Auto_ML
a.k.a
Automated Machine Learning

Tags:

(python) (pandas) (pipelines) (supervised-learning) (sklearn) (featuretools) (feature-engineering) (visual-analytics) (pandas_profiling) (plotly) (flask) (javascript) (html) (css) (bootstrap) (ajax)

PREFACE

I'll start this documentation with a standard example, with numerous constraints.
These constraints will be relaxed one by one, and at the end, a demonstration of the entire flow will be presented.
These relaxations will be addressed as (R1,R2,R3..)

NOTE:
Due to the sensitivity of the dataset(s) involved, I will deliberately blur out some sections and will only share important attributes of the dataset(s) required by the application to run.

CONTENT

INTRODUCTION

- In a standard supervised learning problem, we usually deal with a single dataset like the following (We'll call it `d1.csv`):

X1	X2	X3	Y
A	1	0	1.5
A	2	0	5
B	3	1	2.3
A	4	0	6
B	5	1	9
B	6	1	3

- We'll be dealing with a Regression problem throughout the course of this documentation, but, as you'll see the flow can deal with classification problems as well.
- So, we take `X = (X1,X2,X3)` as our predictor variables and `Y` as the target variable.
- We split the dataset into training and test part:

  `(X_train,y_train),(X_test,y_test)`
- We'll use `KNeighborsRegressor` as our model here.

- The training part goes as follows:
```
regressor = KNeighborsRegressor()
regressor.fit(X_train,y_train)
```
- After the training part, we check how the fitted model performed on the test dataset like so:
```
regressor.score(X_test,y_test)
```
- By default `score()` returns the `r-Squared` `score`.
- So, now when new data`(X_pred)` comes, we can make predictions like so:
```
predictions = regessor.predict(X_pred)
```
- `KNeighborsRegressor` accepts different parameters like `n_neighbors, weights, metric` etc.

  What if, I want to test different values of these parameters against my model score ?

  We'll see this in the next slide.

R1: Hyper Parameter Grid

- By default, `n_neighbors=5` and `weights='uniform'` for `KNeighborsRegressor`.
- Now, we want to check the model score for different values of these parameters:

```
n_neighbors = [5,6,7]
weights = ['uniform','distance']
```

- So, we want someone who would go through all these parameters, calculate respective model scores and return the best parameters.
- `Python Pipelines` to our rescue.
- In a very raw sense, whatever operations we want our data to go though, we'll package them in a `Pipeline` and then ask `sklearn` to do all those operations automatically.

- First, we create a hyper parameter grid, or the parameter space as follows:

```
parameter_grid = {
     ‘model__n_neighbors’: [5,6,7],
    ‘model__weights’: [‘uniform’,’weights’]
                  }
```

- Now, we create a `Pipeline` like so:

```
pipe =
Pipeline([(‘model’,KneighborsRegrssor())])
```

- Finally, we want something that'll plug these together: `GridSearchCV`

  Here, `cv` stands for `Cross-Validation`.

  Yes, `GridSearchCV` will take care of cross-validation also.

  We can use `RandomSearchCV` or any other parameter optimization algorithm here.

```
grid_cv = GridSearchCV(pipe,parameter_grid)
grid_cv.fit(X_train_y_train)
```

- Here, `GridSearchCV` would iterate over every possible combination of the parameter space provided, and will rank them according to the model score.
  We can get the best parameters like so:
  ```
  grid_cv.best_estimator_
  grid_cv.best_score_
  ```
- To make predictions, we use `grid_cv` object like so:
  ```
  predictions = grid_cv.predict(X_pred)
  ```
- So, now we have the ability to pass different values to the parameters of our model, and `GridSearchCV` will automatically return the best parameter corresponding to the best model score.
  The next slide will deal with another such constraint.

R2: Multiple Datasets

- So far, we've been dealing with a single dataset which had our predictor variables and the target variable: (X,Y)
- In actual scenarios, we almost never deal with single datasets.
  In fact, these 'single' datasets are engineered with the help of different datasets which are then aggregated to form a single dataset.
- Now, we'll add another dataset `d2.csv` to our flow:

pid	X4
1	5
1	6
2	3
2	1.5
2	7
3	9
3	2
4	2.3
4	6.5
4	8
5	2
5	9
6	7
6	5

R3: Relationships

- So now, instead of a single dataset, we have a set of datasets: `[d1.csv,d2.csv]`
- We'd like to 'merge' them somehow, so that we have an aggregated dataset and we can carry on with our training process.
- So, to somehow 'merge' them, we need to define relationships among the datasets.
- We'll deal with 3 kinds of relationships:

`table_1`:

id_1	id_2	X1

`table_2`:

id_3	id_4	X3

`table_3`:

id_5	X4	X5

(1) Direct:

`table_1` is related to `table_2` via `(id_1,id_3)`

Representation: `table_1.id_1 => table_2.id_3`

(2) Indirect-Parent:

`table_1` is related to `table_2` via `(id_2,id_4)`

Representation: `table_2.id_4 => table_1.id_2`

(3) Indirect-Child:

`table_2` is related to `table_3` via `(id_4,id_5)`

Representation: `table_3.id_5 => table_2.id_4`

- Notice the change in notation, whenever dealing with multiple datasets, we'll always establish relationships as **parent-child**.
- Coming back to our original set `[d1.csv,d2.csv]`, Here, we represent the relationship as:
  `(Direct) d1.csv.X2 => d2.csv.pid`
- So, we have now established the relationships among the datasets, now we can move on to the next constraint.

