auto_ml Documentation

Release 0.1.0

Preston Parry

Nov 27, 2017

Contents

1 Getting Started **3**
- 1.1 Installation . . . 3
- 1.2 Core Functionality Example . . . 3
- 1.3 Advice . . . 4

2 Formatting Data **5**
- 2.1 Training data format . . . 5
- 2.2 Header row information . . . 5

3 Passing in your own feature engineering function **7**

4 Using machine learning for analytics **9**
- 4.1 The code to make this work . . . 9
- 4.2 Tangent time- what do you mean analytics from machine learning? . . . 10
- 4.3 Note- no free lunch . . . 10
- 4.4 Interpreting Results . . . 10
- 4.5 Interpreting Predicted Probability Buckets for Classifiers . . . 10
- 4.6 In the weeds . . . 11

5 Feature Responses **13**
- 5.1 Methodology . . . 13
- 5.2 Output . . . 13
- 5.3 CAVEATS!! . . . 13

6 Categorical Ensembling **15**

7 Deep Learning & Feature Learning **17**
- 7.1 Deep Learning in auto_ml . . . 17
- 7.2 Feature Learning . . . 17

8 Properly Formal API Documenation **19**
- 8.1 auto_ml . . . 19

9 Installation **23**

10 Core Functionality Example **25**

11 XGBoost, Deep Leaarning with TensorFlow & Keras, and LightGBM **27**

12 Classification 29

13 Advice 31

14 What this project does 33

Contents:

CHAPTER 1

Getting Started

1.1 Installation

```
pip install auto_ml
```

1.2 Core Functionality Example

```
from auto_ml import Predictor
from auto_ml.utils import get_boston_dataset

df_train, df_test = get_boston_dataset()

column_descriptions = {
    'MEDV': 'output'
    , 'CHAS': 'categorical'
}

ml_predictor = Predictor(type_of_estimator='regressor', column_descriptions=column_
↪descriptions)

ml_predictor.train(df_train)

ml_predictor.score(df_test, df_test.MEDV)
```

That's it.

Seriously.

Sure, there's a ton of complexity hiding under the surface. And yes, I've got a ton more options that you can pass in to customize your experience with `auto_ml`, but none of that's at all necessary to get some awesome results from this library.

1.3 Advice

Before you go any further, try running the code. Load up some data (either a DataFrame, or a list of dictionaries, where each dictionary is a row of data). Make a *column_descriptions* dictionary that tells us which attribute name in each row represents the value we're trying to predict. Pass all that into *auto_ml*, and see what happens!

Everything else in these docs assumes you have done at least the above. Start there and everything else will build on top. But this part gets you the output you're probably interested in, without unnecessary complexity.

CHAPTER 2

Formatting Data

WARNING: Lazers ahead.

Err, ok, more reasonably, the warning is to actually run your code first before reading any further. There aren't actually any lazers involved unless you're running on some kind of a crazy quantum computer (hardware geeks, feel free to submit a PR to make this joke technically accurate).

You probably don't need to read this page unless you're getting an error message.

Try running your code first, and come back here only if you're not getting the results you expect. The format of the data is designed to be intuitive.

2.1 Training data format

1. Must either be a pandas DataFrame, or a list filled with python dictionaries.
2. The non-header-row objects can be "sparse". You only have to include attributes on each object that actually apply to that row. In fact, we'd recommend passing in None or nan if you have missing values- knowing that a value is missing can itself be signal.

2.2 Header row information

The `column_descriptions` dictionary passed into `Predictor()` is essentially the header row. Here you've gotta specify some basic information about each "column" of data in the DataFrame or list of dictionaries. This column_descriptions object will tell us information about that "column" of data.

NOTE: We assume each column is a numerical column, unless you specify otherwise using one of the types noted below.

1. `attribute_name: 'output'` The `column_descriptions` dictionary must specify one of your attributes as the output column. This is what the `auto_ml` predictor will try to predict. Importantly, the data you pass into `.train()` should have the correct values for this column, so we can teach the algorithms what is right and what is wrong.

2. `attribute_name: 'categorical'` All attribute names that hold a string in any of the rows after the header row will be encoded as categorical data. If, however, you have any numerical columns that you want encoded as categorical data, you can specify that here.
3. `attribute_name: 'nlp'` If any of your data is a text field that you'd like to run some Natural Language Processing on, specify that in the header row. Data stored in this attribute will be encoded using TF-IDF, along with some other feature engineering (count of some aggregations like total capital letters, puncutation characters, smiley faces, etc., as well as a sentiment prediction of that text).
4. `attribute_name: 'ignore'` This column of data will be ignored.
5. `attribute_name: 'date'` Since ML algorithms don't know how to handle a Python datetime object, we will perform feature engineering on this object, creating new features like day_of_week, or minutes_into_day, etc. Then the original date field will be removed from the training data so the algorithsm don't throw a TypeError.

