Stacked Generalization (StackGen)

Author: Mahesh Raja
Date: May 15th, 2017

This report contains details about the StackGen Python package which implements the stacked generalization technique in machine learning. It contains the following sections -

1. Introduction
2. Dataset and Metrics
3. Solution Statement and Benchmarks
4. Implementation, Examples and Code Walkthrough
5. Conclusion and Future Work

1. Introduction

1.1 Domain Background

In the seminal paper titled Stacked Generalization by David H. Wolpert[1], he introduced an idea for combining one or more machine learning models using a meta-learner which resulted in smaller generalization error than any single model used in the combination. Though this idea has been around for sometime, it gained significant popularity in recent times thanks to Kaggle, the data science competitions platform. Almost every winning solution to a Kaggle competition uses the idea stacked generalization or as it is colloquially called - "stacking". Having recently participated in the largest internal (for companies) Kaggle competition myself (AIG Data Science Challenge), I heavily used stacking to finish 8th in a competition that had 820 participants from 24 different countries.

1.2 Problem Statement

Implementing stacking for a supervised learning problem can be a tedious and confusing process (I learnt this the hard way). It involves generating predictions from several base models and using these predictions or "meta-features" as inputs to train a stacking model that combines these predictions to reduce the overall generalization error. Through this capstone project, I aim to develop a Python package that removes the tediousness and hides the complexity behind this idea. The package will provide an easy to use interface for users who want to combine many models and perform stacked generalization to improve prediction accuracies for their supervised learning problem. Efforts were made to ensure that the interface for the package is as familiar as possible to scikit-learn[2] learn users. (eg. fit_predict() methods would be used in the package for fitting models and predicting target values).

---

[1] *Stacked Generalization by David H. Wolpert in Neural Networks December 1992*

[2] *Scikit-learn*

1.3 Application to real world datasets

To demonstrate the application of the stacked generalization technique, we will look two datasets - one for each type of supervised learning i.e. classification and regression.

For the classification case, the dataset we will be working with is the UCI Cardiotocography (CTG) dataset which is described in detail in the next section (2). This is a medical dataset with 21 continuous valued features, which are medical measurements, and output is fetal state which is a discrete variable with 3 classes. Our aim is to train several simple base models (classifiers) and combine their class predictions using a stacking model to see if this will result in lower generalization error than any of the individual base models. Theory behind stacked generalization suggests that this should be the case!

For the regression case, we will be working with the famous Boston Housing Dataset which has 13 features describing houses in Boston suburbs and the target variable is the median value of the house. Our aim here is to train several simple base models (regressors) and combine their house value predictions using a stacking model to see if this will reduce our overall error. The generalization error at the end of the stacking process should be lower than the generalization errors of any of the base models.

1.4 Solution Strategy

The solution would involve building a Python package, using object oriented methodology that provides a simple, familiar API to perform stacked generalization. The solution object would take in various parameters necessary for the stacked generalization process and would return the final test/out of sample (OOS) predictions. It would also give the user options to save their solutions.

API for the package would look like this -

stacked_model = StackGen(base_models = [list of base models], stacker = Stacker_model, classification = True/False, n_folds = integer, stratified = True/False, kf_random_state = integer, save_results = 0,1,2 or 3, stack_with_orig = True/False)

The stacked generalization object would be initialized with a bunch of different parameters including a list of base models, the stacking model, number of folds of cross validation and so on. These parameters are discussed in detail in section 4.

Train and test data can be passed to the object above by calling the following method, which is familiar to scikit-learn users -

final_results = stacked_model.fit_predict(X_train, y_train, X_test, y_test)

Website - http://scikit-learn.org/stable/about.html

The above method returns the final test/OOS predictions from the stacking model. If stacked generalization is a success, the generalization error for the stacker model's OOS predictions should be lower than the generalization error of any of the base models used in the stacking process.

2. Datasets and Metrics

Stacked generalization is much like a "black art" like many concepts in machine learning. Though the idea of stacked generalization can be applied to most supervised learning problems, the degree of success various by the problem at hand. We will focus on two datasets in particular in order to demonstrate that the stacking model may outperform the simpler base models. Please note that there will be **no data preprocessing performed as that is not the focus of our project**.

2.1 Datasets

UCI Cardiotocography (CTG) dataset for classification[3]
This is a medical dataset with 21 continuous valued features and multiclass output with 3 classes. 2126 fetal cardiotocograms (CTGs) were automatically processed and the respective diagnostic features measured. The CTGs were also classified by three expert obstetricians and a consensus classification label assigned to each of them. Classification was both with respect to a morphologic pattern (A, B, C. ...) and to a fetal state (N, S, P). For the purposes of our project, we only consider the fetal state (N, S, P) as target class where Normal(N)=1, Suspect(S)=2, Pathologic(P)=3 in the dataset. The data will be used as it is i.e. no preprocessing will be performed.