R4: Feature Engineering

- Now that we have the relationships established, we'll move on to the aggregation and merging part.
- Consider `[d1.csv,d2.csv]` again:

X1	X2	X3	Y
A	1	0	1.5
A	2	0	5
B	3	1	2.3
A	4	0	6
B	5	1	9
B	6	1	3

`d1.csv.X2 => d2.csv.pid`

pid	X4
1	5
1	6
2	3
2	1.5
2	7
3	9
3	2
4	2.3
4	6.5
4	8
5	2
5	9
6	7
6	5

- Notice something here, for every `X2` in `d1.csv`, we have multiple `pid` in `d2.csv`.
  How about finding `MEAN(X4)` for every `X2` in `d1.csv` ? Or `MAX(X4)` ? Or `MIN(X4)` ?
  Let's see how can we do that..

- We could do this manually, but then the project would be called ‘Manual-ML’ not ‘Auto-ML’ (right?).

- FeatureTools to the rescue !

- Let’s see how FeatureTools automates this, first, we create entities for `[d1.csv,d2.csv]` like so:

```
import featuretools as ft
es = ft.EntitySet(id=’ml-flow’)
es =
es.entity_from_dataframe(entity_id=’d1’,dat
aframe=d1,index=’X2’)
es =
es.entity_from_dataframe(entity_id=’d2’,dat
aframe=d2,make_index=True,index=’child_id’)
```

  We add the relationship like so:

```
es =
es.add_relationship(ft.Relationship(es['d1'
]['X2'],es['d2']['pid']))
```

Now, we generate features:

```
feature_matrix, feature_names =
ft.dfs(entityset=es,target_entity='d1')
```

where `feature_matrix` is the resultant of our aggregations and `feature_names` represents the names of the generated features.

Let's see what `feature_matrix` looks like:

X1	X3	Y	SUM(d2.X4)	STD(d2.X4)	MAX(d2.X4)	MIN(d2.X4)	MEAN(d2.X4)	COUNT(d2)
A	0	1.5	11	0.707107	6	5	5.5	2
A	0	5	10	2.828427	7	3	5	3
B	1	2.3	11	4.949747	9	2	5.5	2
A	0	6	8	3.526812	8	8	8	3
B	1	9	11	4.949747	9	2	5.5	2
B	1	3	12	1.414214	7	5	6	2

- So, we have the ability to generate features automatically, now we'll return to the model training part.

R5: Multiple Models

- Now that we have generated a single aggregated dataset, we return to the model training part.
- Earlier, we were using a single model KNeighborsRegressor and optimizing the model score with respect to the parameters it accepts.
- What if, we want to add other models to the flow ?
- We'll handle this my modifying the parameter grid, which we had defined earlier.

  Say, we decide to add LinearRegressor to the flow:

```
parameter_grid = {
                       {
    'model': KNeighborsRegressor(),
    'model__n_neighbors': [5,6,7],
    'model__weights': ['uniform','distance']
                        },
                        {
    'model': LinearRegression()
                        }
                       }
```

- Remaining part remains as it is:

```
pipe =
Pipeline([('model',KneighborsRegrssor())])
grid_cv = GridSearchCV(pipe,parameter_grid)
grid_cv.fit(X_train_y_train)
score = grid_cv.score(X_test,y_test)
```

- Finally, for making predictions:

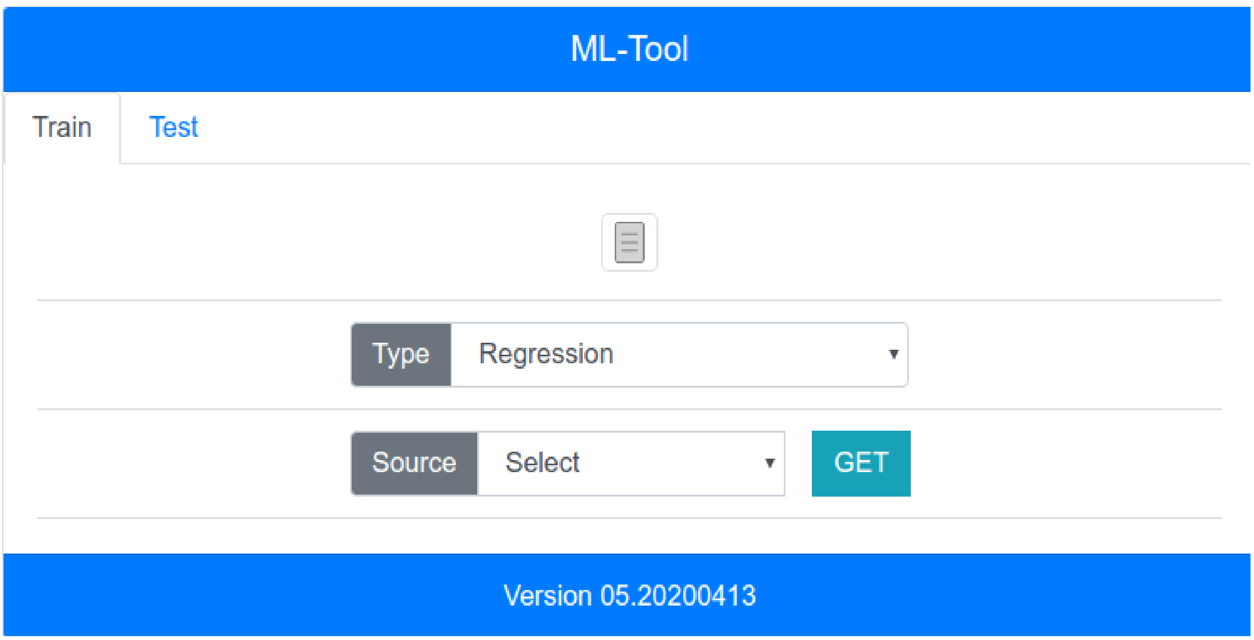
```
predictions = grid_cv.predict(X_pred)
```

- So far, we've been dealing with hypothetical datasets `[d1.csv,d2.csv]`,

  Let's move on to a real life problem !

