Tabular Prediction using Auto Machine Learning (AutoGluon) | by Kb Pachauri | Towards Data Science

Tabular Prediction using Auto Machine Learning (AutoGluon)

In this post, I am sharing my exploration with the Tabular Prediction (predicting target column of the tabular dataset using the remaining column) using **Auto M**achine **L**earning (**AutoML**), **AutoGluon** from AWS labs, and details around its internal working.

AutoML frameworks provide enticing options as they remove the barriers for novice to train high-quality models and for an expert they reduce the time to the first workable model, which expert can further fine-tune by feature engineering based on data analysis and domain knowledge. Some of the popular open-source AutoML platforms are

TPOT: https://github.com/EpistasisLab/tpot

H20: https://www.h2o.ai/

AutoWEKA: http://www.cs.ubc.ca/labs/beta/Projects/autoweka/

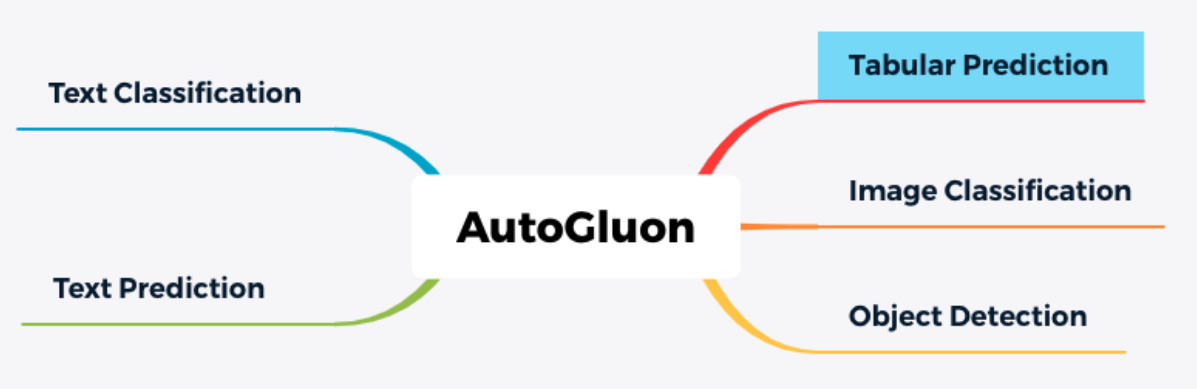
auto-sklearn: https://github.com/automl/auto-sklearn

AutoGluon: https://github.com/awslabs/autogluon

AutoGluon

AutoGluon is an AutoML toolkit for deep learning which automates end

to end machine learning tasks and enables us to achieve strong predictive performance with few lines of code. AutoGluon currently supports five major tasks as shown in the below image and also has support for custom mxnet or PyTorch models. In this blog, we will deep dive into the **Tabular prediction** task using AutoGluon and its internal working.



Predefined Tasks in AutoGluon

The major factors which enable high-quality modeling using AutoGluon for the tabular prediction task are

Novel Tabular Neural Network.

Auto Stacking and Ensembling Multiple Models.

Robust Data Preprocessing.

AutoGluon Tabular supports binary classification, multi-class classification, and regression problem type, and the actual problem type is auto-inferred based on the values of the target column if the problem type is not provided. AutoGluon Tabular currently train all the below models