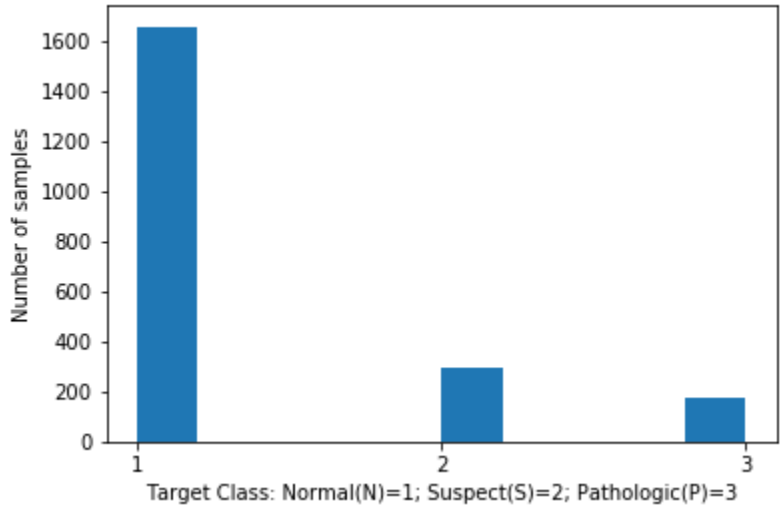

Figure 1 – Distribution of target variable NSP

[3] *Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.*
Website - https://archive.ics.uci.edu/ml/datasets/Cardiotocography

From the above histogram, it is clear that the 3 target classes in our dataset are unequally distributed. In order to avoid imbalance during sampling, we will follow a stratified method to split our data into train-test splits and also use stratified k-fold cross validation in order to replicate the conditions of our original data.

Here is a sample of the dataset –

	LB	AC	FM	UC	DL	DS	DP	ASTV	MSTV	ALTV	...	Min	Max	Nmax	Nzeros	Mode	Mean	Median	Variance	Tendency	NSP
0	120	0.000	0.0	0.000	0.000	0.0	0.0	73	0.5	43	...	62	126	2	0	120	137	121	73	1	2
1	132	0.006	0.0	0.006	0.003	0.0	0.0	17	2.1	0	...	68	198	6	1	141	136	140	12	0	1
2	133	0.003	0.0	0.008	0.003	0.0	0.0	16	2.1	0	...	68	198	5	1	141	135	138	13	0	1
3	134	0.003	0.0	0.008	0.003	0.0	0.0	16	2.4	0	...	53	170	11	0	137	134	137	13	1	1
4	132	0.007	0.0	0.008	0.000	0.0	0.0	16	2.4	0	...	53	170	9	0	137	136	138	11	1	1

Here are the descriptive statistics of the dataset –

	LB	AC	FM	UC	DL	DS	DP	ASTV	MSTV	ALTV
count	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.00000
mean	133.303857	0.003178	0.009481	0.004366	0.001889	0.000003	0.000159	46.990122	1.332785	9.84666
std	9.840844	0.003866	0.046666	0.002946	0.002960	0.000057	0.000590	17.192814	0.883241	18.39688
min	106.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	12.000000	0.200000	0.00000
25%	126.000000	0.000000	0.000000	0.002000	0.000000	0.000000	0.000000	32.000000	0.700000	0.00000
50%	133.000000	0.002000	0.000000	0.004000	0.000000	0.000000	0.000000	49.000000	1.200000	0.00000
75%	140.000000	0.006000	0.003000	0.007000	0.003000	0.000000	0.000000	61.000000	1.700000	11.00000
max	160.000000	0.019000	0.481000	0.015000	0.015000	0.001000	0.005000	87.000000	7.000000	91.00000

Min	Max	Nmax	Nzeros	Mode	Mean	Median	Variance	Tendency	NSP
2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000
93.579492	164.025400	4.068203	0.323612	137.452023	134.610536	138.090310	18.808090	0.320320	1.304327
29.560212	17.944183	2.949386	0.706059	16.381289	15.593596	14.466589	28.977636	0.610829	0.614377
50.000000	122.000000	0.000000	0.000000	60.000000	73.000000	77.000000	0.000000	-1.000000	1.000000
67.000000	152.000000	2.000000	0.000000	129.000000	125.000000	129.000000	2.000000	0.000000	1.000000
93.000000	162.000000	3.000000	0.000000	139.000000	136.000000	139.000000	7.000000	0.000000	1.000000
120.000000	174.000000	6.000000	0.000000	148.000000	145.000000	148.000000	24.000000	1.000000	1.000000
159.000000	238.000000	18.000000	10.000000	187.000000	182.000000	186.000000	269.000000	1.000000	3.000000

Figure 2 – Descriptive statistics for UCI CTG dataset

Boston housing dataset for regression[4]

This is a classic dataset for regression with a sample size of 506. It has 13 features describing houses in the Boston suburb area and the target value is the median value of a house in $1000's. The dataset comes standard with Python's scikit-learn package and can easily be imported for testing. The data will be used as it is i.e. no preprocessing will be performed.