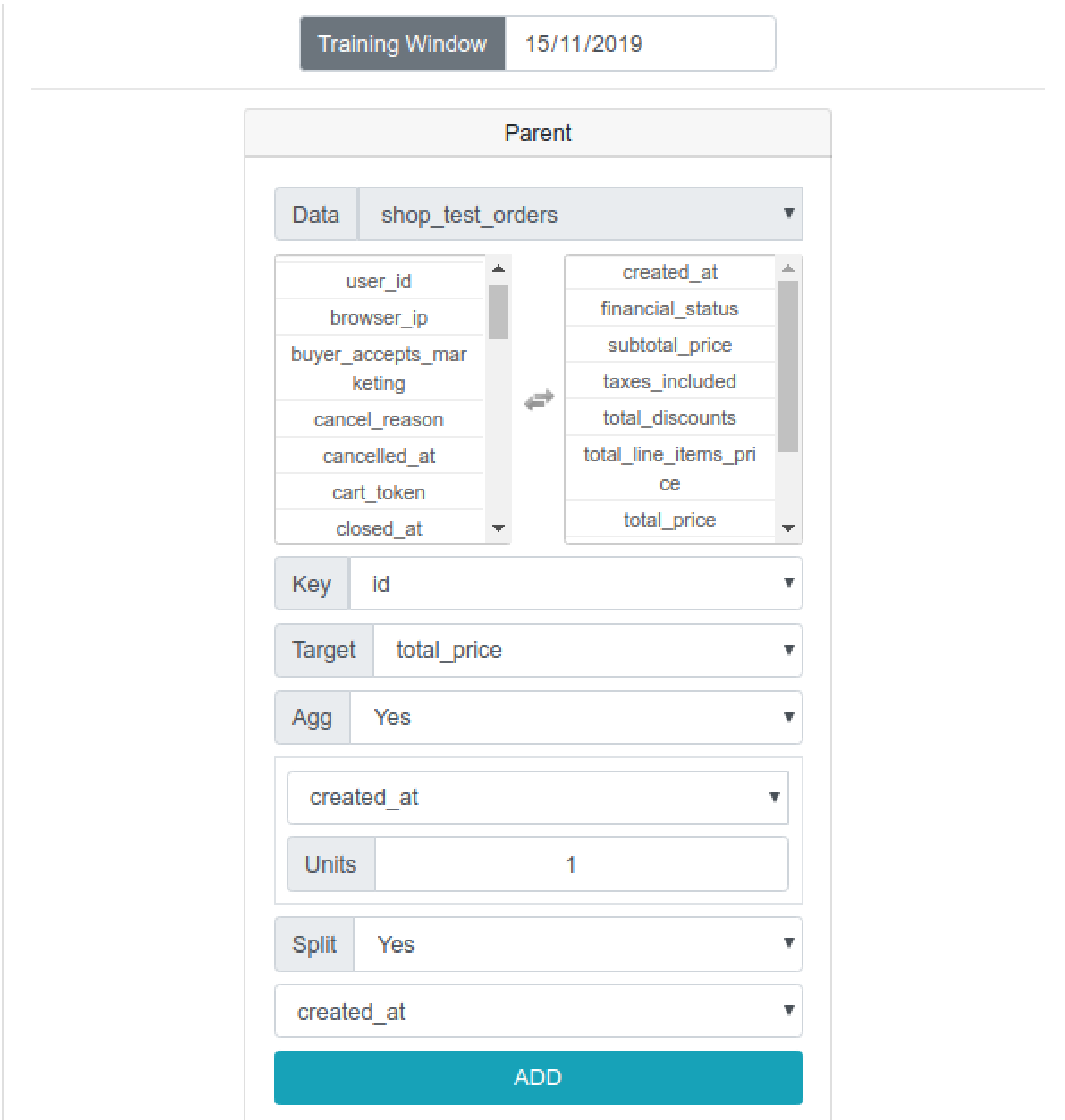
Demonstration: An end-to-end Application

- With all the constraints relaxed, we'll now see whatever we have done so far in action, via this web application:



Here, we mention the type of the problem we'll be dealing with, Regression in this context.
Also, the source.
The data currently resides at Google BigQuery Database. So, we'll select that.

- We'll take the training window as: `15-11-2019`
- We'll start adding datasets, like so:



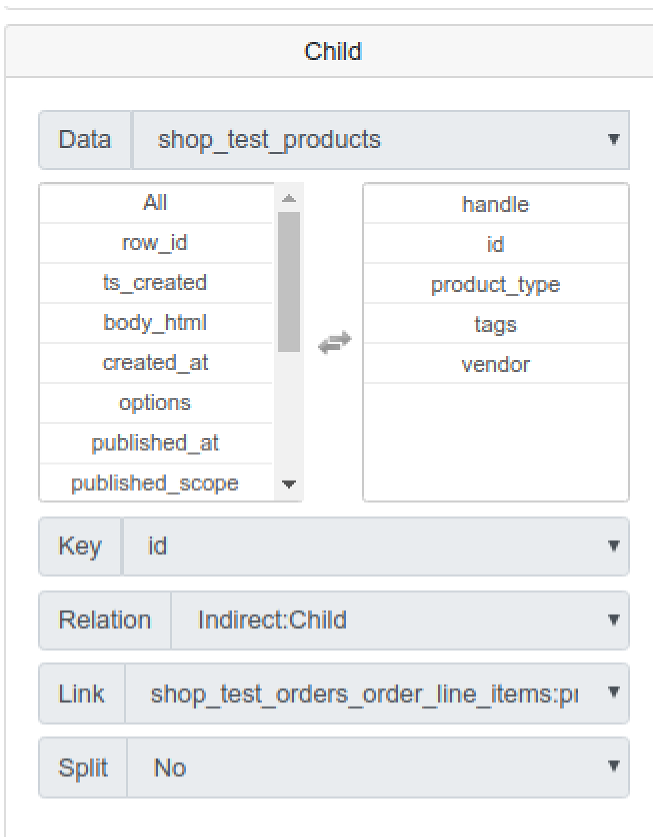
This will be our parent dataset: `shop_test_orders`.