k-Nearest Neighbors

Random Forests

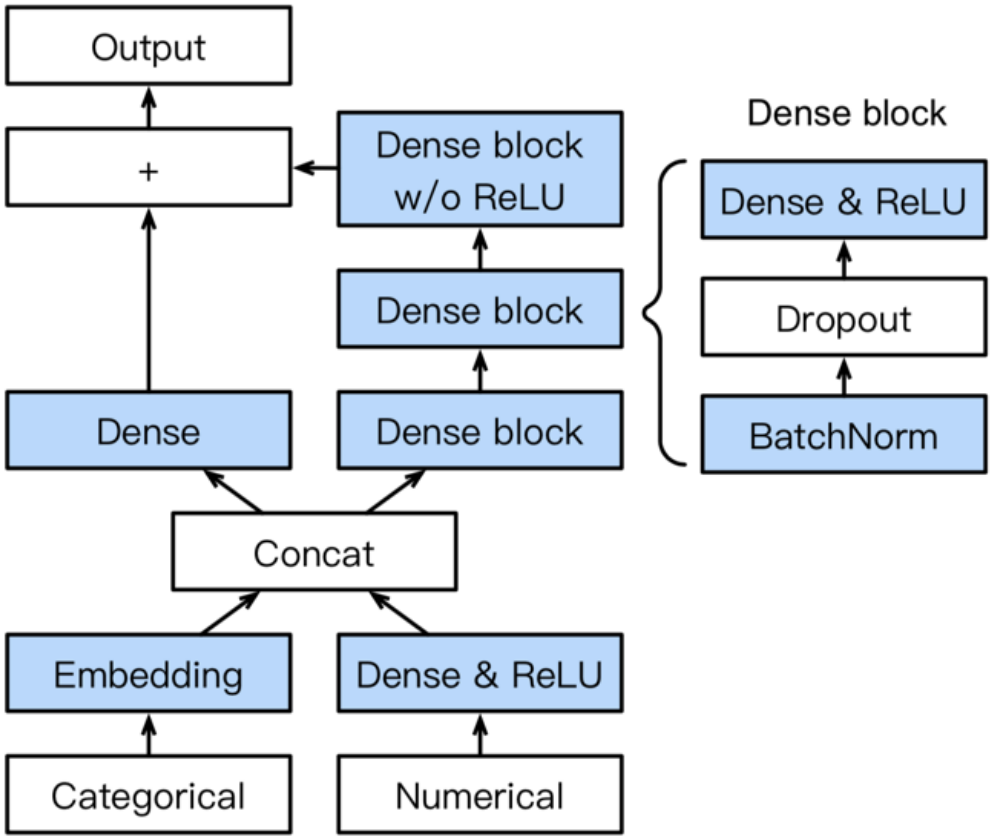
Extra Randomized Trees

Gradient Boosting Trees (LightGBM, CatBoost)

Tabular Deep Neural Network

Tabular Deep Neural Network

AutoGluon Tabular deep neural network is a carefully architected neural network which consists of a feedforward network with a linear shortcut path along with per variable embedding for tabular data that lacks translation invariance and locality of images or text which can be exploited by convolutions. The architecture and model details of the Tabular deep neural network is shown below.



AutoGluon Neural Network Architecture. Image Credit: https://arxiv.org/pdf/2003.06505.pdf

```
EmbedNet(
  (numeric_block): NumericBlock(
    (body): Dense(None -> 515, Activation(relu))
  )
  (output_block): WideAndDeepBlock(
    (deep): FeedforwardBlock(
      (body): HybridSequential(
        (0): BatchNorm(axis=1, eps=1e-05, momentum=0.9, fix_gamma=False, use_global_stats=False, in_channels=None)
        (1): Dropout(p = 0.1, axes=())
        (2): Dense(None -> 256, Activation(relu))
        (3): BatchNorm(axis=1, eps=1e-05, momentum=0.9, fix_gamma=False, use_global_stats=False, in_channels=None)
        (4): Dropout(p = 0.1, axes=())
        (5): Dense(None -> 128, Activation(relu))
        (6): BatchNorm(axis=1, eps=1e-05, momentum=0.9, fix_gamma=False, use_global_stats=False, in_channels=None)
        (7): Dropout(p = 0.1, axes=())
        (8): Dense(None -> 9, linear)
      )
    )
    (wide): Dense(None -> 9, linear)
  )
)
```