[4] *http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_boston.html*

Here is a sample of the dataset –

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	Median Value of House
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Here are the descriptive statistics of the dataset –

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.593761	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534
std	8.596783	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000
75%	3.647423	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000

B	LSTAT	Median Value of House
506.000000	506.000000	506.000000
356.674032	12.653063	22.532806
91.294864	7.141062	9.197104
0.320000	1.730000	5.000000
375.377500	6.950000	17.025000
391.440000	11.360000	21.200000
396.225000	16.955000	25.000000
396.900000	37.970000	50.000000

Figure 3 – Descriptive statistics for Boston Housing dataset

2.2 Metrics

Log Loss - In case of classification, the evaluation metric we will use to compare the models will be multiclass logarithm loss[5].

[5] *Logarithmic Loss – Kaggle*
Website - *https://www.kaggle.com/wiki/LogLoss*

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\log(p_{i,j})$$

Where *N* is the number of observations, *M* is the number of class labels, *log* is the natural logarithm, $y_{i,j}$ is 1 if observation *i* is in class *j* and 0 otherwise, and $p_{i,j}$ is the predicted probability that observation *i* is in class *j*.

The metric log loss is chosen because of its peculiar behavior – it heavily penalizes a classifier when it predicts the wrong class with very high probability. This encourages classifiers to strive for better class probabilities. Misclassification rate (0 for correct class and 1 for incorrect class) and other similar metrics simply view the predictions in a black and white world. Log loss however sees how close a classifier came to predicting the true class for each sample. Another reason log loss is chosen over other "black and white" error metrics like misclassification rate is because of the evident class imbalance in the dataset. Over 1600 of the 2126 samples belong to class 1 (or label N). A classifier that always predicts a class of 1 for all test samples would have a really low misclassification rate which doesn't help our purpose. However, the same classifier would have really high log loss if it always predicts class 1 (i.e. probability of 0.999 for class 1, 0.001 for class 2, 0.001 for class 3) for all test samples.

Mean Squared Error (MSE) - In case of regression, we will use the simple mean squared error as evaluation metric.

MSE = $\frac{1}{n}\sum_{i=1}^{n}(yhat_i - y_i)^2$

where *yhat* is a vector of n predictions, *y* is the vector of n observed values / true target values corresponding to the inputs to the model which generated the predictions and *i* is the observation number ranging from 1 to n.

MSE was chosen as the metric mainly because of its simplicity. This is a metric that many machine learning practitioners, both beginner and advanced alike, are familiar with. Moreover, changes in generalization error of MSE are more pronounced when compared with root mean squared error for example where the changes in generalization error are in smaller units (because of the square root). This would help demonstrate the boost that stacked generalization technique provides to generalization error (i.e. how much it reduces it)!

3. Solution Statement and Benchmarks

General procedure for stacked generalization is explained in the following steps. The practical implementation was learned in part from Kaggle grandmaster Triskelion [6] -

---

[6] *Kaggle Ensembling Guide – Kaggle user Triskelion*

Step 1: Split the dataset into training split **TR** (with target values **Y**) and testing split **TE** (with corresponding target values **Y'**). In real world scenarios, the test split is ideally the unseen or out of sample data that we want to perform predictions for.

Step 2: Learn several generalizers on the train dataset **TR** and output **Y**. Let's call these generalizers as level-0 or base generalizers. Let the generalizers learnt in this step be $h_1,\ h_2, h_3 \ldots . h_n$ where n is the number of generalizers.

Each generalizer is trained in a k-fold cross-validation fashion. The generalizer is trained on k-1 folds of the train data **TR** and makes a prediction on the remaining fold. Training is repeated such that the generalizer makes predictions on all folds exactly once. All the out-of-fold predictions are stacked together to form an intermediate dataset. Let's call this dataset $\mathbf{TR_{h1}}$ for generalizer h1. For our purposes, we can also this dataset the output of generalizer h_1.

During each iteration of the k-fold cross validation, a generalizer that is trained on k-1 folds of the train dataset TR also makes predictions on the test dataset **TE**. Predictions on dataset **TE** are made exactly k times (number of folds) and these predictions are averaged to form another intermediate dataset. Let's call this dataset $\mathbf{TE_{h1}}$ for generalizer h_1.

Step 2: Construct a new dataset **TR'** based on outputs generated by the base generalizers in **Step 1**. The new dataset (also called meta-features) is constructed by concatenating (horizontally stacking) the outputs of generalizers $h_1,\ h_2, h_3 \ldots . h_n$ as follows **TR'** = ($\mathbf{TR_{h1}}$, $\mathbf{TR_{h2}}$, ….$\mathbf{TR_{hn}}$)

Similarly construct a new test dataset **TE'** is generated by concatenating the test dataset outputs generated in **Step 1** as follows **TE'** = ($\mathbf{TE_{h1}}$,$\mathbf{TE_{h2}}$,…$\mathbf{TE_{hn}}$)

Note that datasets **TR'** and **TE'** would have the same number of columns! These datasets would serve as the new train and test sets for the stacking model respectively.