CHAPTER 3

Passing in your own feature engineering function

You can pass in your own function to perform feature engineering on the data. This will be called as the first step in the pipeline that `auto_ml` builds out.

You will be passed the entire X dataset (not the y dataset), and are expected to return the entire X dataset.

The advantage of including it in the pipeline is that it will then be applied to any data you want predictions on later. You will also eventually be able to run GridSearchCV over any parameters you include here.

Limitations: You cannot alter the length or ordering of the X dataset, since you will not have a chance to modify the y dataset. If you want to perform filtering, perform it before you pass in the data to train on.

CHAPTER 4

Using machine learning for analytics

Intended Audience:

1. Analysts. Yes, as an analyst who knows just a tiny bit of Python, you can run machine learning that will make your analytics both more accurate, and much cooler sounding.
2. Engineers looking to improve their models by figuring out what feature engineering to build out next.
3. Anyone interested in making business decisions, not just engineering decisions.

This is one of my favorite parts of this project: once the machines have learned all the complex patterns in the data, we can ask them what they've learned!

4.1 The code to make this work

It's super simple. When you train, simply pass in `ml_for_analytics=True`, like so: `ml_predictor.train(training_data, ml_for_analytics=True)`

Here's the whole code block that will get you analytics results in your console:

```
from auto_ml import Predictor

# If you pass in any categorical data as a number, tell us here and we'll take care
↪of it.
col_desc_dictionary = {col_to_predict: 'output', state_code: 'categorical'}

ml_predictor = Predictor(type_of_estimator='classifier', column_descriptions=col_desc_
↪dictionary)
# Can pass in type_of_estimator='regressor' as well

ml_predictor.train(df, ml_for_analytics=True)
# Wait for the machine to learn all the complex and beautiful patterns in your data...

# And this time, in your shell, it will print out the results for what it found was
↪useful in making predictions!
```

```
ml_predictor.predict(new_data)
# Where new_data is a single dictionary, or a DataFrame
```

4.2 Tangent time- what do you mean analytics from machine learning?

One of my favorite analogies for this (and really, for machine learning in general), is to think of a loan officer at a regional bank in, say, the 1940's or some other pre-computer era. She's been there for 30 years. She's seen thousands of loans cross her desk, and over time she's figured out what makes a loan likely to default, or likely to be healthy.

As a bright-eyed and bushy-tailed (mabye not bushy-tailed, you probably kept your furry tendencies in the closet back then) new loan officer, you probably wanted to learn her secrets! What was it that mattered? How could you read a loan application and figure out whether to give them a huge chunk of money or not?

It's the exact same process with machine learning. You feed the machine a ton of loan applications (in digital form now, not analog). You tell it which ones were good and which ones were bad. It learns the patterns in the data. And based on those patterns, it's able to learn in a matter of minutes what used to take our amazing loan officer decades of experience to figure out.

And just like our awesome loan officer, we can ask the machine to tell us what it learned.

4.3 Note- no free lunch

As with nearly everything in life (other than that occasional post-climb burrito where they get the crispiness just right, or that parallel parking job in the tight space you get on the first go), this isn't perfect. Once you dive deeper into the weeds, there are, of course, all kinds of caveats. However, this approach to analytics is typically much more robust than simply building a chart to compare two variables without considering anything else. The approach of just taking two variables and charting them against each other opens us up to huge classes of errors. The approaches used here will usually knock out most of those huge classes of errors, and instead open us up to much smaller classes of errors.

Again, this isn't perfect (and really, as an analyst, you should rarely if ever be claiming that something is perfect), but it is typically a vast improvement over most analytics workflows, and very straightforward to use.

4.4 Interpreting Results

The information coming from regression based models will be the coefficient for each feature. Note that by default, features are scaled to roughly the range of [0,1].

Roughly, you can read the results as "all else equal, we'd expect a change from being the smallest to the largest value on this particular variable will lead to [coefficient] impact on our output variable".

The information coming from random-forest-esque models will be roughly the amount of variance that is explained by this feature. These values will, in total, sum up to 1.