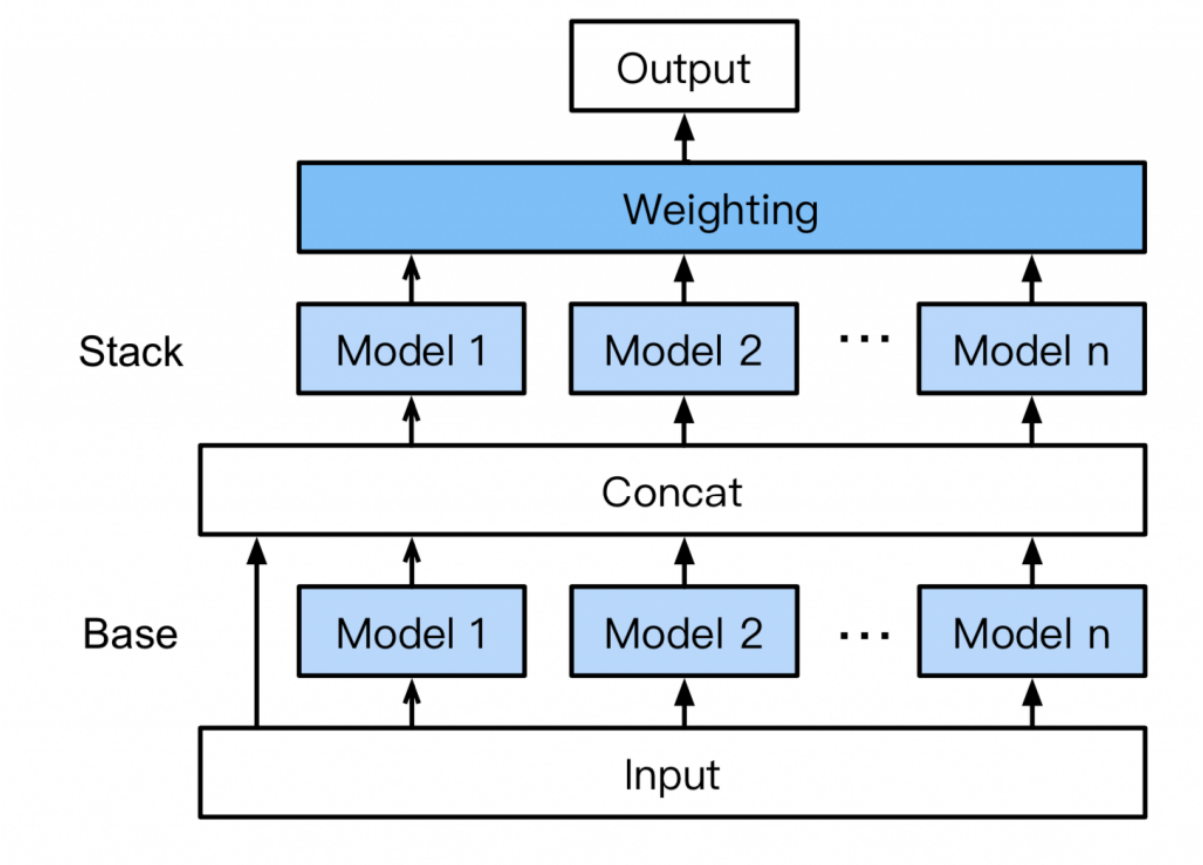
AutoGluon Neural Network Model Details

Tabular Neural Network uses a Numeric block which consists of a densely connected neural network layer with Relu activation for the

real-valued features. The **wide and deep block** of the tabular network consists of a hybrid sequential block that has two densely connected neural network layers of size [**256, 128**] which are scaled according to the number of classes for multiclass classification. The wide part of the network helps in **memorization** and the deep part of the network helps in **generalization**.

Auto Stacking

AutoGluon also uses multi-layer stack ensembling along with k-fold bagging to drastically reduce the variance of the final prediction as shown in the below image. This multi-layer stacking as shown in the below image has multiple models at the base layer, whose outputs are concatenated and fed to the next layer which also consists of multiple stacker models. These stacker models are base models with the same hyperparameters as the base layer and fed with concatenated output of the base layer models along with input.