Step 3: Learn a "stacking" model (which is also a generalizer) on newly constructed train dataset **TR'** as input and **Y** as output using cross-validation. Use this trained model to make predictions on the newly constructed test set **TE'**.

The term "generalizer" in the above steps is a supervised learning model that learns from provided data and "generalizes" on unseen samples.

In case of a regression problem, Step 1 would be a set of regression models and the newly constructed dataset **TR'** in Step 2 would include all continuous valued predictions of models from Step 1. For example, in case of the Boston housing dataset, three different regression models might predict median values of houses as (in $1000's) 5, 6.5 and 6.2 for a particular sample. Then the corresponding entry in **TR'** for that sample would look like (5, 6.5, 6.2)

Website- https://mlwave.com/kaggle-ensembling-guide/

In case of a classification problem, Step 1 would be a set of classifiers and the newly constructed dataset **TR'** in Step 2 would be a concatenation of the probabilities of all classes predicted by every classifier in Step 1. For example, in case of the CTG dataset, three different classification models might predict the class probabilities for the same sample as (0.3, 0.3, 0.4), (0.9, 0.05, 0.05) and (0.5, 0.4, 0.1). Then the corresponding entry in **TR'** for that sample would look like (0.3, 0.3, 0.4, 0.9, 0.05, 0.05, 0.5, 0.4, 0.1)

The package aims to provide implementations for following special cases of stacked generalization -

Using a "stacking" or level-1 model to combine base generalizers – The predictions from level-0 models are given as inputs (also called "meta-features") to a level-1 stacking model.

Using a "stacking" model to combine base generalizers while including original features along with meta-features for the stacking model. – The predictions from level-0 models are horizontally stacked along with the original features. This combination of original features and meta-features are given as inputs to a level-1 stacking model.

Model averaging – As the name suggests, this involves averaging the outputs of all the level-0 or base generalizers and it's the simplest way of combining models.

3.1 Benchmarks

For the purposes of our project, we will consider the individual level-0 or base models as benchmark models. Our stacked model should outperform its component level-0 models. Please note that we will also only use benchmarks for "**Using a "stacking" or level-1 model to combine base generalizers"** case from the above segment as mentioned in the proposal.

4. Implementation, Examples and Code Walkthrough

The following sections contain examples for classification and regression problems in which we aim to apply the techniques of stacked generalization to improve our generalization scores!

4.1 Classification Case

4.1.1 Model averaging

We first import our stacked generalization package StackGen and other required packages.

```
>>> from code_folder.StackGen import StackGen
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.linear_model import LogisticRegression
```

```
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.model_selection import train_test_split
>>> import numpy as np

>>> data = np.genfromtxt('CTG - required cols with NSP.csv', delimiter=',')

>>> X = data[1:,0:21]
>>> Y = (data[1:, 21:22]).flatten()

>>> X_TR, X_TE, y_TR, y_TE = train_test_split(X, Y, test_size=0.3, stratify = Y,
>>>                                           random_state=9)
```

In the code above, we first import the UCI CTG dataset described in the previous section and create the X and Y variables. In the next step, we split the samples and target values into train X_TR and test X_TE splits in a 7:3 ratio. The test split X_TE in our problem will actually serve as the out of sample (OOS) or unseen data that we want our stacked generalization technique to provide superior predictions for! We also stratify the splits in order to replicate the original dataset and avoid any class imbalance. A seed is also set to ensure that the results are reproducible. As mentioned previously, we do not perform any preprocessing on the dataset as it is not in the scope of our project.

Next we initialize the StackGen object and pass the required parameters

```
>>> stacked_classifier = StackGen(base_models = [KNeighborsClassifier(n_neighbors=10),
>>> LogisticRegression(solver = 'newton-cg'), GaussianNB()], stacker = None, classification =
>>> True, n_folds = 5, stratified = True, kf_random_state = 9, save_results = 0,
>>> stack_with_orig = False)
```

Base models: We pass three models as our level-0 or base models – K-Neighbors Classifier with k=10, a simple Logistic Regressions classifier with a solver used for multiclass classification and Gaussian Naïve Bayes classifier with default parameters. The reason these models are chosen is that they are very simple classifiers which are pretty diverse in their techniques. Our goal is to show that simple classifiers such as the ones chosen above can also be 'stacked' to achieve higher predictive accuracy.

Stacker: None. We first run the stacking process without any stacker model and simply average the outputs of the base models to see if this reduces generalization error.

classification: True. This flag indicates that the problem is indeed a classification problem.

stratified: True. This flag tells the StackGen object to use Stratified K-fold for the cross validation splits instead of normal K-fold. This is to avoid class imbalance.

n_folds: 5. This is the number of folds used in stratified k-fold cross validation

kf_random_state: 9. This is a seed for the stratified k-fold object. This is set in order to replicate results.

save_results: 0. This parameter indicates that results do not need to be saved.

stack_with_orig: False. This parameter tells the StackGen object to NOT include the old train and test/OOS data while creating the new train and test/oos sets.

