Predicting Columns in a Table - Quick Start — AutoGluon Documentation 0.5.1 documentation

Predicting Columns in a Table - Quick Start

Via a simple `fit()` call, AutoGluon can produce highly-accurate models to predict the values in one column of a data table based on the rest of the columns' values. Use AutoGluon with tabular data for both classification and regression problems. This tutorial demonstrates how to use AutoGluon to produce a classification model that predicts whether or not a person's income exceeds $50,000.

To start, import AutoGluon's TabularPredictor and TabularDataset classes:

```
from autogluon.tabular import TabularDataset, TabularPredictor
```

Load training data from a CSV file into an AutoGluon Dataset object. This object is essentially equivalent to a Pandas DataFrame and the same methods can be applied to both.

```
train_data =
TabularDataset('https://autogluon.s3.amazonaws.com/datasets/Inc/trai
n.csv')
subsample_size = 500 # subsample subset of data for faster demo, try
setting this to much larger values
train_data = train_data.sample(n=subsample_size, random_state=0)
train_data.head()
```

	age	workclass	fnlwgt	education	education-num	marital-status	occupation
6118	51	Private	39264	Some-college	10	Married-civ-spouse	Exec-managerial
23204	58	Private	51662	10th	6	Married-civ-spouse	Other-service
29590	40	Private	326310	Some-college	10	Married-civ-spouse	Craft-repair
18116	37	Private	222450	HS-grad	9	Never-married	Sales
33964	62	Private	109190	Bachelors	13	Married-civ-spouse	Exec-managerial

Note that we loaded data from a CSV file stored in the cloud (AWS s3 bucket), but you can you specify a local file-path instead if you have already downloaded the CSV file to your own machine (e.g., using wget). Each row in the table `train_data` corresponds to a single training example. In this particular dataset, each row corresponds to an individual person, and the columns contain various characteristics reported during a census.

Let's first use these features to predict whether the person's income exceeds $50,000 or not, which is recorded in the `class` column of this table.

```
label = 'class'
print("Summary of class variable: n", train_data[label].describe())
```

```
Summary of class variable:
count 500
```

```
unique 2
top <=50K
freq 365
Name: class, dtype: object
```

Now use AutoGluon to train multiple models:

```
save_path = 'agModels-predictClass' # specifies folder to store
trained models
predictor = TabularPredictor(label=label,
path=save_path).fit(train_data)
```

```
Beginning AutoGluon training ...
AutoGluon will save models to "agModels-predictClass/"
AutoGluon Version: 0.5.1b20220701
Python Version: 3.9.13
Operating System: Linux
Train Data Rows: 500
Train Data Columns: 14
Label Column: class
Preprocessing data ...
AutoGluon infers your prediction problem is: 'binary' (because only
two unique label-values observed).
2 unique label values: [' >50K', ' <=50K']
If 'binary' is not the correct problem_type, please manually specify
the problem_type parameter during predictor init (You may specify
problem_type as one of: ['binary', 'multiclass', 'regression'])
Selected class <--> label mapping: class 1 = >50K, class 0 = <=50K
Note: For your binary classification, AutoGluon arbitrarily selected
which label-value represents positive ( >50K) vs negative ( <=50K)
class.
To explicitly set the positive_class, either rename classes to 1 and
0, or specify positive_class in Predictor init.
Using Feature Generators to preprocess the data ...
Fitting AutoMLPipelineFeatureGenerator...
Available Memory: 22189.81 MB
Train Data (Original) Memory Usage: 0.29 MB (0.0% of available
memory)
Inferring data type of each feature based on column values. Set
feature_metadata_in to manually specify special dtypes of the
features.
Stage 1 Generators:
Fitting AsTypeFeatureGenerator...
Note: Converting 1 features to boolean dtype as they only contain 2
```

```
unique values.
Stage 2 Generators:
Fitting FillNaFeatureGenerator...
Stage 3 Generators:
Fitting IdentityFeatureGenerator...
Fitting CategoryFeatureGenerator...
Fitting CategoryMemoryMinimizeFeatureGenerator...
Stage 4 Generators:
Fitting DropUniqueFeatureGenerator...
Types of features in original data (raw dtype, special dtypes):
('int', []) : 6 | ['age', 'fnlwgt', 'education-num', 'capital-gain',
'capital-loss', ...]
('object', []) : 8 | ['workclass', 'education', 'marital-status',
'occupation', 'relationship', ...]
Types of features in processed data (raw dtype, special dtypes):
('category', []) : 7 | ['workclass', 'education', 'marital-status',
'occupation', 'relationship', ...]
('int', []) : 6 | ['age', 'fnlwgt', 'education-num', 'capital-gain',
'capital-loss', ...]
('int', ['bool']) : 1 | ['sex']
0.1s = Fit runtime
14 features in original data used to generate 14 features in
processed data.
Train Data (Processed) Memory Usage: 0.03 MB (0.0% of available
memory)
Data preprocessing and feature engineering runtime = 0.08s ...
AutoGluon will gauge predictive performance using evaluation metric:
'accuracy'
To change this, specify the eval_metric parameter of Predictor()
Automatically generating train/validation split with
holdout_frac=0.2, Train Rows: 400, Val Rows: 100
Fitting 13 L1 models ...
Fitting model: KNeighborsUnif ...
0.73 = Validation score (accuracy)
0.0s = Training runtime
0.01s = Validation runtime
Fitting model: KNeighborsDist ...
0.65 = Validation score (accuracy)
0.0s = Training runtime
0.01s = Validation runtime
Fitting model: LightGBMXT ...
0.83 = Validation score (accuracy)
0.6s = Training runtime
0.0s = Validation runtime
Fitting model: LightGBM ...
0.85 = Validation score (accuracy)
0.22s = Training runtime
0.0s = Validation runtime
```

```
Fitting model: RandomForestGini ...
0.84 = Validation score (accuracy)
0.46s = Training runtime
0.05s = Validation runtime
Fitting model: RandomForestEntr ...
0.83 = Validation score (accuracy)
0.45s = Training runtime
0.05s = Validation runtime
Fitting model: CatBoost ...
/var/lib/jenkins/miniconda3/envs/autogluon-tutorial-tabular-
v3/lib/python3.9/site-packages/xgboost/compat.py:31: FutureWarning:
pandas.Int64Index is deprecated and will be removed from pandas in a
future version. Use pandas.Index with the appropriate dtype instead.
from pandas import MultiIndex, Int64Index
0.85 = Validation score (accuracy)
0.75s = Training runtime
0.0s = Validation runtime
Fitting model: ExtraTreesGini ...
0.82 = Validation score (accuracy)
0.45s = Training runtime
0.05s = Validation runtime
Fitting model: ExtraTreesEntr ...
0.81 = Validation score (accuracy)
0.44s = Training runtime
0.05s = Validation runtime
Fitting model: NeuralNetFastAI ...
0.82 = Validation score (accuracy)
1.62s = Training runtime
0.01s = Validation runtime
Fitting model: XGBoost ...
0.87 = Validation score (accuracy)
0.22s = Training runtime
0.01s = Validation runtime
Fitting model: NeuralNetTorch ...
0.85 = Validation score (accuracy)
1.95s = Training runtime
0.01s = Validation runtime
Fitting model: LightGBMLarge ...
0.83 = Validation score (accuracy)
0.48s = Training runtime
0.0s = Validation runtime
Fitting model: WeightedEnsemble_L2 ...
0.87 = Validation score (accuracy)
0.28s = Training runtime
0.0s = Validation runtime
AutoGluon training complete, total runtime = 8.71s ... Best model:
"WeightedEnsemble_L2"
TabularPredictor saved. To load, use: predictor =
```

```
TabularPredictor.load("agModels-predictClass/")
```

