()** function is used to train the coefficients on the training dataset and **logistic_regression_predict()** is used to make a prediction for a row of data.

123456789101112131415161718	**# Make a prediction with coefficientsdef logistic_regression_predict(model, row): yhat = model[0] for i in range(len(row)-1): yhat += model[i + 1] * row[i] return 1.0 / (1.0 + exp(-yhat)) # Estimate logistic regression coefficients using stochastic gradient descentdef logistic_regression_model(train, l_rate=0.01, n_epoch=5000): coef = [0.0 for i in range(len(train[0]))] for epoch in range(n_epoch): for row in train: yhat = logistic_regression_predict(coef, row) error = row[-1] - yhat coef[0] = coef[0] + l_rate * error * yhat * (1.0 - yhat) for i in range(len(row)-1): coef[i + 1] = coef[i + 1] + l_rate * error * yhat * (1.0 - yhat) * row[i] return coef**

The **logistic_regression_model()** defines a learning rate and number of epochs as default parameters, and as with the other algorithms, these parameters were found with a little trial and error and were not optimized.

Now that we have implementations of sub-models and the aggregator model, let's see how we can combine the predictions from multiple models.

2. Combining Predictions

For a machine learning algorithm, learning how to combine predictions is much the same as learning from a training dataset.

A new training dataset can be constructed from the predictions of the sub-models, as follows:

- Each row represents one row in the training dataset.
- The first column contains predictions for each row in the training dataset made by the first sub-model, such as k-Nearest Neighbors.
- The second column contains predictions for each row in the training dataset made by the second sub-model, such as the Perceptron algorithm.
- The third column contains the expected output value for the row in the training dataset.

Below is a contrived example of what a constructed stacking dataset may look like:

123456	**kNN, Per, Y0, 0 01, 0 10, 1 01, 1 10, 1 0**

A machine learning algorithm, such as logistic regression can then be trained on this new dataset. In essence, this new meta-algorithm learns how to best combine the prediction from multiple submodels.

Below is a function named **to_stacked_row()** that implements this procedure for creating new rows for this stacked dataset.

The function takes a list of models as input, these are used to make predictions. The function also takes a list of functions as input, one function used to make a prediction for each model. Finally, a single row from the training dataset is included.

A new row is constructed one column at a time. Predictions are calculated using each model and the row of training data. The expected output value from the training dataset row is then added as the last column to the row.

12345678	**# Make predictions with sub-models and construct a new stacked rowdef to_stacked_row(models, predict_list, row): stacked_row = list() for i in range(len(models)): prediction = predict_listi stacked_row.append(prediction) stacked_row.append(row[-1]) return stacked_row**

On some predictive modeling problems, it is possible to get an even larger boost by training the aggregated model on both the training row and the predictions made by sub-models.

This improvement gives the aggregator model both the context of all the data in the training row to help determine how and when to best combine the predictions of the sub-models.

We can update our **to_stacked_row()** function to include this by aggregating the training row (minus the final column) and the stacked row as created above.

Below is an updated version of the **to_stacked_row()** function that implements this improvement.

12345678	**# Make predictions with sub-models and construct a new stacked rowdef to_stacked_row(models, predict_list, row): stacked_row = list() for i in range(len(models)): prediction = predict_listi stacked_row.append(prediction) stacked_row.append(row[-1]) return row[0:len(row)-1] + stacked_row**

It is a good idea to try both approaches on your problem to see which works best.

Now that we have all of the pieces for stacked generalization, we can apply it to a real-world problem.

3. Sonar Dataset Case Study

In this section, we will apply the Stacking algorithm to the Sonar dataset.

The example assumes that a CSV copy of the dataset is in the current working directory with the filename **sonar.all-data.csv**.

The dataset is first loaded, the string values converted to numeric and the output column is converted from strings to the integer values of 0 to 1. This is achieved with helper functions **load_csv()**, **str_column_to_float()** and **str_column_to_int()** to load and prepare the dataset.

We will use k-fold cross validation to estimate the performance of the learned model on unseen data. This means that we will construct and evaluate k models and estimate the performance as the mean model error. Classification accuracy will be used to evaluate the model. These behaviors are provided in the **cross_validation_split()**, **accuracy_metric()** and **evaluate_algorithm()**helper functions.

We will use the k-Nearest Neighbors, Perceptron and Logistic Regression algorithms implemented above. We will also use our technique for creating the new stacked dataset defined in the previous step.

A new function name **stacking()** is developed. This function does 4 things:

1. It first trains a list of models (kNN and Perceptron).
2. It then uses the models to make predictions and create a new stacked dataset.
3. It then trains an aggregator model (logistic regression) on the stacked dataset.
4. It then uses the sub-models and aggregator model to make predictions on the test dataset.

The complete example is listed below.

1234567891011121314151617181920212223242526272829303132333435363738394041424344454647484950515253545556575859606162636465666768697071727374757677787980818283848586878889909192939495969798991001

A k value of 3 was used for cross-validation, giving each fold 208/3 = 69.3 or just under 70 records to be evaluated upon each iteration.

Running the example prints the scores and mean of the scores for the final configuration.

12	**Scores: [78.26086956521739, 76.81159420289855, 69.56521739130434]Mean Accuracy: 74.879%**

Extensions

This section lists extensions to this tutorial that you may be interested in exploring.

- **Tune Algorithms**. The algorithms used for the submodels and the aggregate model in this tutorial were not tuned. Explore alternate configurations and see if you can further lift performance.
- **Prediction Correlations**. Stacking works better if the predictions of submodels are weakly correlated. Implement calculations to estimate the correlation between the predictions of submodels.
- **Different Sub-models**. Implement more and different sub-models to be combined using the stacking procedure.
- **Different Aggregating Model**. Experiment with simpler models (like averaging and voting) and more complex aggregation models to see if you can boost performance.
- **More Datasets**. Apply stacking to more datasets on the UCI Machine Learning Repository.

Did you explore any of these extensions? Share your experiences in the comments below.

Review

In this tutorial, you discovered how to implement the stacking algorithm from scratch in Python.

Specifically, you learned:

- How to combine the predictions from multiple models.
- How to apply stacking to a real-world predictive modeling problem.