4.5 Interpreting Predicted Probability Buckets for Classifiers

Sometimes, it's useful to know how a classifier is doing very granularly, beyond just accuracy. For instance, pretend an expensive event (say, burning a whole batch of pastries) has a 5% chance of occurring.

If you train a model, obtaining 95% accuracy looks pretty bad on the surface- it's no better htan average! And in fact, you'll probably find that most (or all) of the predictions are 0- predicting that the pastries will not burn.

It's easy to disregard the model at this point.

However, we might still find some use for it, if we dive deeper into the predicted probabilities. Maybe, for 80% of deliveries, the model predicts 0 probability of fire, while for 20% of deliveries, the model predicts 25% chance of fire. That would be quite useful if it's accurate at each of those probabilities! We're able to correctly identify all the batches that have very low risk of fire, and a subset of the batches that are 5x as risky as our average batch. That sounds pretty promising!

That's what we report out in advanced scoring for classifiers.

We take the model's predicted probabilities on every item in the scoring dataset. Then we order the predicted probabilities from lowest to highest. We bucket those sorted predictions into 10 buckets, with the lowest bucket holding the 10% of the dataset that the model predicted the lowest probability for, and the highest bucket holding the 10% of the dataset that the model predicted the highest probability for.

Then, for each bucket, we simply report what the average predicted probability was, and what the actual event occurrence was, along with what the max and min predicted probabilities were for that bucket.

4.6 In the weeds

A collection of random thoughts that are much deeper into the weeds, and should really only be read once you've run the code above a few times, if at all.

1. The only types of models that support introspection like this currently are tree-based models (DecisionTree, RandomForest, XGBoost/GradientBoosted trees, etc.), and regression models (LinearRegression, LogisticRegression, Ridge, etc.).
2. Note that we are not doing any kind of PCA or other method of handling features that might be correlated with each other. So if you feed in two features that are highly related (say, number of items in order, and total order cost), you can oftentimes find some weird results, particularly from regression-based models. For instance, it might have an extra large positive coefficient for one of the features (number of items), and a negative coefficient on the other (total cost) to balance that out. If you find this happening, try running the model again with one of the two correlated features removed.
3. For forests handling two correlated variables, it will typically pick one of the variables as being highly important, and the other as relatively unimportant.
4. We scale all features to the range of roughly [0,1], so when you're interpreting the coefficients, they're fairly directly comparable in scale. For example, say we have two variables: number of items in order, and order total, in cents. Your order total variable might reasonably range from 50 to 10,000, while your number of items might only range from 1 - 10. Thus, the coefficient on the raw order total is going to be much smaller than the coefficient on number of items. But this might not accurately reflect the relative impact of these two features, because the order total feature can multiply that coefficient by a mugh larger range. When we scale both features to fall in the range of [0,1], we can now directly compare the coefficients. The way to read this then changes slightly. It's now "if we go from being the smallest to largest on this measure, what impact would we expect this to have on our output variable?".
5. Features with more granularity are typically more useful for models to differentiate on. Going back to our order total vs. number of items example, order total can potentially take on one of 10,000 values, while number of items can only take on 10 values. All else equal, the model will find order total more useful, simply because it has more options to perform the differentiation on.
6. The random forest will report results on features that are most broadly applicable, since it reports results on what reduces global variance/error. The regression models will report results on which features have the strongest impact WHEN THEY ARE PRESENT. So being in the state of Hawaii might come up very highly for our

regression, because we find that when a row holds data form the state of Hawaii, we need to make a large adjustment. However, the tree-based model likely won't report that variable to be too useful, since very little of your data likely comes from the state of Hawaii. It will likely find a more global variable like income (which is likely present in every row) to be more useful to reduce overall error/variance.

CHAPTER 5

Feature Responses

People love linear models for their explainability. Let's take a look at the classic Titanic example, and say that th "Fare" feature has a coefficient of 0.1. The way it's classically interpreted, that means that, holding all else equal, for every unit increase in fare, the person's likelihood of surviving increased by 10%.

There are two important parts here: holding all else equal, and the response of the output variable to this feature.

These are things we can get from tree-based models as well.

5.1 Methodology

Getting feature responses from a tree-based model is pretty easy. First, we take a portion of our training dataset (10k rows, by default, though user-configurable). Then, for each column, we find that column's standard deviation. Holding all else constant, we then increment all values in that column by one standard deviation. We get predictions for all rows, and compare them to our baseline predictions. The feature_response is how much the output predictions responded to this change in the feature. We repeat the process twice for each column, once incrementing by one std, and once decrementing by one std.