Multi-layer stacking, ImageCredit: https://arxiv.org/pdf/2003.06505.pdf

If auto stacking is True, the number of stacking level is calculated heuristically using the equation ***min(1, max(0, math.floor (number train rows /750))***. and number of bagging fold is calculated heuristically using the equation **min(10, max(5, math.floor(num_train_rows / 100)))**

Data Pre-Processing

AutoGluon uses median imputation for missing numeric values, quantile normalization for skewed distribution, and standard normalization was applied to all other variables.

For categorical features, AutoGluon uses the embedding layer if discrete

levels are greater than 4, else uses one-hot encoding.

Fit()

AutoGluon **fit()** function call does all the magic underneath to learn a high-quality prediction model and below pseudocode summarize the overall training strategy. Each model after training is immediately saved to disk for fault tolerance and models are trained sequentially in order of low to high complexity to maintain the time envelope.

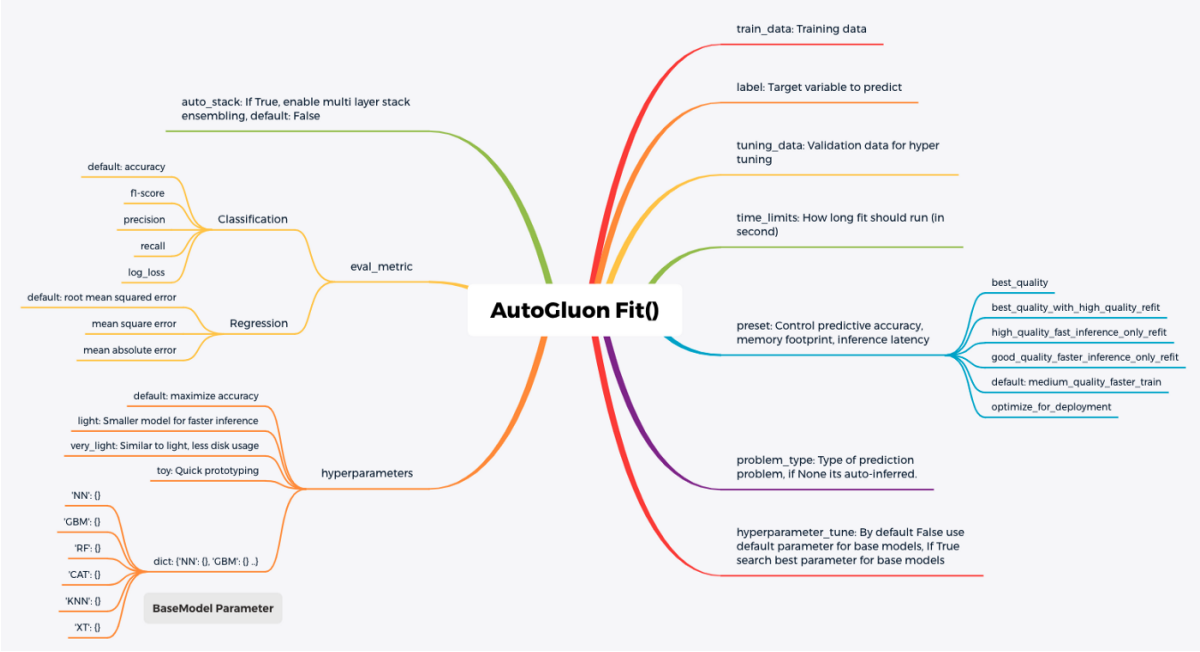
Algorithm 1 AutoGluon-Tabular Training Strategy (multi-layer stack ensembling + n-repeated k-fold bagging).