Next, we run the stacked generalization process by calling the fit_predict() method and pass the train data, train labels, test/OOS data, test/OOS labels.

```
>>> stacked_classifier.fit_predict(X_TR, y_TR, X_TE, y_TE)
```

On calling the fit_predict() method, we get the following results –

For the KNeighbors classifier with k=10, log loss is as follows

```
Average CV error is  0.80678020795
OOS error --->  0.339359399417
```

For Logistic Regression classifier, log loss is as follows

```
Average CV error is  0.333615969715
OOS error --->  0.292313606775
```

For Gaussian Naïve Bayes classifier, log loss is as follows

```
Average CV error is  1.40093140504
OOS error --->  1.07114753643
```

Averaging the test/oos predictions of these three base models since the stacker model was not provided, log loss is as follows

```
OOS error --->  0.261039325565
```

These results are in line with our expectations. The final OOS error obtained after averaging the individual OOS predictions of base models is smaller than the individual OOS errors of the base models themselves!

4.1.2 Using a "stacking" or level-1 model to combine base generalizers

Keep in mind that the above result was simply obtained by averaging the oos predictions. We next try to include a stacker model in our process that combines the base models' predictions.

```
>>> stacked_classifier = StackGen(base_models = [KNeighborsClassifier(n_neighbors=10),
>>> LogisticRegression(solver = 'newton-cg'), GaussianNB()], stacker =
>>> RandomForestClassifier(n_estimators = 300, random_state= 9), classification = True,
>>> n_folds = 5, stratified = True, kf_random_state = 9, save_results = 0, stack_with_orig =
>>> False)
```

The only difference here from the previous case is that we have included a 'stacker' model in our list of parameters. The stacking model is a Random Forest Classifier with 300 estimators and no tuning whatsoever. A random state has been set in order replicate results. Now we run the fit_predict() method and obtain the following results.

For the stacker model Random Forest Classifier, log loss is as follows

```
Average CV error is  0.308288635808
OOS error --->  0.210399051654
```

This OOS error is a decent boost from our previous result where we just averaged the predictions of the base models! Let's see if we can further improve our generalization error.

4.1.3 Using a "stacking" model to combine base generalizers and using original features along with meta-features for the stacking model.

As the title suggests, we now make a small tweak in our parameters and set the stack_with_orig parameter to True. This ensures that the new train and oos datasets that are generated from the predictions of the base models include the original features (horizontally stacked) along with the newly generated meta-features! Let's see how this performs.

```
>>> stacked_classifier = StackGen(base_models = [KNeighborsClassifier(n_neighbors=10),
>>> LogisticRegression(solver = 'newton-cg'), GaussianNB()], stacker =
>>> RandomForestClassifier(n_estimators = 300, random_state= 9), classification = True,
>>> n_folds = 5, stratified = True, kf_random_state = 9, save_results = 0, stack_with_orig =
>>> True)
```

On running fit_predict() method we get the following result -

For the stacker model Random Forest Classifier (stacked with original features included) , log loss is as follows

```
Average CV error is  0.222072317406
OOS error --->  0.160545275301
```

This OOS error is a huge boost when compared with all of our base models! It also out-performs the previous case where the original features were not included for stacking. This is a pretty handy technique used in Kaggle competitions to improve generalization error and move up the leaderboard.

Here is a summary of the OOS errors for the base models and stacker models

Model	Type	OOS error (Log Loss)
KNeighborsClassifier(n_neighbors=10),	Base Model	0.3394
LogisticRegression(solver = 'newton-cg')	Base Model	0.2923
GaussianNB()	Base Model	1.0712

Average of base models' OOS preds	Stacking Model	0.2610
RandomForestClassifier(n_estimators = 300, random_state = 9) Meta-features do NOT include original features	Stacking Model	0.2104
RandomForestClassifier(n_estimators = 300, random_state =9) Meta-features include original features	Stacking Model	0.1605

4.1.4 Robustness of Solution

In the words of David Wolpert, stacked generalization is like "black magic". This essentially means that a model pipeline i.e. a list of base models along with a stacker model that worked really well with a particular dataset may not do so well with a different dataset! There is no one solution fits all in stacked generalization. However, we can consider our solution to be robust if it consistently gives low generalization error even for different samples of the dataset (i.e. split using different seeds) and different samples sizes (i.e. different train test split sizes).

Let's try demonstrating this concept for our classification dataset.