Next, load separate test data to demonstrate how to make predictions on new examples at inference time:

```
test_data =
TabularDataset('https://autogluon.s3.amazonaws.com/datasets/Inc/test
.csv')
y_test = test_data[label] # values to predict
test_data_nolab = test_data.drop(columns=[label]) # delete label
column to prove we're not cheating
test_data_nolab.head()
```

```
Loaded data from:
https://autogluon.s3.amazonaws.com/datasets/Inc/test.csv | Columns =
15 / 15 | Rows = 9769 -> 9769
```

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	rela
0	31	Private	169085	11th	7	Married-civ-spouse	Sales	Wife
1	17	Self-emp-not-inc	226203	12th	8	Never-married	Sales	Own
2	47	Private	54260	Assoc-voc	11	Married-civ-spouse	Exec-managerial	Husl
3	21	Private	176262	Some-college	10	Never-married	Exec-managerial	Own
4	17	Private	241185	12th	8	Never-married	Prof-specialty	Own

We use our trained models to make predictions on the new data and then evaluate performance:

```
predictor = TabularPredictor.load(save_path) # unnecessary, just demonstrates how to load previously-trained predictor from file

y_pred = predictor.predict(test_data_nolab)
print("Predictions: n", y_pred)
perf = predictor.evaluate_predictions(y_true=y_test, y_pred=y_pred, auxiliary_metrics=True)
```

```
Evaluation: accuracy on test data: 0.8374449790152523
Evaluations on test data:
{
"accuracy": 0.8374449790152523,
"balanced_accuracy": 0.7430558394221018,
"mcc": 0.5243657567117436,
"f1": 0.621904761904762,
"precision": 0.69394261424017,
"recall": 0.5634167385677308
}
```

```
Predictions:
0 <=50K
1 <=50K
2 <=50K
3 <=50K
4 <=50K
...
9764 <=50K
9765 <=50K
9766 <=50K
9767 <=50K
9768 <=50K
Name: class, Length: 9769, dtype: object
```

