# Automate Stacking In Python

## How to Boost Your Performance While Saving Time

Lukas Frei Follow
Jan 10 · 5 min read

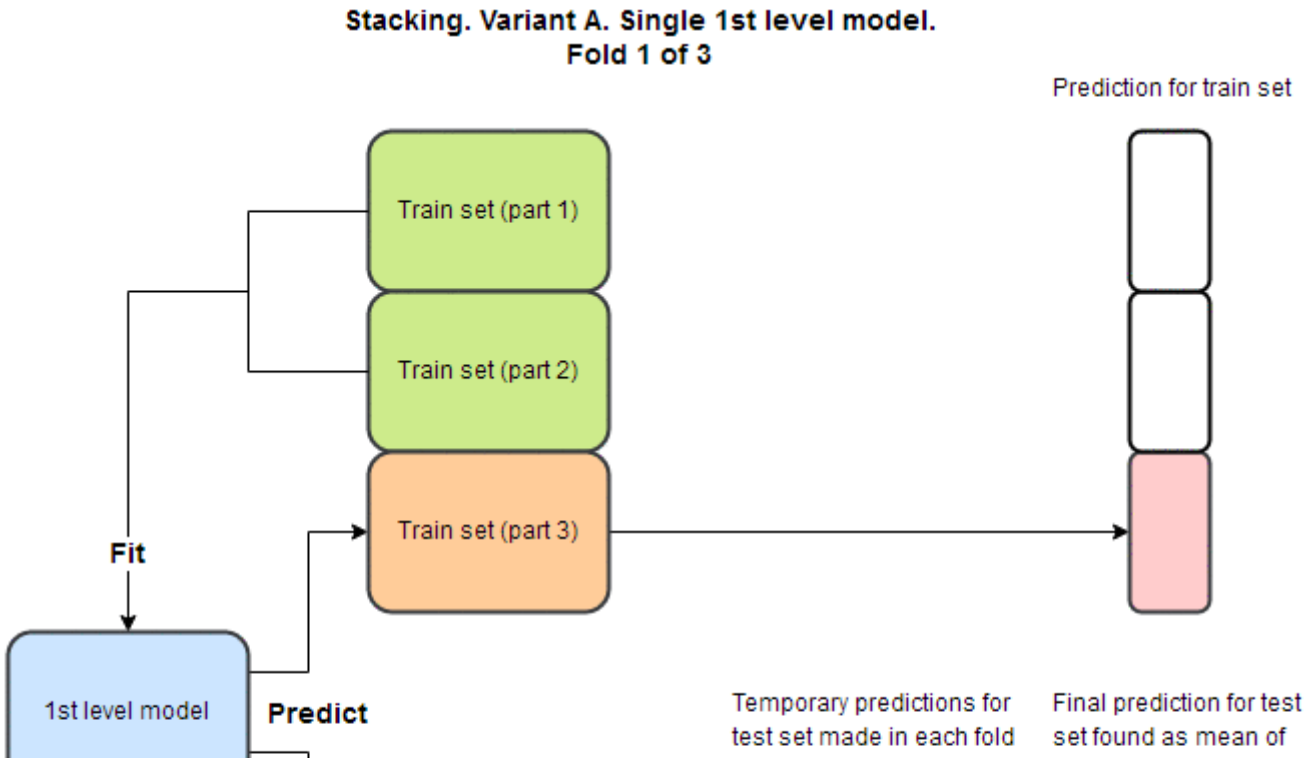
## Introduction

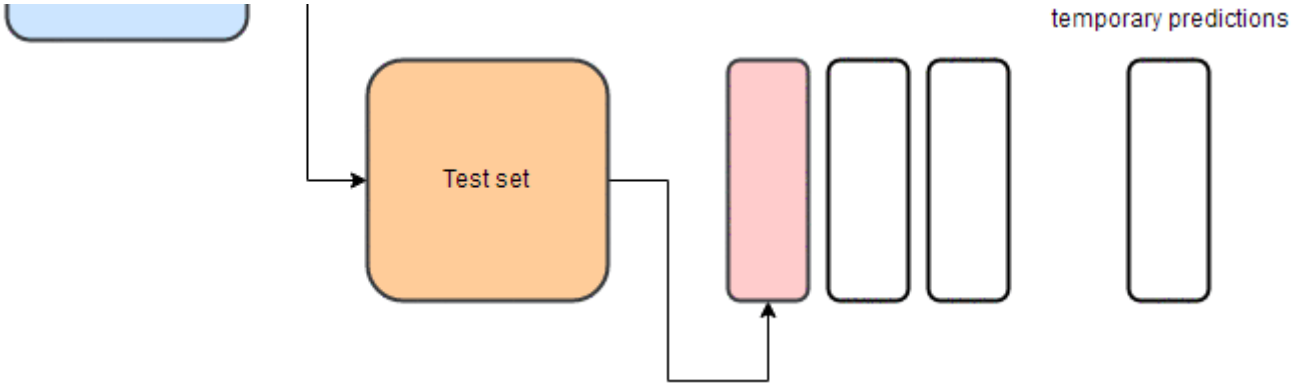
Utilizing stacking (stacked generalizations) is a very hot topic when it comes to pushing your machine learning algorithm to new heights. For instance, most if not all winning Kaggle submissions nowadays make use of some form of stacking or a variation of it. First introduced in the 1992 paper *Stacked Generalization* by David Wolpert, their main purpose is to reduce the generalization error. According to Wolpert, they can be understood "as a more sophisticated version of cross-validation". While Wolpert himself noted at the time that large parts of stacked generalizations are "black art", it seems that building larger and larger stacked generalizations win over smaller stacked generalizations. However, as these models keep increasing in size, they also increase in complexity. Automating the process of building different architectures would significantly simplify this process. The remainder of this article will deal with the package ***vecstack*** I recently came across that is attempting just this.


Source: https://giphy.com/gifs/funny-food-hRsayJrDAx8WY

## What Does a Stacked Generalization Look like?

The main idea behind the structure of a stacked generalization is to use one or more first level models, make predictions using these models and then use these predictions as features to fit one or more second level models on top. To avoid overfitting, cross-validation is usually used to predict the OOF (out-of-fold) part of the training set. There are two different variants available in this package but I'm going to describe 'Variant A' in this paragraph. To get the final predictions in this variant, we take the mean or mode of all of our predictions. The whole process can be visualized using this GIF from vecstacks' documentation:






## Use Case: Building a Stacked Generalization for Classification

After having taken a look at the documentation, it was time to try using the package myself and see how it works. To do so, I decided to use the wine data set available on the UCI Machine Learning Repository. The problem statement for this data set is to use the 13 features, which all represent different aspects of the wine, to predict from which of three cultivars in Italy the wine was derived.

To get started, let's first import the packages we are going to need for our project:

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from vecstack import stacking
```

Now we are ready to import our data and take a look at it to gain a better understanding of what it looks like:

```
link = 'https://archive.ics.uci.edu/ml/machine-learning-
databases/wine/wine.data'
names = ['Class', 'Alcohol', 'Malic acid', 'Ash',
         'Alcalinity of ash' ,'Magnesium', 'Total phenols',
         'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
'Color intensity', 'Hue', 'OD280/OD315 of diluted wines',
         'Proline']
```

```
df = pd.read_csv(link, header=None, names=names)
df.sample(5)
```

Running the code chunk above gives us:



Note that I used **.sample()** instead if **.head()** to avoid being potentially misled by assuming the entire data set has the structure of the first five rows. Luckily, this data set does not have any missing values, so we can easily use it to test our package right away without any of the usually required data cleaning and preparation.

Following this, we are going to separate the response from the input variables and perform an 80:20 train-test-split following the example on vecstacks' documentation.

```
y = df[['Class']]
X = df.iloc[:,1:]

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=0)
```

We are getting closer to the interesting part. Remember the GIF from earlier? It is time now to define a few first level models for our stacked generalization. This step definitely deserves its own article but for purposes of simplicity, we are going to use three models: A KNN-Classifier, a Random Forest Classifier and an XGBoost Classifier.