Case 1: Train-Test split ratio for original data = 7:3; seed = 9

```
>>> X_TR, X_TE, y_TR, y_TE = train_test_split(X, Y, test_size=0.3, stratify = Y,
>>>                                           random_state=9)
```

This is same as the scenario described in the above sections. The data is fit to the following stacking pipeline, same as before.

```
>>> stacked_classifier = StackGen(base_models = [KNeighborsClassifier(n_neighbors=10),
>>> LogisticRegression(solver = 'newton-cg'), GaussianNB()], stacker =
>>> RandomForestClassifier(n_estimators = 300, random_state= 9), classification = True,
>>> n_folds = 5, stratified = True, kf_random_state = 9, save_results = 0, stack_with_orig =
>>> True)

>>> stacked_classifier.fit_predict(X_TR, y_TR, X_TE, y_TE)
```

For the KNeighbors classifier with k=10, log loss is as follows

```
OOS error --->  0.339359399417
```

For Logistic Regression classifier, log loss is as follows

```
OOS error --->  0.292313606775
```

For Gaussian Naïve Bayes classifier, log loss is as follows

```
OOS error --->  1.07114753643
```

For the stacker model Random Forest Classifier (stacked with original features included) , log loss is as follows

```
OOS error --->  0.160545275301
```

Case 2: Train-Test split ratio for original data = 7:3; seed = 17

Now, let's change the seed for the train-test split keeping the split ratio same. This would essentially give us different samples than before in our train and test datasets. Rest of the stacking pipeline remains as it.

For the KNeighbors classifier with k=10, log loss is as follows

```
OOS error --->  0.443181735888
```

For Logistic Regression classifier, log loss is as follows

```
OOS error --->  0.315728142612
```

For Gaussian Naïve Bayes classifier, log loss is as follows

```
OOS error --->  1.49903175983
```

For the stacker model Random Forest Classifier, log loss is as follows

```
OOS error --->  0.171572992408
```

The OOS error of the stacker is still lower than the OOS errors of the individual base models.

Case 3: Train-Test split ratio for original data = 7:3; seed = 42

Let's do this exercise one more time – change the seed and run the stacking model again.

For the KNeighbors classifier with k=10, log loss is as follows

```
OOS error --->  0.578855080985
```

For Logistic Regression classifier, log loss is as follows

```
OOS error --->  0.342431030996
```

For Gaussian Naïve Bayes classifier, log loss is as follows

```
OOS error --->  1.61026525996
```

For the stacker model Random Forest Classifier, log loss is as follows

```
OOS error --->  0.186444828049
```

Like the above cases, the OOS error of the stacker is still lower than the OOS errors of the individual base models.

Case 4: Train-Test split ratio for original data = 1:1; seed = 786

This time, let's change the train-test split ratio to 1:1 and the seed to 786 –

For the KNeighbors classifier with k=10, log loss is as follows
```
OOS error --->  0.620564379996
```

For Logistic Regression classifier, log loss is as follows
```
OOS error --->  0.357591477825
```

For Gaussian Naïve Bayes classifier, log loss is as follows
```
OOS error --->  1.4125918585
```

For the stacker model Random Forest Classifier, log loss is as follows
```
OOS error --->  0.200845069036
```

The OOS error of the stacker is still lower than the OOS errors of the individual base models.

Case 5: Train-Test split ratio for original data = 3:7; seed = 786

Finally time, let's change the train-test split ratio to 3:7 and check if the stacking model still out performs the base models –

For the KNeighbors classifier with k=10, log loss is as follows
```
OOS error --->  0.670567307101
```

For Logistic Regression classifier, log loss is as follows
```
OOS error --->  0.349176714413
```

```
For Gaussian Naïve Bayes classifier, log loss is as follows
OOS error --->  1.52662547717
```

For the stacker model Random Forest Classifier, log loss is as follows
```
OOS error --->  0.225062561967
```

Even in this case, where the train and test/OOS datasets are in 3:7 ratio, the stacking model continues to outperform the individual base models.

The above cases prove that our solution is pretty robust to changes in the dataset (size, split-seed etc.) given we follow the same model pipeline.

4.2 Regression Case

Now, let's look at an example where we apply stacked generalization for a regression problem!

```
>>> from code_folder.StackGen import StackGen
>>> from sklearn.linear_model import Ridge, Lasso
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.model_selection import train_test_split
>>> from sklearn import datasets

>>> boston = datasets.load_boston()
>>> X = boston.data
>>> Y = boston.target
>>> X_TR, X_TE, y_TR, y_TE = train_test_split(X, Y, test_size=0.3, random_state=9)
```

We first load StackGen package and other required packages. For the regression problem, we use the Boston Housing dataset as mentioned before. No data preprocessing is done. The data is then split into train and test datasets in a 7:3 ratio.

4.2.1 Using a "stacking" or level-1 model to combine base generalizers

```
>>> stacked_regressor = StackGen([Ridge(), Lasso()],stacker =
>>> RandomForestRegressor(random_state = 9), classification = False, n_folds = 3,
>>> kf_random_state = 9, stack_with_orig = False, save_results = 0)
```

Similar to the classification scenario, we pass two base models to the StackGen object – Ridge and Lasso regression models with default parameters. Again, these models are chosen for their simplicity and to demonstrate the fact that simple models such as these can be combined using the stacked generalization technique to improve the prediction accuracy!