We can also evaluate the performance of each individual trained model on our (labeled) test data:

```
predictor.leaderboard(test_data, silent=True)
```

	model	score_test	score_val	pred_time_test	pred_time_va
0	RandomForestGini	0.842973	0.84	0.126296	0.051974
1	CatBoost	0.842461	0.85	0.010687	0.004766
2	RandomForestEntr	0.841130	0.83	0.126020	0.052501
3	LightGBM	0.839799	0.85	0.014659	0.004906
4	XGBoost	0.837445	0.87	0.034302	0.006877
5	WeightedEnsemble_L2	0.837445	0.87	0.036686	0.007429
6	LightGBMXT	0.836421	0.83	0.008456	0.004678
7	ExtraTreesGini	0.833453	0.82	0.135617	0.053392
8	ExtraTreesEntr	0.832839	0.81	0.130296	0.052263
9	LightGBMLarge	0.828949	0.83	0.016930	0.004989
10	NeuralNetFastAI	0.818610	0.82	0.129103	0.012194
11	NeuralNetTorch	0.810523	0.85	0.137713	0.010975
12	KNeighborsUnif	0.725970	0.73	0.025161	0.006787
13	KNeighborsDist	0.695158	0.65	0.023775	0.005426

Now you're ready to try AutoGluon on your own tabular datasets! As long as they're stored in a popular format like CSV, you should be able to achieve strong predictive performance with just 2 lines of code:

```
from autogluon.tabular import TabularPredictor
predictor = TabularPredictor(label=<variable-name>).fit(train_data=
<file-name>)
```

Note: This simple call to `fit()` is intended for your first prototype model. In a subsequent section, we'll demonstrate how to maximize predictive performance by additionally specifying the `presets` parameter to `fit()` and the `eval_metric` parameter to `TabularPredictor()`.

Description of fit():

Here we discuss what happened during `fit()`.

Since there are only two possible values of the `class` variable, this was a binary classification problem, for which an appropriate performance metric is *accuracy*. AutoGluon automatically infers this as well as the type of each feature (i.e., which columns contain continuous numbers vs. discrete categories). AutoGluon can also automatically handle common issues like missing data and rescaling feature values.

We did not specify separate validation data and so AutoGluon automatically choses a random training/validation split of the data. The data used for validation is seperated from the training data and is used to determine the models and hyperparameter-values that produce the best results. Rather than just a single model, AutoGluon trains multiple models and ensembles them together to ensure superior predictive performance.

By default, AutoGluon tries to fit various types of models including neural networks and tree ensembles. Each type of model has various hyperparameters, which traditionally, the user would have to specify. AutoGluon automates this process.

AutoGluon automatically and iteratively tests values for hyperparameters to produce the best performance on the validation data. This involves repeatedly training models under different hyperparameter settings and evaluating their performance. This process can be computationally-intensive, so `fit()` can parallelize this process across multiple threads (and machines if distributed resources are available). To control runtimes, you can specify various arguments in `fit()` as demonstrated in the subsequent **In-Depth** tutorial.

For tabular problems, `fit()` returns a `Predictor` object. For classification, you can easily output predicted class probabilities instead of predicted classes:

```
pred_probs = predictor.predict_proba(test_data_nolab)
pred_probs.head(5)
```

	<=50K	>50K
0	0.982107	0.017893
1	0.988337	0.011663
2	0.573505	0.426495
3	0.998272	0.001728
4	0.990299	0.009701

Besides inference, this object can also summarize what happened during fit.

```
results = predictor.fit_summary(show_plot=True)
```

*** Summary of fit() ***
Estimated performance of each model:
model score_val pred_time_val fit_time pred_time_val_marginal fit_time_marginal stack_level can_infer fit_order
0 XGBoost 0.87 0.006877 0.216331 0.006877 0.216331 1 True 11
1 WeightedEnsemble_L2 0.87 0.007429 0.496433 0.000552 0.280102 2 True 14
2 CatBoost 0.85 0.004766 0.746675 0.004766 0.746675 1 True 7
3 LightGBM 0.85 0.004906 0.216884 0.004906 0.216884 1 True 4
4 NeuralNetTorch 0.85 0.010975 1.947703 0.010975 1.947703 1 True 12
5 RandomForestGini 0.84 0.051974 0.457605 0.051974 0.457605 1 True 5
6 LightGBMXT 0.83 0.004678 0.595197 0.004678 0.595197 1 True 3
7 LightGBMLarge 0.83 0.004989 0.478182 0.004989 0.478182 1 True 13
8 RandomForestEntr 0.83 0.052501 0.451235 0.052501 0.451235 1 True 6
9 NeuralNetFastAI 0.82 0.012194 1.617176 0.012194 1.617176 1 True 10
10 ExtraTreesGini 0.82 0.053392 0.450538 0.053392 0.450538 1 True 8
11 ExtraTreesEntr 0.81 0.052263 0.443687 0.052263 0.443687 1 True 9
12 KNeighborsUnif 0.73 0.006787 0.004329 0.006787 0.004329 1 True 1
13 KNeighborsDist 0.65 0.005426 0.003546 0.005426 0.003546 1 True 2
Number of models trained: 14
Types of models trained:
{'NNFastAiTabularModel', 'WeightedEnsembleModel', 'KNNModel', 'RFModel', 'XTModel', 'TabularNeuralNetTorchModel', 'CatBoostModel', 'LGBModel', 'XGBoostModel'}
Bagging used: False
Multi-layer stack-ensembling used: False
Feature Metadata (Processed):
(raw dtype, special dtypes):
('category', []) : 7 | ['workclass', 'education', 'marital-status', 'occupation', 'relationship', ...]
('int', []) : 6 | ['age', 'fnlwgt', 'education-num', 'capital-gain', 'capital-loss', ...]
('int', ['bool']) : 1 | ['sex']
*** End of fit() summary ***

```
/var/lib/jenkins/workspace/workspace/autogluon-tutorial-tabular-
v3/core/src/autogluon/core/utils/plots.py:138: UserWarning:
AutoGluon summary plots cannot be created because bokeh is not
installed. To see plots, please do: "pip install bokeh==2.0.1"
warnings.warn('AutoGluon summary plots cannot be created because
```

```
bokeh is not installed. To see plots, please do: "pip install
bokeh==2.0.1"')
```