5.2 Output

The output is how much the output variable responded to these one std increments and decrements to each feature. The value we report is the average across all predictions. It is very literally, "holding all else constant, on average, how does the output respond to this change in a feature?"

5.3 CAVEATS!!

Tree-based models do not have linear relationships between features and the output variable. The relationships are more complex, which is why tree-based models can be more predictive. So even if we find that the *average* feature_response for an increase in fare is positive, that relationship may not hold for all passengers, and certainly not

equally for all passengers. For instance, the model might have found a group of 1-st class passengers who paid a lot of money to be in cabins that were as isolated from the main deck as possible, and thus, were less likely to survive. So even though for 3rd class passengers, an increase in fare meant they were more likely to survive, the predictions for some 1-st class passengers may actually decrease as fare increases.

Because of heteroskedasticity in the dataset, standard deviation might not be representative. Adding one std to every 3rd class passenger fare probably triples the cost of that ticket. But adding it to 1st Class passengers is a much more trivial percentage of their total fare.

Might lead to some unrealistic scenarios. Again, adding a std to 3rd class passengers probably makes most of them 2nd class passengers, but we're still holding class constant for all of them.

Finally, this entire approach is relatively experimental, and is not based on any journal articles or previous proofs that I know of.

CHAPTER 6

Categorical Ensembling

Let's say you operate in 100 different markets across the world. At some point in time, your local leader in Johannesburg is likely going to ask your for her own model, because her market is different than everyone else's market.

You'll probably respond with some version of "the giant monolith model we've got running right now should take everything in the dataset into account, so if your market behaves differently than everyone else's, the model will have already picked that up."

Except, if you're curious, you might try to go train a model just for Johannesburg, and find that, uh, well, that model's a bit more accurate for Johannesburg than your monolith model. Not dramatically, but maybe by 2%.

So then you modify your answer to this awesome local leader lady whose ask ended up being surprisingly good: "Uh, well, turns out that you had a good idea, but, um, I really, really don't want to support 100 different models in production. There are so many ways that can go wrong. So, I'd love to give you your own model, but I can't do that for everyone, so, until you can hire me 100x as many ML engineers, I can't do that yet."

Well buckle up, 'cause you're about to become a 100x ML engineer :)

categorical_ensembling is exactly this use case. auto_ml will automatically train one model for each category of thing (in this case, 100 different models, one for each local market). You still just call *trained_pipeline.predict(data)*, and it will return the prediction from the correct model. Heck, unless you use deep learning for each of the 100 models, the trained pipeline is saved into just one file, just like a normal auto_ml trained pipeline. You've basically abstracted away all the complexity of training 100 models for each of your 100 markets, while getting around a 2% performance boost.

How does this work in practice?

Slightly different UI: instead of *ml_predictor.train()*, you'll invoke *ml_predictor.train_categorical_ensemble()*.

You'll need to pass in an identifier for which column of data is our *categorical_column*. In our example above, it would be something like *ml_predictor.train_categorical_ensemble(df_train, categorical_column='market_name')*.

The only other thing you have to keep in mind is to pass in a value for *'market_name'* for each new item you want to get a prediction for. The API for this doesn't change from our normal predict API, I'm just calling out that this column should have a value in it.

```
` test_data = {'blah':  1, 'market_name':  'Mumbai'} trained_ml_predictor.
predict(test_data) `
```

A couple user-friendly features I've built on top:

1. Default category: Your company just launched 25 new markets- awesome! Except, uh, we obviously don't have any data for them yet, so we can't train a model for them. You can specify a default category, so that when we're asked to make a prediction for a category that wasn't in the training data, we use the predictor trained on this default category to generate the prediction. If you don't specify a default_category, we will choose the most-commonly-occurring (largest) category as the default. Or, you can specify *ml_predictor.train_categorical_ensemble(df_train, categorical_column='market_name', default_category='_RAISE_ERROR')* to raise an error if we're getting a prediction for a category that wasn't in our training data.
2. min_category_size: For a number of reasons, training an ML predictor on too few data points is messy. You can certainly clean the data yourself to have an effective min_category_size, but we built this in as a convenience for the user. If there's a category that has less than min_category_size observations in our training data, we won't train a model for that category, and we'll default to using the default_category to get predictions. The default value here right now is 5 (which seems really low, if you've got the data for it, I'd recommend a min size of at least a few thousand). To finish the example above: *ml_predictor.train_categorical_ensemble(df_train, categorical_column='market_name', default_category='Beijing', min_category_size=5000)*
3. You can still pass in any of the normal arguments for *.train()* that you know and love!