This time, the 'classification' parameter is set to False as this is a regression task. Number of folds in k-fold cross validation is set to 3.

Let's call the fit_predict() method and pass the train and test/OOS data

```
>>> stacked_regressor.fit_predict(X_TR, y_TR, X_TE, y_TE)
```

We get the following mean squared errors for different models -

For Ridge regression, the mean squared error is

```
Average CV error is  26.4588225668
OOS error --->  19.8432255394
```

For Lasso, the mean squared error is

```
Average CV error is  30.4266279898
OOS error --->  26.9220710546
```

For the stacker – Random Forest Regressor, the mean squared error is

```
Average CV error is  30.5799313559
OOS error --->  16.0319447368
```

The OOS error for the stacking model is lower than the two base models from our above results. Let's see if we can further lower this error!

4.2.2 Using a "stacking" model to combine base generalizers and using original features along with meta-features for the stacking model.

```
>>> stacked_regressor = StackGen([Ridge(), Lasso()],stacker =
>>> RandomForestRegressor(random_state = 9), classification = False, n_folds = 3,
>>> kf_random_state = 9, stack_with_orig = True, save_results = 0)
```

We change the stack_with_orig parameter to True and see the results of fit_predict().

For the stacker – Random Forest Regressor, the mean squared error is

```
Average CV error is  17.3340596045
OOS error --->  11.7684244883
```

This OOS error is significantly lower than the base models' OOS errors and also lower than the case where stacking was performed without including the original features with meta-features dataset for the stacking model. This indicates that our stacked generalization process is a success!

Here is a summary of the OOS errors for the base models and stacker models

Model	Type	OOS error (MSE)
Ridge()	Base Model	19.8432
Lasso()	Base Model	26.9221
RandomForestRegressor(random_state = 9) Meta-features do NOT include original features	Stacking Model	16.0319
RandomForestRegressor(random_state = 9) Meta-features include original features	Stacking Model	11.7684

4.2.3 Robustness of Solution

Following the similar style from classification case (section 4.1.4), let's try demonstrating the fact that our solution is robust even for our regression problem.

Case 1: Train-Test split ratio for original data = 7:3; seed = 9

```
>>> X_TR, X_TE, y_TR, y_TE = train_test_split(X, Y, test_size=0.3, stratify = Y,
>>>                                           random_state=9)
```

Consider this as the original scenario. The data is fit to the following stacking pipeline, same as before.

```
>>> stacked_regressor = StackGen([Ridge(), Lasso()],stacker =
>>> RandomForestRegressor(random_state = 9), classification = False, n_folds = 3,
>>> kf_random_state = 9, stack_with_orig = True, save_results = 0)

>>> stacked_regressor.fit_predict(X_TR, y_TR, X_TE, y_TE)
```

For Ridge regression, the mean squared error is

```
OOS error --->  19.8432255394
```

For Lasso, the mean squared error is

```
OOS error --->  26.9220710546
```

For the stacker – Random Forest Regressor (stacked with original features included), the mean squared error is

```
OOS error --->  11.7684244883
```

Case 2: Train-Test split ratio for original data = 7:3; seed = 42

Let's do this exercise one more time – change the seed and run the stacking model again.

For Ridge regression, the mean squared error is

```
OOS error --->  21.8684472122
```

For Lasso, the mean squared error is

```
OOS error --->  25.3655133042
```

For the stacker – Random Forest Regressor (stacked with original features included), the mean squared error is

```
OOS error --->  11.134684576
```

Like the original case, the OOS error of the stacker is still lower than the OOS errors of the individual base models.

Case 3: Train-Test split ratio for original data = 1:1; seed = 504

This time, change the train-test split and the seed.

For Ridge regression, the mean squared error is

```
OOS error --->  23.8415043426
```

For Lasso, the mean squared error is

```
OOS error --->  27.7441108529
```

For the stacker – Random Forest Regressor (stacked with original features included), the mean squared error is

```
OOS error --->  12.2047551603
```

Even after changing the train-test split ratio, the OOS error of the stacker is lower than the OOS errors of the individual base models.

Case 4: Train-Test split ratio for original data = 3:7; seed = 333

For the final case, change the train-test split and the seed again.

For Ridge regression, the mean squared error is

```
OOS error --->  29.7756818224
```

For Lasso, the mean squared error is

```
OOS error --->  33.7372347634
```

For the stacker – Random Forest Regressor (stacked with original features included), the mean squared error is

```
OOS error --->  16.6694818466
```

The stacking model continues to outperform the base models. This proves that our solution is robust to changes in input.

5. Conclusion

Through the examples above, we have shown that predictions from relatively simple (and diverse!) base models can be combined using the technique of stacked generalization to significantly reduce the generalization error. The StackGen package achieves this result while hiding the tedious complexity behind the process.

