project

March 11, 2019

1 Machine Learning Engineer Nanodegree

1.1 Capstone Project

Virtual Machine (VM) preparation time prediction

Sergey Sergeev
March 11th, 2019

1.2 Definition

1.2.1 Project Overview

I work for the software development company, and we pay high attention to the test automation. During 15 years of our product development and many implemented customer projects we accumulated a lot of autotests. Autotests check various parts of our product in different environments, e.g.

- Clusters of various size: 1, 2, 4, 8 virtual machines (VMs)
- RedHat Enterprise Linux (RHEL) version 6 or 7
- Different versions of Java (our primary programming language)
- etc...

Individual test's execution times vary from few minutes to several hours depending on complexity and size.

We built the private cloud (based on Openstack) for the continuous autotest execution and a simple Web interface to manage it. On the dedicated page software engineer may choose the list of tests to be executed, a number of virtual machines, versions of the 3rdparty software to be provisioned in the VMs, and so on.

The test request is then queued to be executed as soon as possible, according to actual cloud capacity and VM consumption.

As a result, engineer receives an email with the test report. Test reports are also stored in some shared directory for reference, comparison and further analysis needs.

1.2.2 Problem Statement

For the higher cloud capacity utilization and better user experience we would like to have a model capable to predict overall test execution time. This time is a sum of three:

- **Queue time (`t1`)**: time spent by an execution request in the queue waiting for free cloud capacity
- **VM preparation time (`t2`)**: time to create a cluster of VMs and provision it with the requested 3rdparty software
- **Actual test execution time (`t3`)**: time to execute selected tests on the prepared VM cluster till the final report

In this project, we would focus on `t2` estimation only, leaving `t1` and `t3` for future.
Here is the brief workflow to prepare the solution:

- Explore and clean up data
- Prepare benchmark model for final solution evaluation
- Remove useless features - e.g. success/failure indicator, number of attempts performed
- Transform original features
- Try various regression models provided by scikit-learn
- Tune the best model parameters with grid search
- Test final model on a testing set, compare with the benchmark model

1.2.3 Metrics

R^2 regression score (coefficient of determination) would be used as the main evaluation metric.

Additionally, MAE (mean absolute error) and RMSE (root mean squared error) would be used for illustrative purposes.

1.3 Analysis

1.3.1 Data Exploration

Let's load the dataset from csv file and explore it:

```
The original dataset has 1095 data points with 32 variables each.
```

```
Out[3]: cluster_name                object
        attempts                     int64
        start_ts                     int64
        end_ts                       int64
        failed                        bool
        completed                     bool
        vm_count                     int64
        build_name                  object
        with_conda                    bool
        with_conda_version          object
        with_docker                   bool
        with_docker_version         object
        with_flavor                 object
        with_foundation               bool
        with_foundation_version     object
        with_gemfire                  bool
```

```
with_gemfire_version       object
with_image                 object
with_java_version          object
with_kubernetes              bool
with_kubernetes_version    object
with_memcached               bool
with_memcached_version     object
with_oracle                  bool
with_oracle_version        object
with_os_version            object
with_postgresql              bool
with_postgresql_version    object
with_tibco                   bool
with_tibco_version         object
with_ulticom                 bool
with_ulticom_version       object
dtype: object
```

Brief description of CSV columns:

- `cluster_name`: The name of a cluster, primary key of the dataset
- `attempts`: Number of attempts performed to create and provision requested cluster, can be from `1` to `10`. The procedure gives up if a cluster is still failed after 10 subsequent attempts to build it
- `start_ts`: Start timestamp (in milliseconds from 1970, Jan, 1, 00:00:00 UTC)
- `end_ts`: End timestamp (in milliseconds from 1970, Jan, 1, 00:00:00 UTC)
- `failed`: Was it finished successfully or not?
- `completed`: Was it completed (even with failure) or interrupted?
- `vm_count`: Cluster size. Number of requested virtual machines in a cluster, can be from `1` to `26`
- `build_name`: Some unique software version identifier in our build system

Other columns (theier names are started with `with_` prefix) describe how each of a cluster machine is to be provisioned:

- `with_flavor`: Size of VM (in terms of # of CPU, RAM and HDD size)
- `with_os_version`: Operating system version to be used, can be `RHEL6.9`, `RHEL7.3` or `RHEL7.4`
- `with_image`: Custom base VM image to be used, or `_default_` in case of some pre-provisioned RHEL image is sufficient
- `with_java_version`: Java version. For example, `1.8.0_74`, `1.8.0_102`

Other `with_` columns come in pairs:

- `with_<X>` (3rdparty indicator column): Should `X` software be installed and configured or not?
- `with_<X>_version` (3rdparty version column): If `X` should be installed, which version?