From this summary, we can see that AutoGluon trained many different types of models as well as an ensemble of the best-performing models. The summary also describes the actual models that were trained during fit and how well each model performed on the held-out validation data. We can view what properties AutoGluon automatically inferred about our prediction task:

```
print("AutoGluon infers problem type is: ", predictor.problem_type)
print("AutoGluon identified the following types of features:")
print(predictor.feature_metadata)
```

```
AutoGluon infers problem type is: binary
AutoGluon identified the following types of features:
('category', []) : 7 | ['workclass', 'education', 'marital-status',
'occupation', 'relationship', ...]
('int', []) : 6 | ['age', 'fnlwgt', 'education-num', 'capital-gain',
'capital-loss', ...]
('int', ['bool']) : 1 | ['sex']
```

AutoGluon correctly recognized our prediction problem to be a **binary classification** task and decided that variables such as `age` should be represented as integers, whereas variables such as `workclass` should be represented as categorical objects. The `feature_metadata` attribute allows you to see the inferred data type of each predictive variable after preprocessing (this is its *raw* dtype; some features may also be associated with additional *special* dtypes if produced via feature-engineering, e.g. numerical representations of a datetime/text column).

We can evaluate the performance of each individual trained model on our (labeled) test data:

```
predictor.leaderboard(test_data, silent=True)
```

	model	score_test	score_val	pred_time_test	pred_time_va
0	RandomForestGini	0.842973	0.84	0.126064	0.051974
1	CatBoost	0.842461	0.85	0.010285	0.004766
2	RandomForestEntr	0.841130	0.83	0.127470	0.052501
3	LightGBM	0.839799	0.85	0.014990	0.004906
4	XGBoost	0.837445	0.87	0.031723	0.006877
5	WeightedEnsemble_L2	0.837445	0.87	0.033945	0.007429
6	LightGBMXT	0.836421	0.83	0.008862	0.004678
7	ExtraTreesGini	0.833453	0.82	0.135097	0.053392
8	ExtraTreesEntr	0.832839	0.81	0.128687	0.052263
9	LightGBMLarge	0.828949	0.83	0.017033	0.004989
10	NeuralNetFastAI	0.818610	0.82	0.130164	0.012194
11	NeuralNetTorch	0.810523	0.85	0.136908	0.010975
12	KNeighborsUnif	0.725970	0.73	0.027105	0.006787
13	KNeighborsDist	0.695158	0.65	0.025970	0.005426

When we call `predict()`, AutoGluon automatically predicts with the model that displayed the best performance on validation data (i.e. the weighted-ensemble). We can instead specify which model to use for predictions like this:

```
predictor.predict(test_data, model='LightGBM')
```

Above the scores of predictive performance were based on a default evaluation metric (accuracy for binary classification). Performance in certain applications may be measured by different metrics than the ones AutoGluon optimizes for by default. If you know the metric that counts in your application, you should specify it as demonstrated in the next section.

Presets

AutoGluon comes with a variety of presets that can be specified in the call to `.fit` via the `presets` argument. `medium_quality` is used by default to encourage initial prototyping, but for serious usage, the other presets should be used instead.

Preset	Model Quality	Use Cases	Fit Time (Ideal)	Inference Time (Relative to medium_quality)	Disk Usage
best_q uality	State-of-the - art (SOTA), much better than high_qualit y	When accuracy is what matters	16x+	32x+	16 x+
		When a very			

high_q uality	Better than good_qualit y	When a very powerful, portable solution with fast inference is required: Large-scale batch inference	16x	4x	2x
good_q uality	Significantl y better than medium_qual ity	When a powerful, highly portable solution with very fast inference is required: Billion-scale batch inference, sub-100ms online-inference, edge-devices	16x	2x	0. 1x
medium_qualit y	Competitive with other top AutoML Frameworks	Initial prototyping, establishing a performance baseline	1x	1x	1x

We recommend users to start with `medium_quality` to get a sense of the problem and identify any data related issues. If `medium_quality` is taking too long to train, consider subsampling the training data during this

prototyping phase.
Once you are comfortable, next try `best_quality`. Make sure to specify at least 16x the `time_limit` value as used in `medium_quality`. Once finished, you should have a very powerful solution that is often stronger than `medium_quality`.
Make sure to consider holding out test data that AutoGluon never sees during training to ensure that the models are performing as expected in terms of performance.
Once you evaluate both `best_quality` and `medium_quality`, check if either satisfies your needs. If neither do, consider trying `high_quality` and/or `good_quality`.
If none of the presets satisfy requirements, refer to Predicting Columns in a Table - In Depth for more advanced AutoGluon options.