In retrospect, the most challenging aspect of the project was using object oriented concepts to build the package. The reason I chose this approach was maintainability and code reuse. Having well defined class/methods for the package would also make integrating further improvements easier while avoiding duplicate code. Another challenging part was passing the outputs (blended train and OOS datasets) from the base models to the stacker model for both regression and classification cases properly. This was overcome via the member functions fit_base_models() and fit_stacker() which handled the data exchange along with the fit_predict() function.

I first wrote the package using just functions. Soon this turned out to tedious as it did not provide the flexibility that classes and objects provide. Once the core functions were defined, I then created the StackGen class and added these functions as member functions. I also followed the bottom up approach; I built a working package for really simple cases with very few parameters,

tested the package and went on to add more parameters and functionality. This helped in testing the package thoroughly (even for corner conditions) and ensured that the member functions meshed together well for the final output.

To recap the process of stacked generalization, I have created the following flowchart to illustrate the core idea. It shows the flow of data between the models and how the final predictions are obtained.

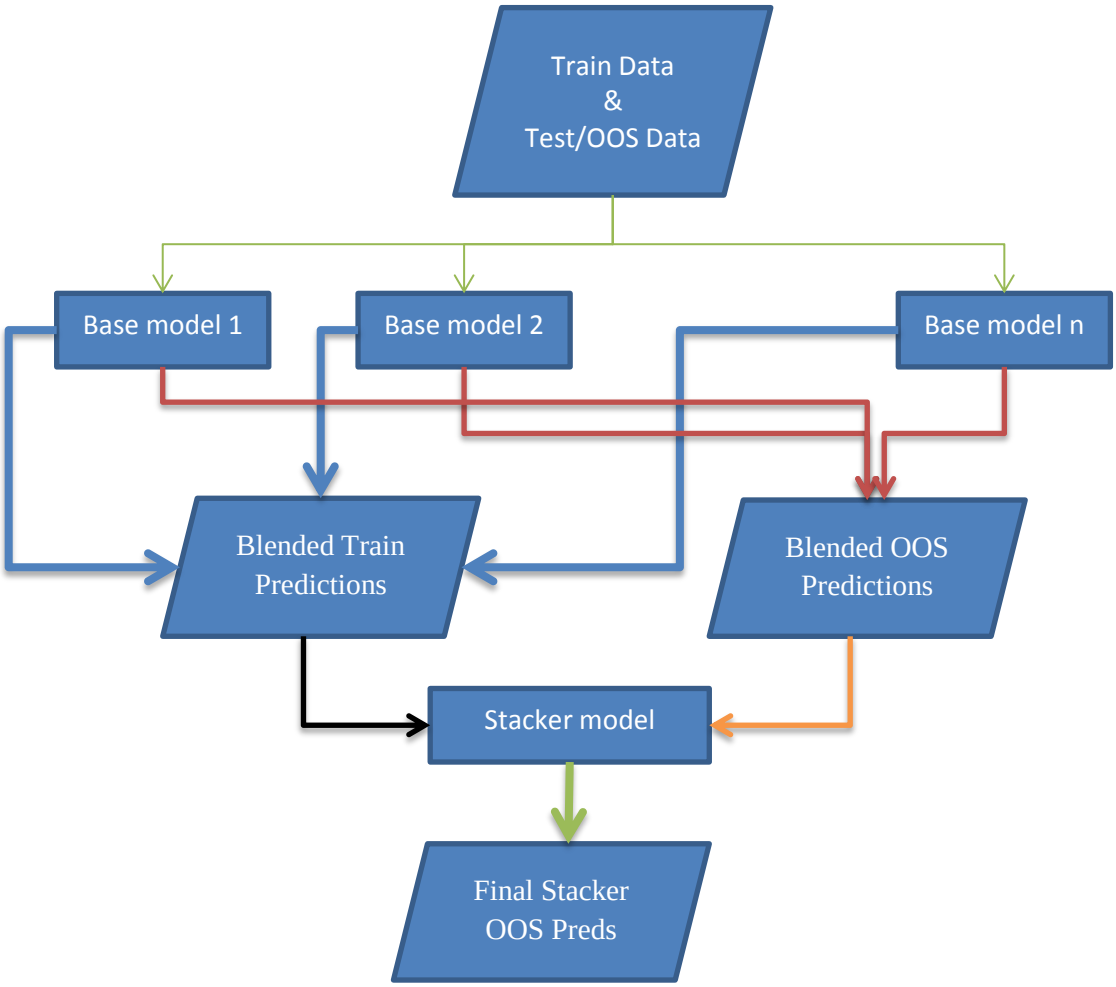

Figure 4 – Overview of the stacking process and dataflow between base models and stacker

The package definitely has room for future work/improvements –

1. The package can include tools that perform data preprocessing.
2. It can include tools that perform automatic stacking where high performing models are retained and rest is discarded.
3. It can include support for different error metrics.
4. It can provide tools for multiple levels of stacking as opposed to just one level.

5. Support for datasets that do not fit in RAM.
6. Support for distributed processing