For example, [`with_docker = 'True'` and `with_docker_version='18.03.1.ce'`] means Docker 18.03.1.ce should be installed.

Note that the dataset also contains records such as [`with_docker = 'False'` and `with_docker_version='18.03.1.ce'`] - this means that no Docker is needed (version column is ignored in this case).

Few data examples are below:

```
Out[5]:             cluster_name  attempts       start_ts         end_ts  failed  \
        0  kalexey.20181227174135         1  1545922452338  1545923103007   False
        1  sdmitry.20181128184423         1  1543421316335  1543421667461   False
        2   igarus.20181221190350         2  1545408262310  1545409537436   False
        3  sdmitry.20181211190002         1  1544544132560  1544544935177   False
        4  abondar.20181207141530         1  1544181372440  1544181831985   False

           completed  vm_count                      build_name  with_conda  \
        0       True        13  20181227_174049_Proj_2127259257       False
        1       True         1  20181128_183651_Proj_1735463230       False
        2       True        14     20181220_115814_Proj_2696985       False
        3       True         9     20181211_161447_Proj_2631643       False
        4       True         4  20181207_083303_Proj_1027219150       False

          with_conda_version           ...           with_memcached_version  \
        0             4.5.11           ...                        _default_
        1             4.5.11           ...                        _default_
        2             4.5.11           ...                        _default_
        3             4.5.11           ...                        _default_
        4             4.5.11           ...                        _default_

          with_oracle with_oracle_version  with_os_version with_postgresql  \
        0       False              11.2.0          rhel7.3           False
        1        True              11.2.0          rhel7.4           False
        2       False              11.2.0          rhel7.4           False
        3        True              11.2.0          rhel7.4           False
        4        True              11.2.0          rhel7.3           False

           with_postgresql_version with_tibco with_tibco_version with_ulticom  \
        0                      9.5      False              8.4.5        False
        1                      9.5      False              8.4.5        False
        2                      9.5      False              8.4.5         True
        3                      9.5      False              8.4.5        False
        4                      9.5      False              8.4.5        False

           with_ulticom_version
        0             _default_
        1             _default_
        2                  9s65
        3             _default_
        4             _default_
```

```
[5 rows x 32 columns]
```

The **target variable** (time in seconds to create a cluster) `create_time` is defined as a difference between end and start timestamps.

Let's explore statistics over **numerical** columns:

Out[7]:

	attempts	start_ts	end_ts	vm_count	create_time
count	1095.000000	1.095000e+03	1.095000e+03	1095.000000	1095.000000
mean	1.296804	1.546146e+12	1.546147e+12	5.042009	593.556932
std	1.198162	1.907391e+09	1.907442e+09	5.017708	372.183086
min	1.000000	1.543070e+12	1.543071e+12	1.000000	50.761000
25%	1.000000	1.544439e+12	1.544439e+12	1.000000	353.266000
50%	1.000000	1.545925e+12	1.545926e+12	3.000000	499.470000
75%	1.000000	1.548057e+12	1.548057e+12	9.000000	729.423000
max	10.000000	1.549356e+12	1.549356e+12	26.000000	4539.504000

Numerical feature statistics conclusions:

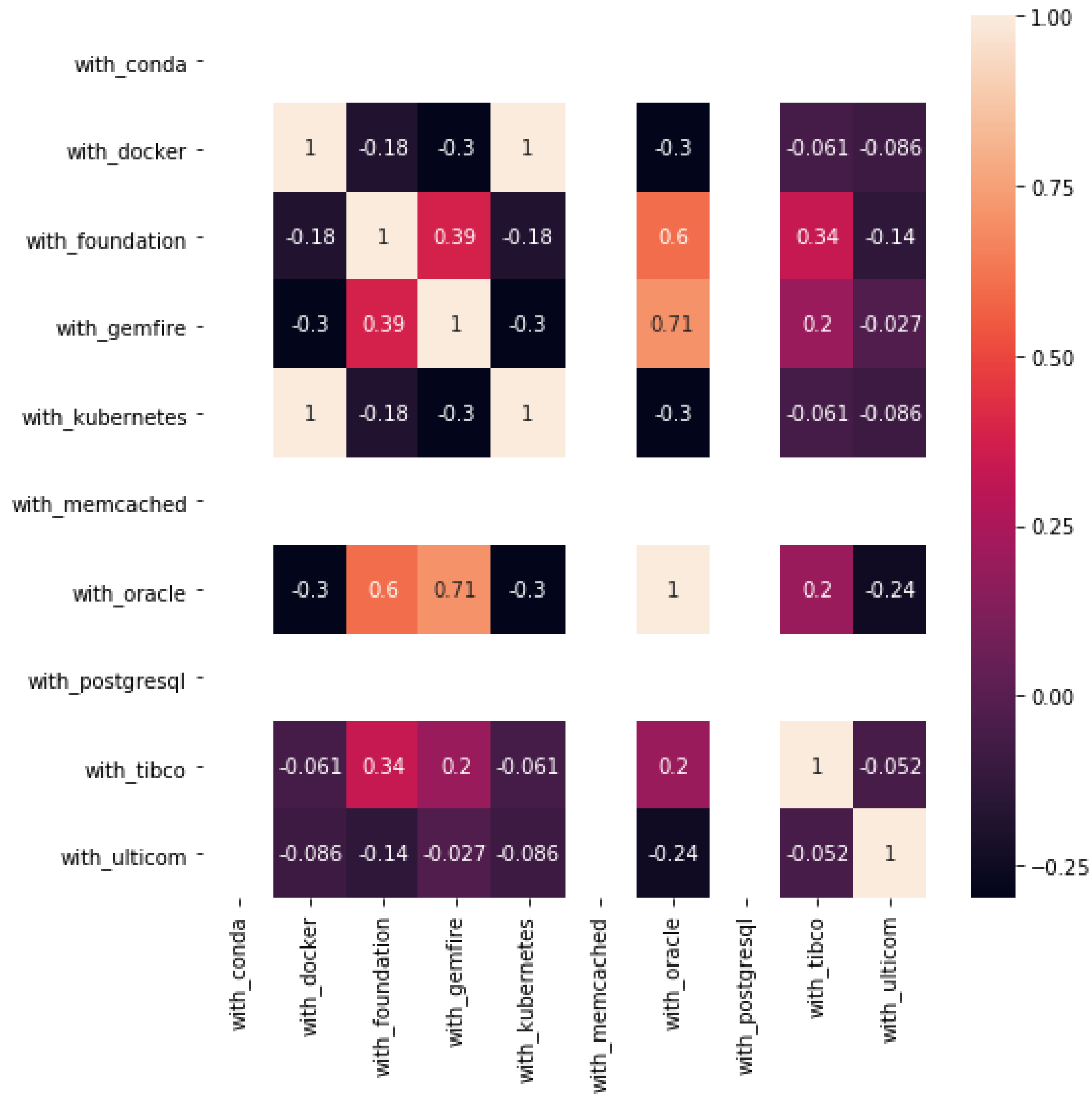
- Most of the cluster requests were completed in 1 attempt
- Cluster size (`vm_count`) averages are low (`median = 3` and `mean = 5`) but the tail is heavy
- There are outliers in the dataset (see max `attempts` and max `create_time`)

Now, let's look at the **boolean and categorical** columns:

	unique values
boolean feature	
failed	2
completed	2
with_conda	1
with_docker	2
with_foundation	2
with_gemfire	2
with_kubernetes	2
with_memcached	1
with_oracle	2
with_postgresql	1
with_tibco	2
with_ulticom	2

As one can see, not all possible software indicators are used in the dataset.

Let's look at the software indicator correlations:



Conclusions about software indicator features:

- Optional software `conda`, `memcached` and `postgresql` were not used in the dataset
- `docker` and `kebernetes` were used together always as the correlation is equal to 1.0
- There is a good correlation of `oracle` software with `gemfire` (`0.71`), and `oracle` with `foundation` (`0.6`)

Now let's explore other categorical features:

```
                     unique values
categorical feature
cluster_name                  1095
build_name                     488
with_flavor                      1
```

```
with_image                       1
with_java_version                5
with_os_version                  3
```

Conclusions:

- `with_flavor` and `with_image` are always the same, so they are useless

```
                         unique values
optional software
with_conda_version                   1
with_docker_version                  1
with_foundation_version              1
with_gemfire_version                 5
with_kubernetes_version              1
with_memcached_version               2
with_oracle_version                  1
with_postgresql_version              1
with_tibco_version                   1
with_ulticom_version                 3
```

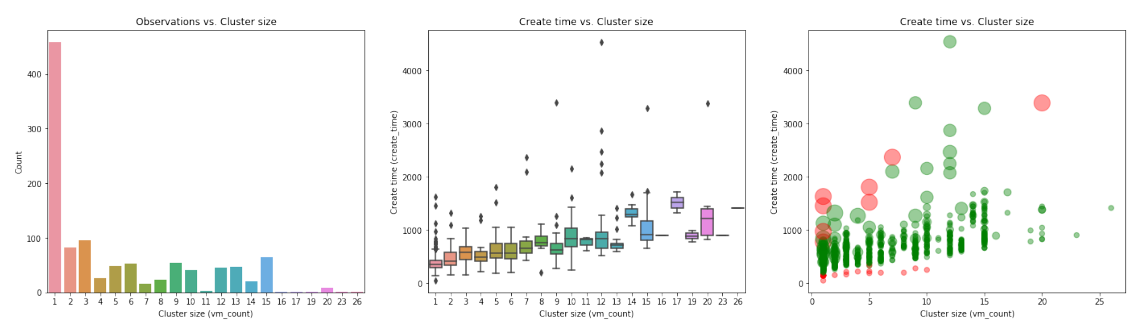
Conclusions above software versions:

- Most of the software versions don't vary, except of `gemfire` and `ulticom`.
- Two versions of `memcached` looks like an error in the dataset. Anyway, `memcached` is not used as we've seen above, so can be ignored.