Maximizing predictive performance

Note: You should not call `fit()` with entirely default arguments if you are benchmarking AutoGluon-Tabular or hoping to maximize its accuracy! To get the best predictive accuracy with AutoGluon, you should generally use it like this:

```
time_limit = 60 # for quick demonstration only, you should set this
to longest time you are willing to wait (in seconds)
metric = 'roc_auc' # specify your evaluation metric here
predictor = TabularPredictor(label,
eval_metric=metric).fit(train_data, time_limit=time_limit,
presets='best_quality')
predictor.leaderboard(test_data, silent=True)
```

```
No path specified. Models will be saved in: "AutogluonModels/ag-
20220701_215233/"
```

```
Presets specified: ['best_quality']
Stack configuration (auto_stack=True): num_stack_levels=0,
num_bag_folds=5, num_bag_sets=20
Beginning AutoGluon training ... Time limit = 60s
AutoGluon will save models to "AutogluonModels/ag-20220701_215233/"
AutoGluon Version: 0.5.1b20220701
Python Version: 3.9.13
Operating System: Linux
Train Data Rows: 500
Train Data Columns: 14
Label Column: class
Preprocessing data ...
AutoGluon infers your prediction problem is: 'binary' (because only
two unique label-values observed).
2 unique label values: [' >50K', ' <=50K']
If 'binary' is not the correct problem_type, please manually specify
the problem_type parameter during predictor init (You may specify
problem_type as one of: ['binary', 'multiclass', 'regression'])
Selected class <--> label mapping: class 1 = >50K, class 0 = <=50K
Note: For your binary classification, AutoGluon arbitrarily selected
which label-value represents positive ( >50K) vs negative ( <=50K)
class.
To explicitly set the positive_class, either rename classes to 1 and
0, or specify positive_class in Predictor init.
Using Feature Generators to preprocess the data ...
Fitting AutoMLPipelineFeatureGenerator...
Available Memory: 21960.27 MB
Train Data (Original) Memory Usage: 0.29 MB (0.0% of available
memory)
Inferring data type of each feature based on column values. Set
feature_metadata_in to manually specify special dtypes of the
features.
Stage 1 Generators:
Fitting AsTypeFeatureGenerator...
Note: Converting 1 features to boolean dtype as they only contain 2
unique values.
Stage 2 Generators:
Fitting FillNaFeatureGenerator...
Stage 3 Generators:
Fitting IdentityFeatureGenerator...
Fitting CategoryFeatureGenerator...
Fitting CategoryMemoryMinimizeFeatureGenerator...
Stage 4 Generators:
Fitting DropUniqueFeatureGenerator...
Types of features in original data (raw dtype, special dtypes):
('int', []) : 6 | ['age', 'fnlwgt', 'education-num', 'capital-gain',
'capital-loss', ...]
('object', []) : 8 | ['workclass', 'education', 'marital-status',
```

```
'occupation', 'relationship', ...]
Types of features in processed data (raw dtype, special dtypes):
('category', []) : 7 | ['workclass', 'education', 'marital-status',
'occupation', 'relationship', ...]
('int', []) : 6 | ['age', 'fnlwgt', 'education-num', 'capital-gain',
'capital-loss', ...]
('int', ['bool']) : 1 | ['sex']
0.1s = Fit runtime
14 features in original data used to generate 14 features in
processed data.
Train Data (Processed) Memory Usage: 0.03 MB (0.0% of available
memory)
Data preprocessing and feature engineering runtime = 0.08s ...
AutoGluon will gauge predictive performance using evaluation metric:
'roc_auc'
This metric expects predicted probabilities rather than predicted
class labels, so you'll need to use predict_proba() instead of
predict()
To change this, specify the eval_metric parameter of Predictor()
Fitting 13 L1 models ...
Fitting model: KNeighborsUnif_BAG_L1 ... Training model for up to
59.92s of the 59.92s of remaining time.
0.5196 = Validation score (roc_auc)
0.0s = Training runtime
0.01s = Validation runtime
Fitting model: KNeighborsDist_BAG_L1 ... Training model for up to
59.9s of the 59.9s of remaining time.
0.537 = Validation score (roc_auc)
0.0s = Training runtime
0.0s = Validation runtime
Fitting model: LightGBMXT_BAG_L1 ... Training model for up to 59.89s
of the 59.89s of remaining time.
Fitting 5 child models (S1F1 - S1F5) | Fitting with
ParallelLocalFoldFittingStrategy
0.8819 = Validation score (roc_auc)
1.19s = Training runtime
0.02s = Validation runtime
Fitting model: LightGBM_BAG_L1 ... Training model for up to 54.41s
of the 54.41s of remaining time.
Fitting 5 child models (S1F1 - S1F5) | Fitting with
ParallelLocalFoldFittingStrategy
0.867 = Validation score (roc_auc)
1.55s = Training runtime
0.02s = Validation runtime
Fitting model: RandomForestGini_BAG_L1 ... Training model for up to
51.64s of the 51.63s of remaining time.
0.8874 = Validation score (roc_auc)
0.54s = Training runtime
```

```
0.11s = Validation runtime
Fitting model: RandomForestEntr_BAG_L1 ... Training model for up to
50.96s of the 50.96s of remaining time.
0.889 = Validation score (roc_auc)
0.47s = Training runtime
0.11s = Validation runtime
Fitting model: CatBoost_BAG_L1 ... Training model for up to 50.36s
of the 50.36s of remaining time.
Fitting 5 child models (S1F1 - S1F5) | Fitting with
ParallelLocalFoldFittingStrategy
0.8923 = Validation score (roc_auc)
3.34s = Training runtime
0.03s = Validation runtime
Fitting model: ExtraTreesGini_BAG_L1 ... Training model for up to
45.86s of the 45.86s of remaining time.
0.8936 = Validation score (roc_auc)
0.55s = Training runtime
0.14s = Validation runtime
Fitting model: ExtraTreesEntr_BAG_L1 ... Training model for up to
45.14s of the 45.14s of remaining time.
0.8877 = Validation score (roc_auc)
0.45s = Training runtime
0.11s = Validation runtime
Fitting model: NeuralNetFastAI_BAG_L1 ... Training model for up to
44.56s of the 44.56s of remaining time.
Fitting 5 child models (S1F1 - S1F5) | Fitting with
ParallelLocalFoldFittingStrategy
0.8695 = Validation score (roc_auc)
2.56s = Training runtime
0.05s = Validation runtime
Fitting model: XGBoost_BAG_L1 ... Training model for up to 40.83s of
the 40.83s of remaining time.
Fitting 5 child models (S1F1 - S1F5) | Fitting with
ParallelLocalFoldFittingStrategy
0.868 = Validation score (roc_auc)
0.94s = Training runtime
0.03s = Validation runtime
Fitting model: NeuralNetTorch_BAG_L1 ... Training model for up to
38.51s of the 38.51s of remaining time.
Fitting 5 child models (S1F1 - S1F5) | Fitting with
ParallelLocalFoldFittingStrategy
0.8459 = Validation score (roc_auc)
4.35s = Training runtime
0.06s = Validation runtime
Fitting model: LightGBMLarge_BAG_L1 ... Training model for up to
32.87s of the 32.87s of remaining time.
Fitting 5 child models (S1F1 - S1F5) | Fitting with
ParallelLocalFoldFittingStrategy
```

```
0.8433 = Validation score (roc_auc)
1.73s = Training runtime
0.02s = Validation runtime
Completed 1/20 k-fold bagging repeats ...
Fitting model: WeightedEnsemble_L2 ... Training model for up to
59.92s of the 29.88s of remaining time.
0.9033 = Validation score (roc_auc)
1.45s = Training runtime
0.0s = Validation runtime
AutoGluon training complete, total runtime = 31.58s ... Best model:
"WeightedEnsemble_L2"
TabularPredictor saved. To load, use: predictor =
TabularPredictor.load("AutogluonModels/ag-20220701_215233/")
```