Require: data (X, Y), family of models $\mathcal{M}$, # of layers L

Preprocess data to extract features
for $l = 1$ **to** L **do** {Stacking}
  for $i = 1$ **to** n **do** {n-repeated}
    Randomly split data into k chunks $\{X^j, Y^j\}_{j=1}^k$
    for $j = 1$ **to** k **do** {k-fold bagging}
      for each model type m in $\mathcal{M}$ **do**
        Train a type-m model on X^{-j}, Y^{-j}
        Make predictions $\hat{Y}_{m,i}^j$ on OOF data X^j
      end for
    end for
  end for
  Average OOF predictions $\hat{Y}_m = \{\frac{1}{n}\sum_i \hat{Y}_{m,i}^j\}_{j=1}^k$
  $X \leftarrow \text{concatenate}(X, \{\hat{Y}_m\}_{m \in \mathcal{M}})$
end for

Training Strategy, Image Credit: https://arxiv.org/pdf/2003.06505.pdf

The fit() function provides a good degree of freedom through various input parameters and the below image shows all the major input parameters to fit with their default values which produce a reliable high-quality model and can be improved by further experiment with these parameters.



AutoGluon Fit Input Parameters

One of the important parameters, **presets** in **fit** function control predictive accuracy, inference latency, and resource usage. The below table shows a detailed comparison of predictive accuracy, training time, inference time, and kaggle rank in the [Otto Group Product Classification] challenge for different presets.
'high_quality_fast_inference_only_refit' provide the best tradeoff of predictive accuracy and inference latency.

Machine Details	64 Cores, Intel Xeon 2.3 GHz CPU, ~57.6 GB RAM				
Kaggle Competion	Otto Group Product Classification Challenge				
Preset	Train Time (hours)	Validation Loss	Inference Time (hours)	Test Loss	Kaggle Rank
best_quality	1.93	0.4161	4.10	0.4073	34 (under top 1%)
best_quality_with_high_quality_refit	2.73	0.4167	2.07	0.4074	34 (under top 1%)
high_quality_fast_inference_only_refit	2.21	0.4164	0.38	0.4079	36 (under top 1%)
good_quality_faster_inference_only_refit	1.70	0.4129	0.01	0.4284	322 (under top 10%)
medium_quality_faster_train	0.14	0.4383	0.07	0.4358	424 (under top 13%)
optimize_for_deployment	0.16	0.4129	0.13	0.4357	322 (under top 10%)

Otto Group Product Classification AutoGluon Presets Performance

Another important parameter is **time_limits** which control how long the **fit** function should run if a training model for an online environment where a real-time update to model within a specified time is important. The time limit is distributed for different hyperparameter optimization based on stack levels, the number of bagging folds, and the number of base models.

time_limits_hpo = time_limits / (1 + num_bagging_folds * (1 + stack_ensemble_levels))

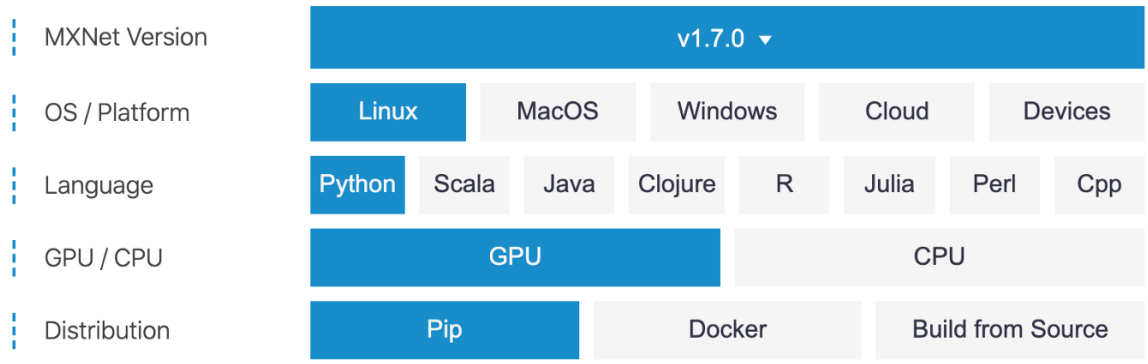
Installation

Instruction to install pip3

```
sudo apt install python3-pip
sudo -H pip3 install -U pip (update to latest version)
```

Instruction to install MXNET

Select a suitable version based on OS/platform from https://mxnet.apache.org/versions/1.7/get_started?