- We'll start with child datasets:
  Child Dataset 1:

Name: `shop_test_orders_order_line_items`

We'll add another child dataset, then we'll go on to explain the problem and the relationship among the datasets.

- Child Dataset 2:



Name: `shop_test_products`

- We'll start with the problem definition:

Datasets: [
```
shop_test_orders,
shop_test_orders_order_line_items,
shop_test_products]
```

We have daily sales (all products) of a particular client, so, we train our models using data before `15-11-2019` and check performance on the data beyond that date.

Split Column: `created_at`
Target Variable: `total_price`

Relationships:

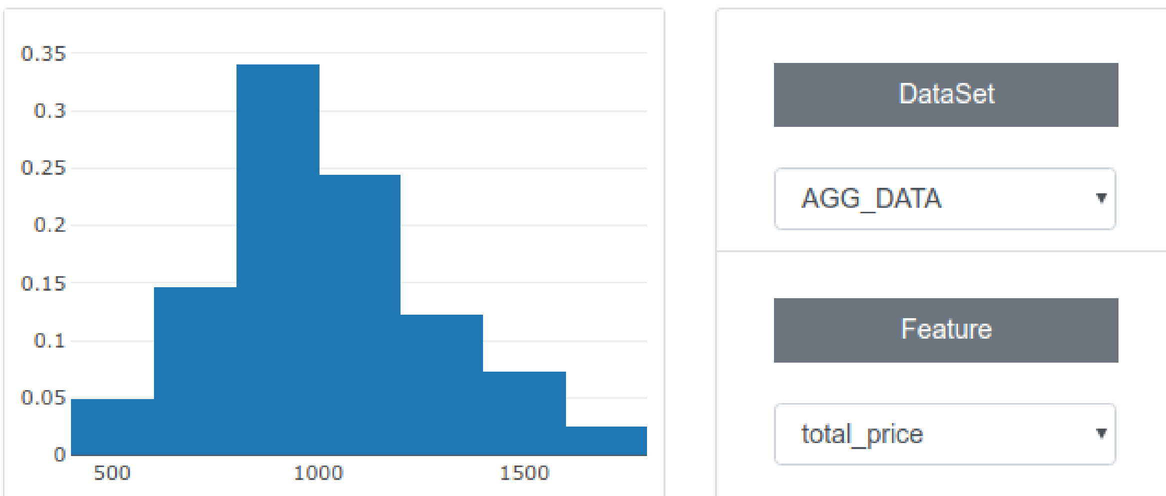
(1) Direct:

`shop_test_orders.id => shop_test_orders_order_line_items.Orders_id`

(2) Indirect-Child:

`shop_test_products.id => shop_test_orders_order_line_items.product_id`

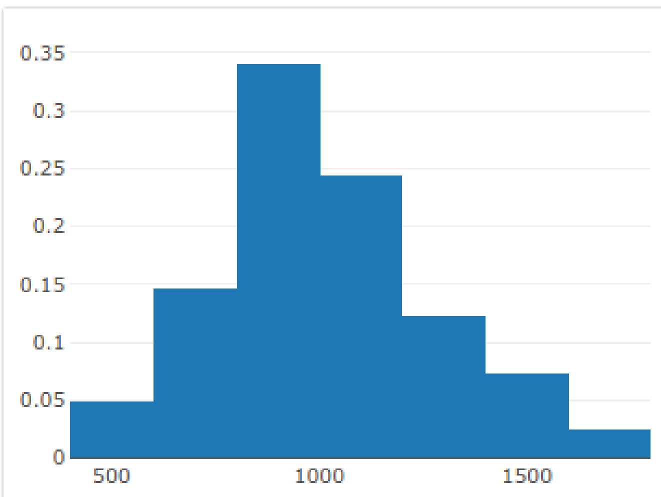
- Before initiating the training part, I'll mention this **summary section** in the flow, which can be used for initial exploratory purposes, for example, here we can see the distribution of our target variable with some summary statistics:

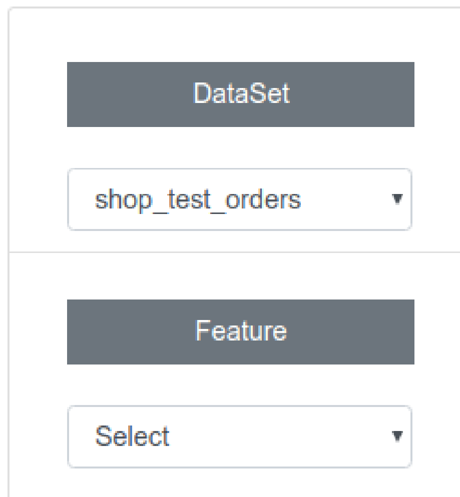


type	NUM	**(mean,median)**	(1.03K,998.02)
p_missing	0.00	**mode**	670.49
p_infinite	0.00	**(variance,std)**	(71.38K,267.17)
p_unique	1.00	**skewness**	0.45
memory_size	456.00	**kurtosis**	0.08
(min,max)	(583.82,1.71K)	**(range,iqr)**	(1.13K,244.59)