	model	score_test	score_val	pred_time_test	pred_tim
0	LightGBMXT_BAG_L1	0.900802	0.881867	0.097268	0.023926
1	CatBoost_BAG_L1	0.900744	0.892278	0.048818	0.028555
2	WeightedEnsemble_L2	0.897912	0.903298	1.769021	0.396751
3	LightGBM_BAG_L1	0.892347	0.866991	0.056381	0.021269
4	XGBoost_BAG_L1	0.891483	0.868006	0.138423	0.032109
5	RandomForestEntr_BAG_L1	0.886810	0.889011	0.126992	0.111098
6	NeuralNetFastAI_BAG_L1	0.885518	0.869508	0.643263	0.053776
7	RandomForestGini_BAG_L1	0.885092	0.887407	0.126224	0.110357
8	NeuralNetTorch_BAG_L1	0.882353	0.845906	0.814653	0.063004
9	ExtraTreesEntr_BAG_L1	0.880568	0.887681	0.133256	0.110385

10	ExtraTreesGini_BAG_L1	0.879806	0.893607	0.132516	0.139864
11	LightGBMLarge_BAG_L1	0.873437	0.843308	0.071421	0.021894
12	KNeighborsDist_BAG_L1	0.525998	0.536956	0.023028	0.004561
13	KNeighborsUnif_BAG_L1	0.514970	0.519604	0.023142	0.005219

This command implements the following strategy to maximize accuracy:

- Specify the argument `presets='best_quality'`, which allows AutoGluon to automatically construct powerful model ensembles based on stacking/bagging, and will greatly improve the resulting predictions if granted sufficient training time. The default value of `presets` is `'medium_quality'`, which produces *less* accurate models but facilitates faster prototyping. With `presets`, you can flexibly prioritize predictive accuracy vs. training/inference speed. For example, if you care less about predictive performance and want to quickly deploy a basic model, consider using: `presets=['good_quality', 'optimize_for_deployment']`.
- Provide the parameter `eval_metric` to `TabularPredictor()` if you know what metric will be used to evaluate predictions in your application. Some other non-default metrics you might use include things like: `'f1'` (for binary classification), `'roc_auc'` (for binary classification), `'log_loss'` (for classification), `'mean_absolute_error'` (for regression), `'median_absolute_error'` (for regression). You can also define your own custom metric function. For more information refer to Adding a custom metric to AutoGluon
- Include all your data in `train_data` and do not provide `tuning_data` (AutoGluon will split the data more intelligently to fit its needs).