Performance Notes: A. We train up only one global transformation_pipeline to minimize disk space when serializing the models B. If *feature_learning=True* is passed in, we will train up one global feature_learning model- we will NOT train up one feature_learning model per category. The model we train for each category can then decide whether and how to use the features from our feature_learning model. Since each feature_learning model has to be serialized to disk separately right now, this design decision was made to reduce complexity, and the risk of things going wrong when transferring trained models to a production environment.

CHAPTER 7

Deep Learning & Feature Learning

7.1 Deep Learning in auto_ml

Deep Learning is available in auto_ml if you've got Keras and TensorFlow installed on your machine. It's simple: just choose *model_names='DeepLearningRegressor'* or model_names='DeepLearningClassifier'' when invoking *ml_predictor.train()*.

That's right, we've got automated Deep Learning that runs at production-ready speeds (roughly 1ms per prediction when getting predictions one-at-a-time).

7.2 Feature Learning

Deep Learning is great for learning features for you. It's not so amazing at turning those features into predictions (No Free Hunch all you will, I don't see too many people using Perceptrons as standalone models to turn features into predictions- frequently Gradient Boosting wins here).

So, why not use both models for what they're best at: Deep Learning to learn features for us, and Gradient Boosting to turn those features into accurate predictions?

That's exactly what the *feature_learning=True* param is for in auto_ml.

First, we'll train up a deep learning model on the *fl_data*, which is a dataset you have to pass into .train: *ml_predictor.train(df_train, feature_learning=True, fl_data=df_fl_data)*. This dataset should be a different dataset than your training data to avoid overfitting.

Once we've trained the feature_learning model, we'll split off it's final layer, and instead use it's penultimate layer, which outputs it's 10 most useful features. We'll hstack these features along with the rest of the features you have in your training data. So if you have 100 features in your training data, we'll add the 10 predicted features from the feature_learning model, and get to 110 features total.

Then we'll train a gradient boosted model (or any model of your choice- the full set of auto_ml models are available to you here) on this combined set of 110 features.

The result from this hybrid approach is up to 5% more accurate than either approach on their own.

All the typical best practices from deep learning apply here: be careful with overfitting, try to have a large enough dataset that this is actually useful, etc.

This isn't a super-novel approach (you could reasonably argue it's just a form of stacking which is already used in pretty much every Kaggle competition ever). However, conceptually, it's fun to think of taking a neural network, and swapping out it's simple output layer for a more complex gradient-boosted model as the output layer.

And, of course, this being auto_ml, it all runs automatically and at production-speeds (predictions take between 1 and 4 ms when getting predictions one-at-a-time).

Let me know if you have any feedback or improvements! It's pretty exciting to find an application for Deep Learning that improves predictions for standard classification and regression problems by up to 5%.

CHAPTER 8

Properly Formal API Documenation

8.1 auto_ml

class Predictor(*type_of_estimator*, *column_descriptions*)

Parameters

- **type_of_estimator** (*'regressor' or 'classifier'*) – Whether you want a classifier or regressor
- **column_descriptions** (*dictionary, where each attribute name represents a column of data in the training data, and each value describes that column as being either ['categorical', 'output', 'nlp', 'date', 'ignore'] Note that 'continuous' data does not need to be labeled as such (all columns are assumed to be continuous unless labeled otherwise)*) – A key/value map noting which column is `'output'`, along with any columns that are `'nlp'`, `'date'`, `'ignore'`, or `'categorical'`. See below for more details.

ml_predictor.**train**(*raw_training_data*, *user_input_func=None*, *optimize_final_model=False*, *perform_feature_selection=None*, *verbose=True*, *ml_for_analytics=True*, *model_names='GradientBoosting'*, *perform_feature_scaling=True*, *calibrate_final_model=False*, *verify_features=False*, *cv=2*, *feature_learning=False*, *fl_data=None*, *prediction_intervals=False*)

Return type None. This is purely to fit the entire pipeline to the data. It doesn't return anything- it saves the fitted pipeline as a property of the `Predictor` instance.

Parameters

- **raw_training_data** (*DataFrame, or a list of dictionaries, where each dictionary represents a row of data. Each row should have both the training features, and the output value we are trying to predict.*) – The data to train on. See below for more information on formatting of this data.