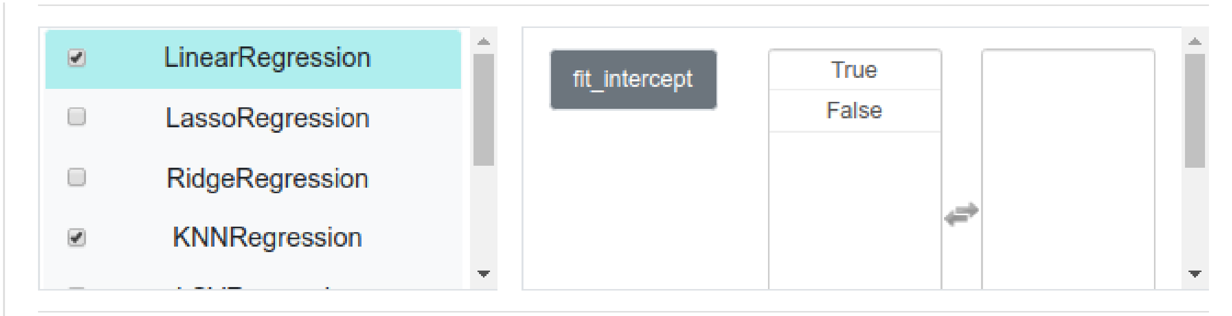
We can also see some statistics regarding the selected dataset:

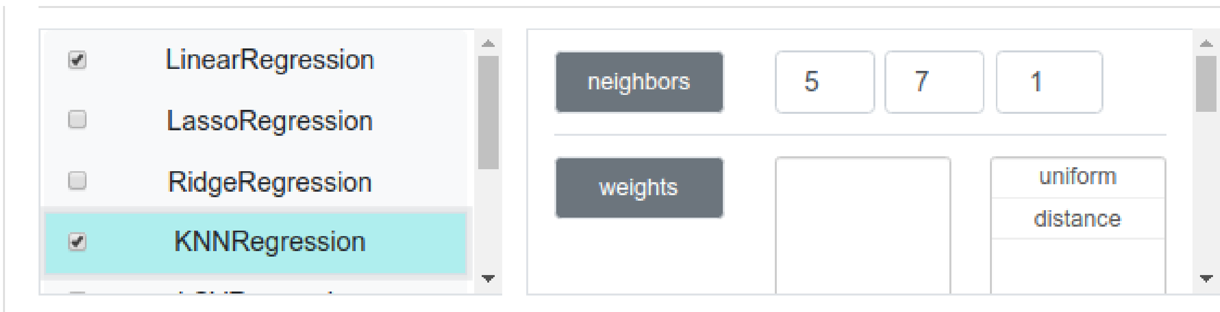
n	1.52K	n_vars_with_missing	0.00
n_var	7.00	n_vars_all_missing	0.00
memory_size	0.28M	p_cells_missing	0.00
record_size	183.16	n_duplicates	1.07K
n_cells_missing	0.00	p_duplicates	0.71
n_vars_with_missing	0.00	complex	0
n_vars_all_missing	0.00	unsupported	0





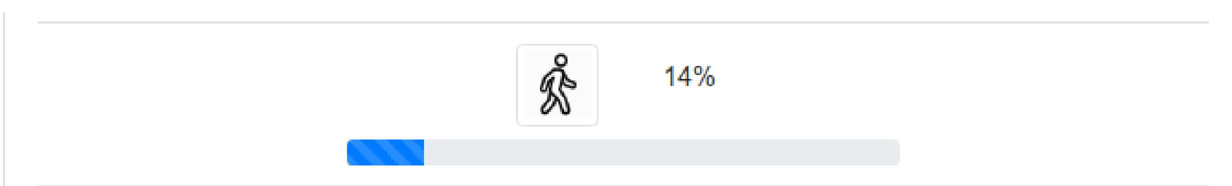
- Now, we'll come to the training part, we'll choose two models for the time being:
  `LinearRegressor` and `KNeighborsRegressor`
  We choose the models and the hyper-parameter in the following fashion:

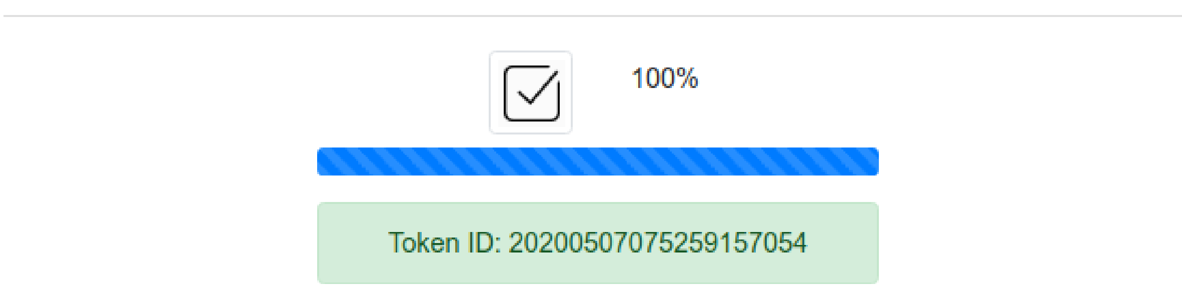




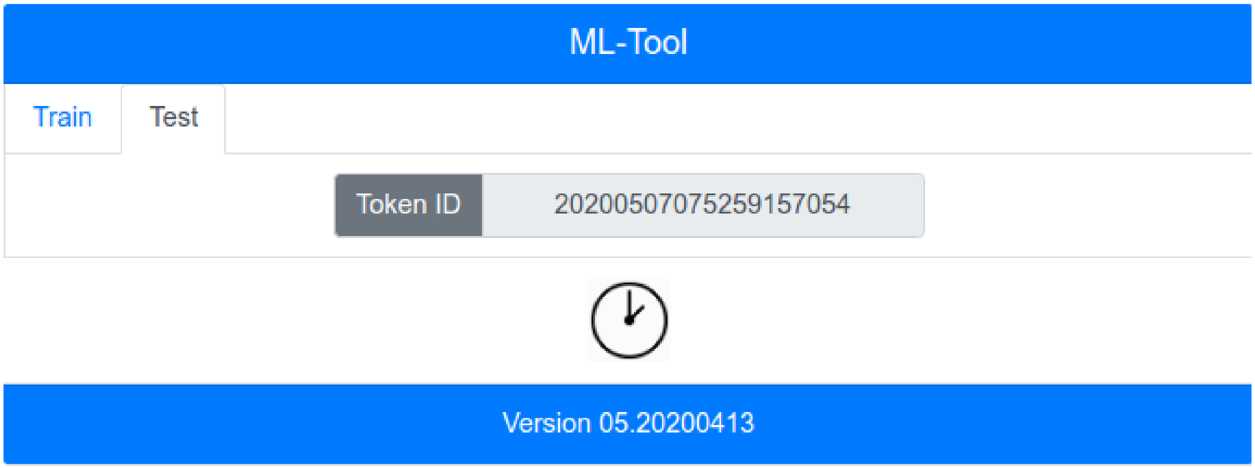
Here we've chosen `LinearRegressor` with default parameters and `KNeighborsRegressor` with different values of `n_neighbors` and `weights`.

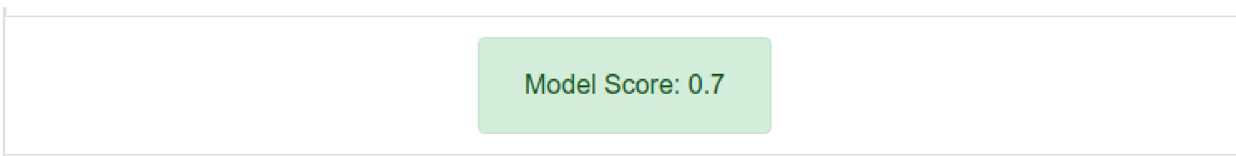
- We'll start the train part now:





- We now move on to the test part:





So, we get a **Model score** of `0.7`.

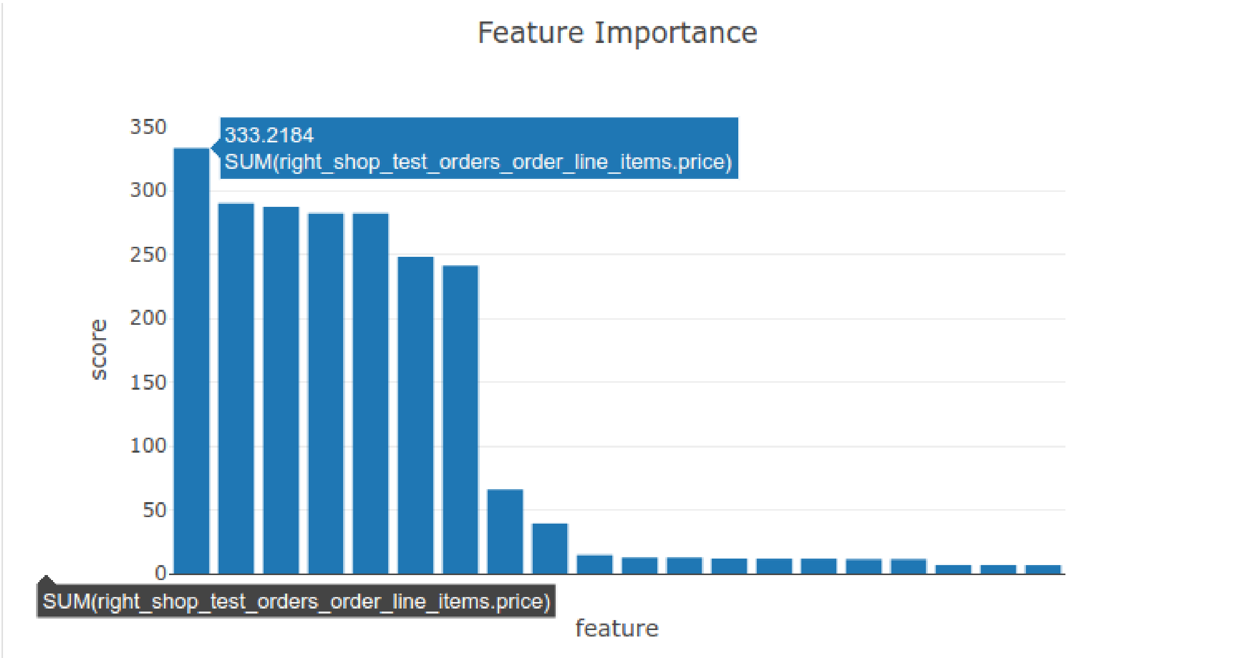
Not great, but we used only two models.