- Do not specify the `hyperparameter_tune_kwargs` argument (counterintuitively, hyperparameter tuning is not the best way to spend a limited training time budgets, as model ensembling is often superior). We recommend you only use `hyperparameter_tune_kwargs` if your goal is to deploy a single model rather than an ensemble.
- Do not specify `hyperparameters` argument (allow AutoGluon to adaptively select which models/hyperparameters to use).
- Set `time_limit` to the longest amount of time (in seconds) that you are willing to wait. AutoGluon's predictive performance improves the longer `fit()` is allowed to run.

Regression (predicting numeric table columns):

To demonstrate that `fit()` can also automatically handle regression tasks, we now try to predict the numeric `age` variable in the same table based on the other features:

```
age_column = 'age'
print("Summary of age variable: n", train_data[age_column].describe())
```

```
Summary of age variable:
count 500.00000
mean 39.65200
std 13.52393
min 17.00000
25% 29.00000
50% 38.00000
75% 49.00000
max 85.00000
Name: age, dtype: float64
```

We again call `fit()`, imposing a time-limit this time (in seconds), and also demonstrate a shorthand method to evaluate the resulting model on the test data (which contain labels):

```
predictor_age = TabularPredictor(label=age_column, path="agModels-
predictAge").fit(train_data, time_limit=60)
performance = predictor_age.evaluate(test_data)
```

```
Beginning AutoGluon training ... Time limit = 60s
AutoGluon will save models to "agModels-predictAge/"
AutoGluon Version: 0.5.1b20220701
Python Version: 3.9.13
Operating System: Linux
Train Data Rows: 500
Train Data Columns: 14
Label Column: age
Preprocessing data ...
AutoGluon infers your prediction problem is: 'regression' (because
dtype of label-column == int and many unique label-values observed).
Label info (max, min, mean, stddev): (85, 17, 39.652, 13.52393)
If 'regression' is not the correct problem_type, please manually
specify the problem_type parameter during predictor init (You may
specify problem_type as one of: ['binary', 'multiclass',
'regression'])
Using Feature Generators to preprocess the data ...
Fitting AutoMLPipelineFeatureGenerator...
Available Memory: 21619.43 MB
Train Data (Original) Memory Usage: 0.32 MB (0.0% of available
memory)
Inferring data type of each feature based on column values. Set
feature_metadata_in to manually specify special dtypes of the
features.
Stage 1 Generators:
Fitting AsTypeFeatureGenerator...
Note: Converting 2 features to boolean dtype as they only contain 2
unique values.
Stage 2 Generators:
Fitting FillNaFeatureGenerator...
Stage 3 Generators:
Fitting IdentityFeatureGenerator...
Fitting CategoryFeatureGenerator...
Fitting CategoryMemoryMinimizeFeatureGenerator...
Stage 4 Generators:
```

```
Fitting DropUniqueFeatureGenerator...
Types of features in original data (raw dtype, special dtypes):
('int', []) : 5 | ['fnlwgt', 'education-num', 'capital-gain',
'capital-loss', 'hours-per-week']
('object', []) : 9 | ['workclass', 'education', 'marital-status',
'occupation', 'relationship', ...]
Types of features in processed data (raw dtype, special dtypes):
('category', []) : 7 | ['workclass', 'education', 'marital-status',
'occupation', 'relationship', ...]
('int', []) : 5 | ['fnlwgt', 'education-num', 'capital-gain',
'capital-loss', 'hours-per-week']
('int', ['bool']) : 2 | ['sex', 'class']
0.1s = Fit runtime
14 features in original data used to generate 14 features in
processed data.
Train Data (Processed) Memory Usage: 0.03 MB (0.0% of available
memory)
Data preprocessing and feature engineering runtime = 0.08s ...
AutoGluon will gauge predictive performance using evaluation metric:
'root_mean_squared_error'
This metric's sign has been flipped to adhere to being
higher_is_better. The metric score can be multiplied by -1 to get
the metric value.
To change this, specify the eval_metric parameter of Predictor()
Automatically generating train/validation split with
holdout_frac=0.2, Train Rows: 400, Val Rows: 100
Fitting 11 L1 models ...
Fitting model: KNeighborsUnif ... Training model for up to 59.92s of
the 59.92s of remaining time.
-15.6869 = Validation score (-root_mean_squared_error)
0.0s = Training runtime
0.01s = Validation runtime
Fitting model: KNeighborsDist ... Training model for up to 59.91s of
the 59.91s of remaining time.
-15.1801 = Validation score (-root_mean_squared_error)
0.0s = Training runtime
0.01s = Validation runtime
Fitting model: LightGBMXT ... Training model for up to 59.9s of the
59.9s of remaining time.
-11.7092 = Validation score (-root_mean_squared_error)
0.28s = Training runtime
0.0s = Validation runtime
Fitting model: LightGBM ... Training model for up to 59.61s of the
59.6s of remaining time.
-11.9295 = Validation score (-root_mean_squared_error)
0.24s = Training runtime
0.0s = Validation runtime
Fitting model: RandomForestMSE ... Training model for up to 59.36s
```

```
of the 59.36s of remaining time.
-11.6669 = Validation score (-root_mean_squared_error)
0.39s = Training runtime
0.04s = Validation runtime
Fitting model: CatBoost ... Training model for up to 58.91s of the
58.91s of remaining time.
-11.7993 = Validation score (-root_mean_squared_error)
0.64s = Training runtime
0.0s = Validation runtime
Fitting model: ExtraTreesMSE ... Training model for up to 58.26s of
the 58.26s of remaining time.
-11.3691 = Validation score (-root_mean_squared_error)
0.37s = Training runtime
0.04s = Validation runtime
Fitting model: NeuralNetFastAI ... Training model for up to 57.83s
of the 57.83s of remaining time.
-12.0733 = Validation score (-root_mean_squared_error)
0.54s = Training runtime
0.01s = Validation runtime
Fitting model: XGBoost ... Training model for up to 57.27s of the
57.27s of remaining time.
-12.2892 = Validation score (-root_mean_squared_error)
0.27s = Training runtime
0.01s = Validation runtime
Fitting model: NeuralNetTorch ... Training model for up to 56.99s of
the 56.99s of remaining time.
-11.9954 = Validation score (-root_mean_squared_error)
1.33s = Training runtime
0.01s = Validation runtime
Fitting model: LightGBMLarge ... Training model for up to 55.64s of
the 55.64s of remaining time.
-12.3153 = Validation score (-root_mean_squared_error)
0.47s = Training runtime
0.0s = Validation runtime
Fitting model: WeightedEnsemble_L2 ... Training model for up to
59.92s of the 54.96s of remaining time.
-11.2086 = Validation score (-root_mean_squared_error)
0.28s = Training runtime
0.0s = Validation runtime
AutoGluon training complete, total runtime = 5.33s ... Best model:
"WeightedEnsemble_L2"
TabularPredictor saved. To load, use: predictor =
TabularPredictor.load("agModels-predictAge/")
Evaluation: root_mean_squared_error on test data:
-10.496803156697219
Note: Scores are always higher_is_better. This metric score can be
multiplied by -1 to get the metric value.
Evaluations on test data:
```

```
{
"root_mean_squared_error": -10.496803156697219,
"mean_squared_error": -110.18287651044871,
"mean_absolute_error": -8.27001225212504,
"r2": 0.4110511363778958,
"pearsonr": 0.6426579499678333,
"median_absolute_error": -6.889945983886719
}
```