- **user_input_func** (*function*) – [default- None] A function that you can define that will be called as the first step in the pipeline, for both training and predictions. The function will be passed the entire X dataset. The function must not alter the order or length of the X dataset, and must return the entire X dataset. You can perform any feature engineering you would like in this function. Using this function ensures that you perform the same feature engineering for both training and prediction. For more information, please consult the docs for scikit-learn's `FunctionTransformer`.
- **ml_for_analytics** (*Boolean*) – [default- True] Whether or not to print out results for which features the trained model found useful. If `True`, auto_ml will print results that an analyst might find interesting.
- **optimize_final_model** (*Boolean*) – [default- False] Whether or not to perform GridSearchCV on the final model. True increases computation time significantly, but will likely increase accuracy.
- **perform_feature_selection** (*Boolean*) – [default- True for large datasets, False for small datasets] Whether or not to run feature selection before training the final model. Feature selection means picking only the most useful features, so we don't confuse the model with too much useless noise. Feature selection typically speeds up computation time by reducing the dimensionality of our dataset, and tends to combat overfitting as well.
- **take_log_of_y** (*Boolean*) – For regression problems, accuracy is sometimes improved by taking the natural log of y values during training, so they all exist on a comparable scale.
- **model_names** (*list of strings*) – [default- relevant 'GradientBoosting'] Which model(s) to try. Includes many scikit-learn models, deep learning with Keras/TensorFlow, and Microsoft's LightGBM. Currently available options from scikit-learn are ['ARDRegression', 'AdaBoostClassifier', 'AdaBoostRegressor', 'BayesianRidge', 'ElasticNet', 'ExtraTreesClassifier', 'ExtraTreesRegressor', 'GradientBoostingClassifier', 'GradientBoostingRegressor', 'Lasso', 'LassoLars', 'LinearRegression', 'LogisticRegression', 'MiniBatchKMeans', 'OrthogonalMatchingPursuit', 'PassiveAggressiveClassifier', 'PassiveAggressiveRegressor', 'Perceptron', 'RANSACRegressor', 'RandomForestClassifier', 'RandomForestRegressor', 'Ridge', 'RidgeClassifier', 'SGDClassifier', 'SGDRegressor']. If you have installed XGBoost, LightGBM, or Keras, you can also include ['DeepLearningClassifier', 'DeepLearningRegressor', 'LGBMClassifier', 'LGBMRegressor', 'XGBClassifier', 'XGBRegressor']. By default we choose scikit-learn's 'GradientBoostingRegressor' or 'GradientBoostingClassifier', or if XGBoost is installed, 'XGBRegressor' or 'XGBClassifier'.
- **verbose** – [default- True] I try to give you as much information as possible throughout the process. But if you just want the trained pipeline with less verbose logging, set verbose=False and we'll reduce the amount of logging.
- **perform_feature_scaling** – [default- True] Whether to scale values, roughly to the range of {-1, 1}. Scaling values is highly recommended for deep learning. auto_ml has it's own custom scaler that is relatively robust to outliers.
- **cv** – [default- 2] How many folds of cross-validation to perform. The default of 2 works well for very large datasets. It speeds up training speed, and helps combat overfitting. However, for smaller datasets, cv of 3, or even up to 9, might make more sense, if you're ok with the trade-off in training speed.
- **calibrate_final_model** – [default- False] Whether to calibrate the probability predictions coming from the final trained classifier. Usefulness depends on your scoring metric, and model. The default auto_ml settings mean that the model does not necessarily need to be calibrated. If True, you must pass in values for X_test and y_test as well. This is the

dataset we will calibrate the model to. Note that this means you cannot use this as your test dataset once the model has been calibrated to them.

- **verify_features** – [default- False] Allows you to verify that all the same features are present in a prediction dataset as the training datset. False by default because it increases serialized model size by around 1MB, depending on your dataset. In order to check whether a prediction dataset has the same features, invoke `trained_ml_pipeline.named_steps['final_model'].verify_features(prediction_data)`. Kind of a clunky UI, but a useful feature smashed into the constraints of a sklearn pipeline.
- **feature_learning** – [default- False] Whether or not to use Deep Learning to learn features from the data. The learned features are then predicted for every row in the training data, and fed into a final model (by default, gradient boosting) to turn those features and the original features into the most accurate predictions possible. If True, you must pass in fl_data as well. For more details, please visit the feature_learning page in these docs.
- **fl_data** – If feature_learning=True, then this is the dataset we will fit the deep learning model on. This dataset should be different than your df_train dataset.
- **prediction_intervals** – [default- False] In addition to predicting a single value, regressors can return upper and lower bounds for that prediction as well. If you pass True, we will return the 95th and 5th percentile (the range we'd expect 90% of values to fall within) when you get predicted intervals. If you pass in two float values between 0 and 1, we will return those particular predicted percentiles when you get predicted intervals. To get these additional predicted values, you must pass in True (or two of your own float values) at training time, and at prediction time, call `ml_predictor.predict_intervals()`. `ml_predictor.predict()` will still return just the prediction.