That's the best part of this tool,

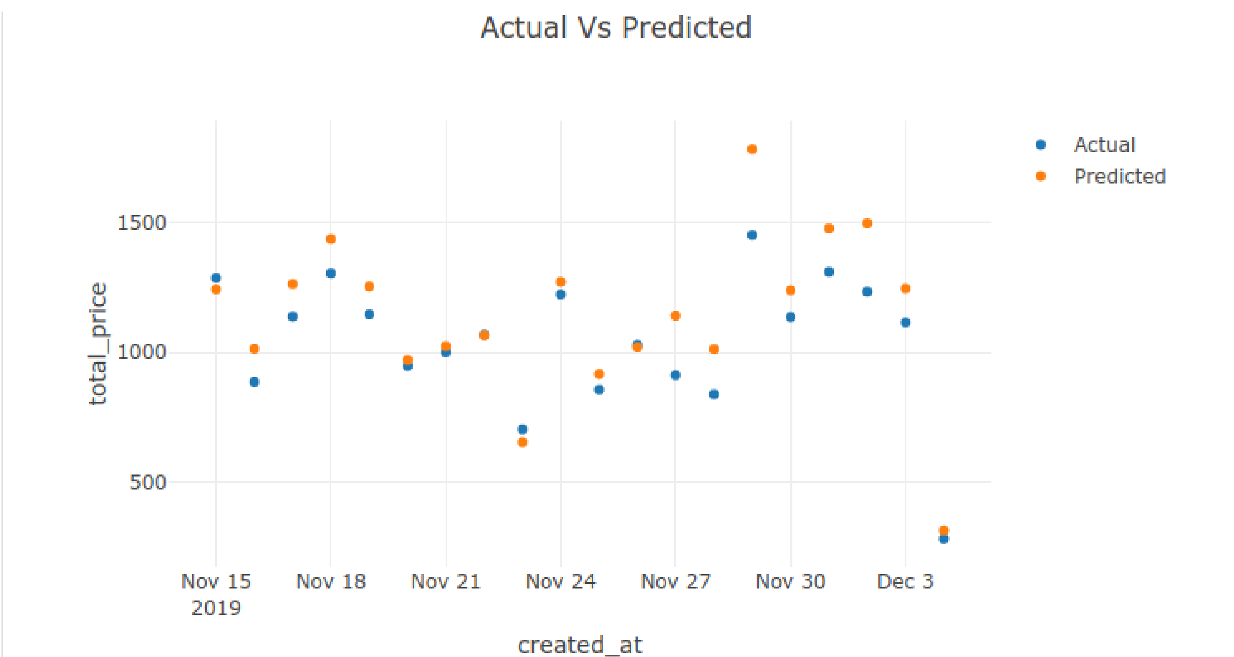
We can check a lot more models and play around with different hyper-parameters.

We also get some useful graphs which are presented in the following slide.

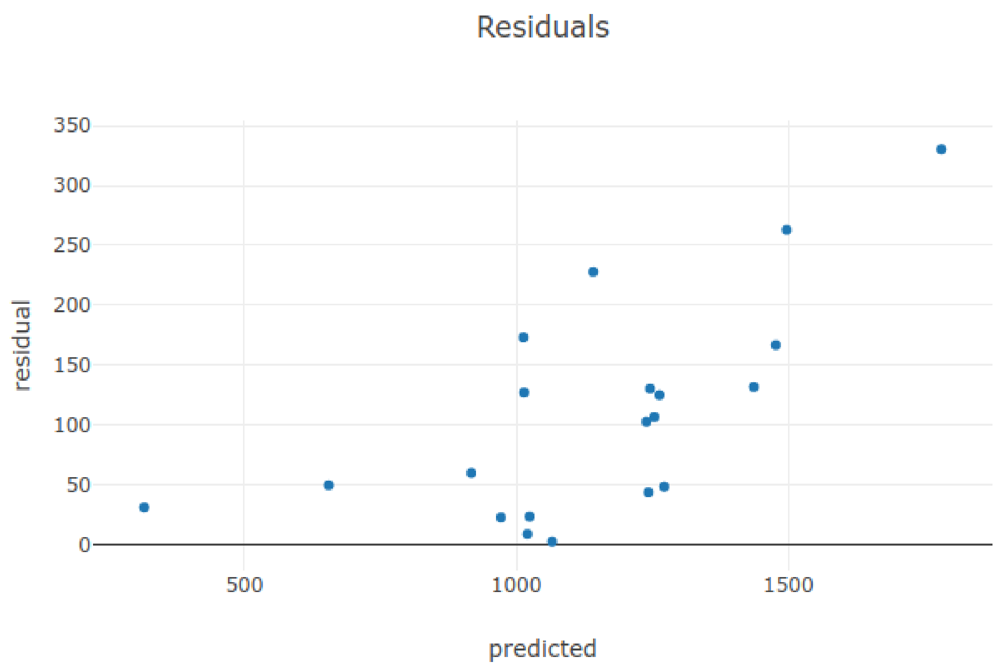
- **Feature Importance:**



- **Actual Vs Predicted:**

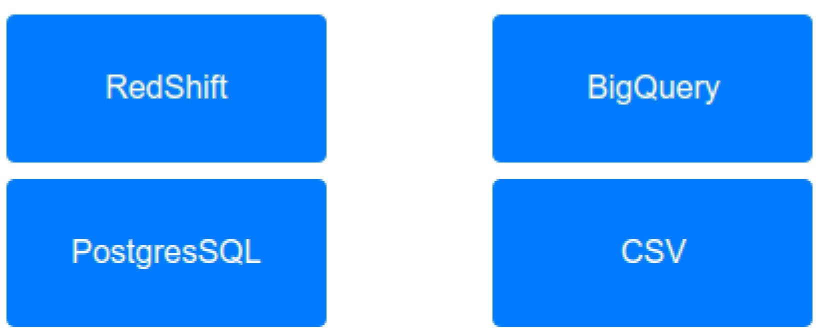


- # Residual Vs Predicted:



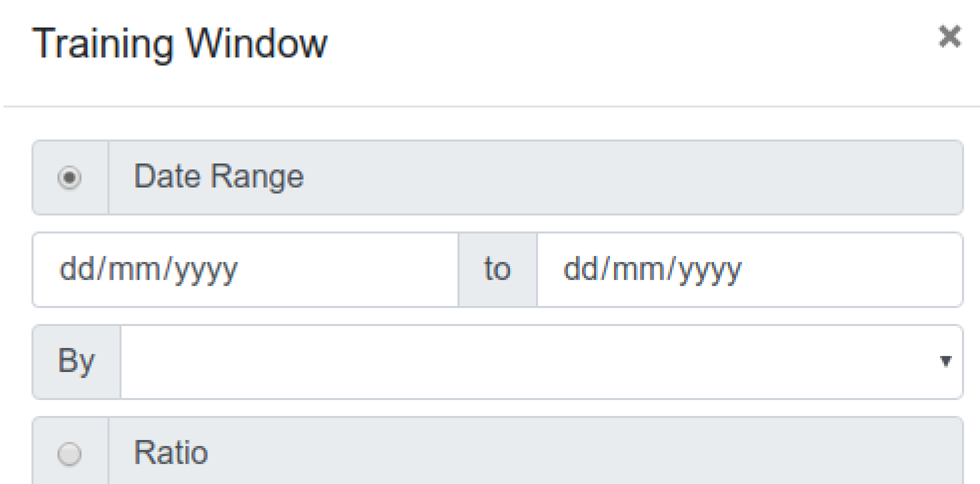
Epilogue: Way Ahead ?