Note that we didn't need to tell AutoGluon this is a regression problem, it automatically inferred this from the data and reported the appropriate performance metric (RMSE by default). To specify a particular evaluation metric other than the default, set the `eval_metric` parameter of `TabularPredictor()` and AutoGluon will tailor its models to optimize your metric (e.g. `eval_metric = 'mean_absolute_error'`). For evaluation metrics where higher values are worse (like RMSE), AutoGluon will flip their sign and print them as negative values during training (as it internally assumes higher values are better).

We can call leaderboard to see the per-model performance:

```
predictor_age.leaderboard(test_data, silent=True)
```

	model	score_test	score_val	pred_time_test	pred_time_va
0	WeightedEnsemble_L2	-10.496803	-11.208568	0.328798	0.064829
1	ExtraTreesMSE	-10.656025	-11.369094	0.107588	0.042292
2	RandomForestMSE	-10.745634	-11.666909	0.100808	0.041925
3	CatBoost	-10.780312	-11.799279	0.011249	0.004880

4	LightGBMXT	-10.837373	-11.709228	0.048573	0.004746
5	LightGBM	-10.972156	-11.929546	0.016873	0.004181
6	XGBoost	-11.115033	-12.289224	0.030071	0.006422
7	NeuralNetFastAI	-11.225699	-12.073282	0.130742	0.011028
8	NeuralNetTorch	-11.448391	-11.995427	0.138675	0.010968
9	LightGBMLarge	-11.469922	-12.315314	0.025914	0.004298
10	KNeighborsUnif	-14.902058	-15.686937	0.023081	0.005714
11	KNeighborsDist	-15.771259	-15.180149	0.024007	0.005308

Data Formats: AutoGluon can currently operate on data tables already loaded into Python as pandas DataFrames, or those stored in files of CSV format or Parquet format. If your data live in multiple tables, you will first need to join them into a single table whose rows correspond to statistically independent observations (datapoints) and columns correspond to different features (aka. variables/covariates).

Refer to the TabularPredictor documentation to see all of the available methods/options.

Advanced Usage

For more advanced usage examples of AutoGluon, refer to Predicting Columns in a Table - In Depth