`ml_predictor.`**`train_categorical_ensemble`**(*df_train*, *categorical_column*, *min_category_size=5*, *default_category='most_frequently_occurring_category'*)

Parameters

- **df_train** – Same as for .train
- **categorical_column** – The column of data that holds the category you want to train each model on. If you want to train a model for each market you operate in, *categorical_column='market_name'*.
- **min_category_size** – [default- 5] The minimum size of a category in the training data. If a category has fewer than this number of observations, we will not train a model for it.
- **default_category** – [Default- most frequently occurring category in the training data] When getting predictions for a category that was not in our training data, which category should we use? By default, uses the largest category from the training data. Can also take on the value "_RAISE_ERROR", which will predictably raise an error.

For more details, please visit the "categorical_ensembling" page in the docs.

Return type None. This is purely to fit the entire pipeline to the data. It doesn't return anything- it saves the fitted pipeline as a property of the `Predictor` instance.

`ml_predictor.`**`predict`**(*prediction_rows*)

Parameters **prediction_rows** – A single dictionary, or a DataFrame, or list of dictionaries. For production environments, the code is optimized to run quickly on a single row passed in as a dictionary (taking around 1 millisecond for the entire pipeline). Batched predictions on thousands of rows at a time are generally more efficient if you're getting predictions for a larger dataset.

Return type list of predicted values, of the same length and order as the `prediction_rows` passed in. If a single dictionary is passed in, the return value will be the predicted value, not nested in a list (so just a single number or predicted class).

`ml_predictor.`**`predict_proba`**(*prediction_rows*)

Parameters **`prediction_rows`** – Same as for predict above.

Return type Only works for 'classifier' estimators. Same as above, except each row in the returned list will now itself be a list, of length (number of categories in training data) The items in this row's list will represent the probability of each category.

`ml_predictor.`**`score`**(*X_test*, *y_test*, *verbose=2*)

Return type number representing the trained estimator's score on the validation data.

Parameters **`verbose`** – [Default- 2] If 3, even more detailed logging will be included.

`ml_predictor.`**`predict_intervals`**(*prediction_rows*, *return_type='df'*)

Return type dict for single predictions, list of lists if getting predictions on multiple rows. The return type can also be specified using return_type below. The list of predicted values for each row will always be in this order: `[prediction, prediction_lower, prediction_median, prediction_upper]`. Similarly, each returned dict will always have the properties `{'prediction': None', 'prediction_lower': None, 'prediction_median': None, 'prediction_upper': None}`

Parameters **`return_type`** – [Default- dict for single prediction, list of lists for multiple predictions] Accepted values are `'df'`, `'list'`, `'dict'`. If `'df'`, we will return a pandas DataFrame, with the columns `[prediction, prediction_lower, prediction_median, prediction_upper]`. If `'list'`, we will return a single (non-nested) list for single predictions, and a list of lists for batch predictions. If `'dict'`, we will return a single (non-nested) dictionary for single predictions, and a list of dictionaries for batch predictions.

`ml_predictor.`**`save`**(*file_name='auto_ml_saved_pipeline.pkl'*, *verbose=True*)

Parameters

- **`file_name`** (*`string`*) – [OPTIONAL] The name of the file you would like the trained pipeline to be saved to.
- **`verbose`** (*`Boolean`*) – If `True`, will log information about the file, the system this was trained on, and which features to make sure to feed in at prediction time.

Return type the name of the file the trained ml_predictor is saved to. This function will serialize the trained pipeline to disk, so that you can then load it into a production environment and use it to make predictions. The serialized file will likely be several hundred KB or several MB, depending on number of columns in training data and parameters used.