- The Tool presented above ticks a lot of boxes, it isn't perfect !
- I'll discuss some shortcomings of the tool, and what progress has been made to remove them.
- **Database Sources:**
  In this version we could only use two sources: Redshift or BigQuery.
  That too, when the credentials were hard-coded in the script.
  Newer version presents much more flexibility:

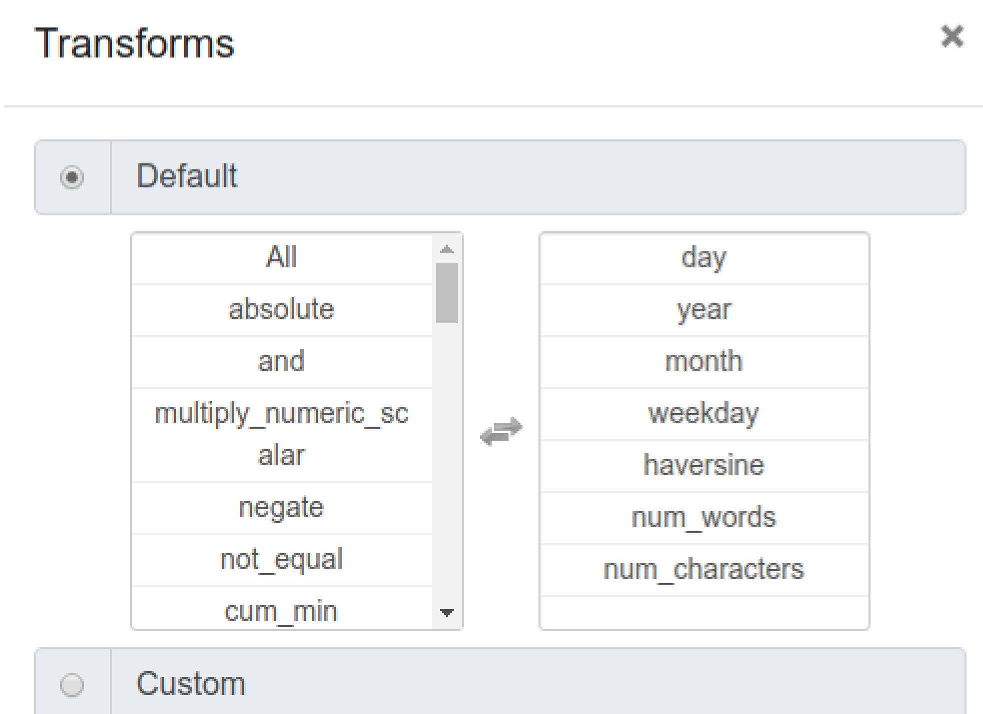


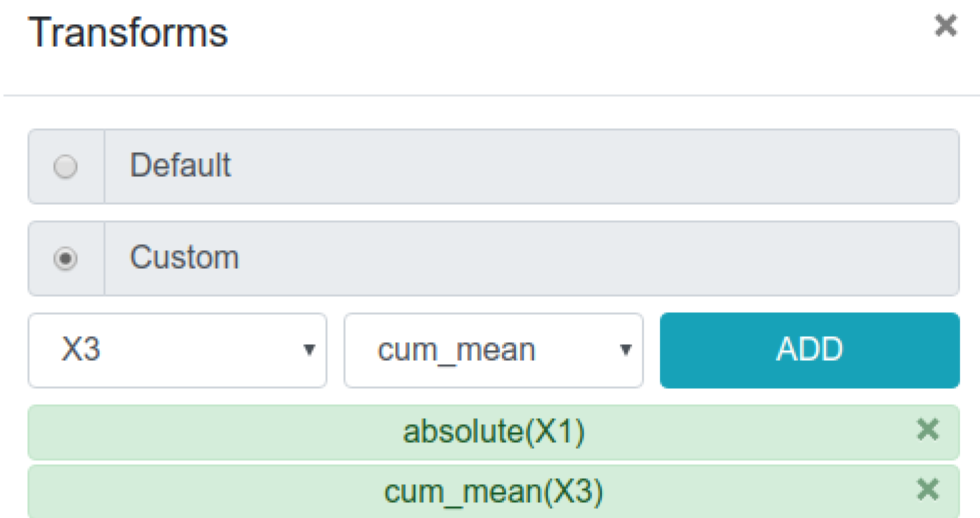
As you can see, more database options, including csv's.

- **Training Window**
  In the version presented above, we could only pass a single date as the training window.
  But what if the client wants to pass both the upper and the lower limit, or doesn't want to pass any date at all ?
  Newer version takes care of that:



- **Feature Engineering:**
  While implementing `featuretools`, we used only the default values.
  As it turns out, `featuretools` can provide a lot more flexibility, where the user can choose the features to be generated.
  Newer version provides a lot more control over `featuretools`:





- Newer version is still a work in progress and will take some time.
  You can check out the progress, even download the tool here:
  https://github.com/rohitduggal21/auto-ml

END