```
models = [
    KNeighborsClassifier(n_neighbors=5,
                         n_jobs=-1),

    RandomForestClassifier(random_state=0, n_jobs=-1,
                           n_estimators=100, max_depth=3),

    XGBClassifier(random_state=0, n_jobs=-1, learning_rate=0.1,
```

```
                    n_estimators=100, max_depth=3)
]
```

These parameters were not tuned prior to setting them as the purpose of this article is testing the package. If you were to optimize performance, you should not just copy and paste these.

Taking the next part of code from the documentation, we are essentially performing the GIF's first part using first level models to make predictions:

```
S_train, S_test = stacking(models,
                           X_train, y_train, X_test,
                           regression=False,

                           mode='oof_pred_bag',

                           needs_proba=False,

                           save_dir=None,

                           metric=accuracy_score,

                           n_folds=4,

                           stratified=True,

                           shuffle=True,

                           random_state=0,

                           verbose=2)
```

The stacking function takes several inputs:

- ***models***: the first level models we defined earlier
- ***X_train, y_train, X_test***: our data
- ***regression***: Boolean indicating whether we want to use the function for regression. In our case set to False since this is a classification
- ***mode:*** using the earlier describe out-of-fold during cross-validation

- ***needs_proba***: Boolean indicating whether you need the probabilities of class labels
- ***save_dir***: save the result to directory Boolean
- ***metric***: what evaluation metric to use (we imported the accuracy_score in the beginning)
- ***n_folds***: how many folds to use for cross-validation
- ***stratified***: whether to use stratified cross-validation
- ***shuffle***: whether to shuffle the data
- ***random_state***: setting a random state for reproducibility
- ***verbose***: 2 here refers to printing all info

Doing so, we get the following output:

```
task:         [classification]
n_classes:    [3]
metric:       [accuracy_score]
mode:         [oof_pred_bag]
n_models:     [4]

model  0:     [KNeighborsClassifier]
    fold  0:  [0.72972973]
    fold  1:  [0.61111111]
    fold  2:  [0.62857143]
    fold  3:  [0.76470588]
    ----
    MEAN:     [0.68352954] + [0.06517070]
    FULL:     [0.68309859]

model  1:     [ExtraTreesClassifier]
    fold  0:  [0.97297297]
    fold  1:  [1.00000000]
    fold  2:  [0.94285714]
    fold  3:  [1.00000000]
    ----
    MEAN:     [0.97895753] + [0.02358296]
    FULL:     [0.97887324]

model  2:     [RandomForestClassifier]
    fold  0:  [1.00000000]
    fold  1:  [1.00000000]
    fold  2:  [0.94285714]
```

```
    fold  3:  [1.00000000]
    ----
    MEAN:     [0.98571429] + [0.02474358]
    FULL:     [0.98591549]

model  3:     [XGBClassifier]
    fold  0:  [1.00000000]
    fold  1:  [0.97222222]
    fold  2:  [0.91428571]
    fold  3:  [0.97058824]
    ----
    MEAN:     [0.96427404] + [0.03113768]
    FULL:     [0.96478873]
```

Again, referring to the GIF, all that's left to do now is fit the second level model(s) of our choice on our predictions to make our final predictions. In our case, we are going to use an XGBoost Classifier. This step is not significantly different from a regular fit-and-predict in sklearn except for the fact that instead of using X_train to train our model, we are using our predictions S_train.

```
model = XGBClassifier(random_state=0, n_jobs=-1, learning_rate=0.1,
                      n_estimators=100, max_depth=3)

model = model.fit(S_train, y_train)

y_pred = model.predict(S_test)

print('Final prediction score: [%.8f]' % accuracy_score(y_test,
y_pred))

Output: Final prediction score: [0.97222222]
```

## Conclusion

Using vecstacks' stacking automation, we've managed to predict the correct wine cultivar with an accuracy of approximately 97.2%! As you can see, the API does not collide with the sklearn API and could, therefore, provide a helpful tool when trying to speed up your stacking workflow.

As always, if you have any feedback or found mistakes, please don't hesitate to reach out to me.

. . .

*References*:

[1] David H. Wolpert, Stacked Generalization (1992), Neural Networks

[2] Igor Ivanov, Vecstack (2016), GitHub

[3] M. Forina et al, Wine Data Set (1991), UCI Machine Learning Repository