CHAPTER 9

Installation

```
pip install auto_ml
```

CHAPTER 10

Core Functionality Example

auto_ml is designed for production. Here's an example that includes serializing and loading the trained model, then getting predictions on single dictionaries, roughly the process you'd likely follow to deploy the trained model.

```
from auto_ml import Predictor
from auto_ml.utils import get_boston_dataset
from auto_ml.utils_models import load_ml_model

# Load data
df_train, df_test = get_boston_dataset()

# Tell auto_ml which column is 'output'
# Also note columns that aren't purely numerical
# Examples include ['nlp', 'date', 'categorical', 'ignore']
column_descriptions = {
  'MEDV': 'output'
  , 'CHAS': 'categorical'
}

ml_predictor = Predictor(type_of_estimator='regressor', column_descriptions=column_
↪descriptions)

ml_predictor.train(df_train)

# Score the model on test data
test_score = ml_predictor.score(df_test, df_test.MEDV)

# auto_ml is specifically tuned for running in production
# It can get predictions on an individual row (passed in as a dictionary)
# A single prediction like this takes ~1 millisecond
# Here we will demonstrate saving the trained model, and loading it again
file_name = ml_predictor.save()

trained_model = load_ml_model(file_name)

# .predict and .predict_proba take in either:
```

```
# A pandas DataFrame
# A list of dictionaries
# A single dictionary (optimized for speed in production evironments)
predictions = trained_model.predict(df_test)
print(predictions)
```

CHAPTER 11

XGBoost, Deep Leaarning with TensorFlow & Keras, and LightGBM

auto_ml has all three of these awesome libraries integrated! Generally, just pass one of them in for model_names. *ml_predictor.train(data, model_names=['DeepLearningClassifier'])*

Available options are - *DeepLearningClassifier* and *DeepLearningRegressor* - *XGBClassifier* and *XGBRegressor* - *LGBMClassifer* and *LGBMRegressor*

All of these projects are ready for production. These projects all have prediction time in the 1 millisecond range for a single prediction, and are able to be serialized to disk and loaded into a new environment after training.

Depending on your machine, they can occasionally be difficult to install, so they are not included in auto_ml's default installation. You are responsible for installing them yourself. auto_ml will run fine without them installed (we check what's isntalled before choosing which algorithm to use). If you want to try the easy install, just *pip install -r advanced_requirements.txt*, which will install TensorFlow, Keras, and XGBoost. LightGBM is not available as a pip install currently.

CHAPTER 12

Classification

Binary and multiclass classification are both supported. Note that for now, labels must be integers (0 and 1 for binary classification). auto_ml will automatically detect if it is a binary or multiclass classification problem- you just have to pass in *ml_predictor = Predictor(type_of_estimator='classifier', column_descriptions=column_descriptions)*

CHAPTER 13

Advice

Before you go any further, try running the code. Load up some dictionaries in Python, where each dictionary is a row of data. Make a `column_descriptions` dictionary that tells us which attribute name in each row represents the value we're trying to predict. Pass all that into `auto_ml`, and see what happens!

Everything else in these docs assumes you have done at least the above. Start there and everything else will build on top. But this part gets you the output you're probably interested in, without unnecessary complexity.

CHAPTER 14

What this project does

Automates the whole machine learning process, making it super easy to use for both analytics, and getting real-time predictions in production.

A quick overview of buzzwords, this project automates:

1. Analytics (pass in data, and auto_ml will tell you the relationship of each variable to what it is you're trying to predict).
2. Feature Engineering (particularly around dates, and soon, NLP).
3. Robust Scaling (turning all values into their scaled versions between the range of 0 and 1, in a way that is robust to outliers, and works with sparse matrices).
4. Feature Selection (picking only the features that actually prove useful).
5. Data formatting (turning a list of dictionaries into a sparse matrix, one-hot encoding categorical variables, taking the natural log of y for regression problems).
6. Model Selection (which model works best for your problem).
7. Hyperparameter Optimization (what hyperparameters work best for that model).
8. Ensembling Subpredictors (automatically training up models to predict smaller problems within the meta problem).
9. Ensembling Weak Estimators (automatically training up weak models on the larger problem itself, to inform the meta-estimator's decision).
10. Big Data (feed it lots of data).
11. Unicorns (you could conceivably train it to predict what is a unicorn and what is not).
12. Ice Cream (mmm, tasty. . .).
13. Hugs (this makes it much easier to do your job, hopefully leaving you more time to hug those those you care about).

Index

P

predict() (ml_predictor method), 21
predict_intervals() (ml_predictor method), 22
predict_proba() (ml_predictor method), 22
Predictor (built-in class), 19

S

save() (ml_predictor method), 22
score() (ml_predictor method), 22

T

train() (ml_predictor method), 19
train_categorical_ensemble() (ml_predictor method), 21