MXNET Configuration, Image from https://mxnet.apache.org/versions/1.7/get_started?platform=linux&language=python&processor=gpu&environ=pip&

```
sudo -H pip3 install mxnet-cu102 (GPU Version with CUDA 10.02)
sudo -H pip3 install mxnet (CPU Version)
```

Instruction to install AutoGluon

```
sudo -H pip3 install autogluon
```

Titanic Survival Prediction as Tabular Prediction

Download the Titanic dataset

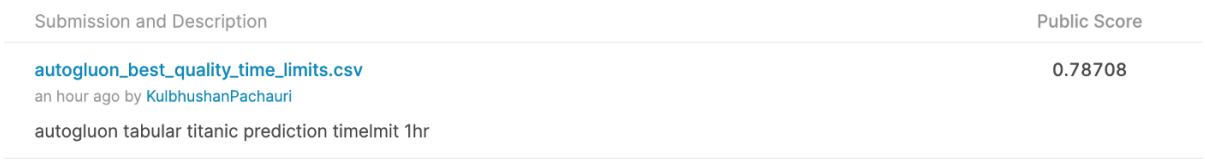
```
sudo -H pip3 install kaggle (install kaggle python package)
#download the key and save it home directorty (/home/userid/.kaggle)
kaggle competitions download -c titanic
```

Train the model using AutoGluon

```
from autogluon import TabularPrediction as task

label_column = 'Survived'

#load data (assuming train.csv and test.csv is alredy
downloaded)
train_data =
task.Dataset(file_path='train.csv').drop(labels=
['PassengerId'],axis=1)

test_data = task.Dataset(file_path='test.csv')

#create temp dataset to hold passenger id.
test_data_tmp = pd.DataFrame()
test_data_tmp['PassengerId'] =
test_data['PassengerId'].copy()
test_data = test_data.drop(labels=
['PassengerId'],axis=1)

#To get the best predictive accuracy, recommended
setting is to make auto_stack=True and time how long we
can wait (time_limits).

metric='accuracy'
predictor = task.fit(train_data=train_data,
label=label_column, eval_metric=metric, auto_stack=True,
time_limits=3600)

#calcutate test prediction
y_pred = predictor.predict(test_data, as_pandas=True)
y_pred = y_pred.to_frame()

#save the prediction to the file
y_pred['PassengerId'] =
test_data_tmp['PassengerId'].copy()
tfilename = 'autogluon_titantic.csv'
y_pred.to_csv(tfilename, index=False)
```

```
#submit test prediction to kaggle for scoring
kaggle competitions submit titanic -f tfilename -m
"autogluon tabular titanic prediction"
```

After the submission, we checked the score on the kaggle competition Titanic, under My Submission page, we got a score of **0.78708**, and which ranks under the top **15%** which is good, and after applying a feature engineering, we can further improve the predictive power of these models.



Conclusion

AutoGluon achieves competitive accuracy for Tabular Prediction tasks in few lines of code and is good to have a toolkit for both novice and expert. Flexibility, efficiency, and ease of use for prediction tasks in AutoGluon are achieved by novel architecture and time constraint fault-tolerant training strategy.

Thanks for reading the article, I hope you found this to be helpful. If you did, please share it on your favorite social media so other folks can find it, too. Also, let us know in the comment section if something is not clear or incorrect.

References

https://mxnet.apache.org/versions/1.7/

https://github.com/apache/incubator-mxnet

https://arxiv.org/pdf/2003.06505.pdf

https://aws.amazon.com/blogs/opensource/machine-learning-with-autogluon-an-open-source-automl-library/

https://towardsdatascience.com/autogluon-deep-learning-automl-5cdb4e2388ec