1.3.2 Exploratory Visualization

We may expect that cluster size (`vm_count`) is the most important feature to predict cluster creation time (`create_time`).

Let's plot the dataset in these two dimensions.



Notes on the third plot:

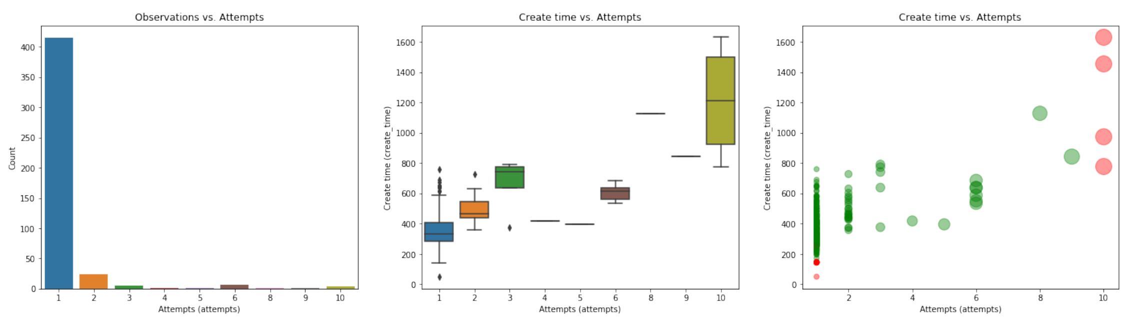
- Markers are red for `failed = True` and green for `failed = False`

- Marker size is proportional to the number of `attempts`

Conclusions from the plots above:

- About a half of the observations are for single VM clusters (`vm_count == 1`)
- We have almost no observations for large clusters (`vm_count > 15`), e.g. for `vm_count in [18, 21, 22, 24, 25]` we have no observations at all
- There are outliers for almost every cluster size
- All data for `vm_count == 14` looks like outliers if we look to the neighbours (`vm_count == 13` and `vm_count == 15`)
- Requests that were either `failed` or completed after a high number of `attempts` are mostly responsible for outliers
- Red markers at the bottom of the third plot are dataset errors. That's because cluster provisioning scripts are supposed to mark a request as failed after 10 unsuccesful attempts only. Looks like we have discovered a bug either in the scripts or in scripts logging.

Now let's zoom in data for `vm_count == 1` - half of the dataset.
In contrast to plots above we'll now split `create_time` over the number of `attempts`:



Conclusions from the plots above:

- Almost half of the original observations (~430 out of 1000) are single VM cluster requests (`vm_count == 1`) completed successfully in a single attempt (`attempts == 1`)
- There are still outliers even for `attempts == 1` (see the plot in the middle)

1.3.3 Algorithms and Techniques

Few scikit-learn models (available out of the box) would be checked below:

- Linear regression
- Decision tree
- Ridge
- RANSAC, Huber and Theil-Sen regressions - promised to be good in case of data with many outliers according the article

As well as some ensemble models:

- Random forest
- Gradient boosting

1.3.4 Benchmark

As a benchmark, we would use a simple model which doesn't take into account anything but cluster size (ignoring features about 3rdparty software to be provisioned).

The model should calculate expected `create_time` as an average over historical data for clusters of the same size (number of VMs).

1.4 Methodology

1.4.1 Data Preprocessing

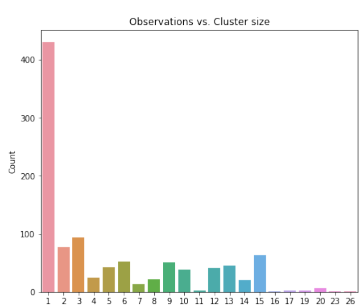
The first step is to remove observations that are either `failed` or completed after many `attempts`.

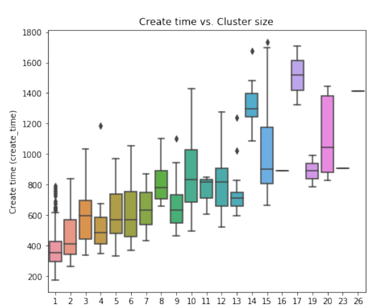
Multiple attempts are usually caused by some intermittent issues in the underlying OpenStack cloud that we can't predict).

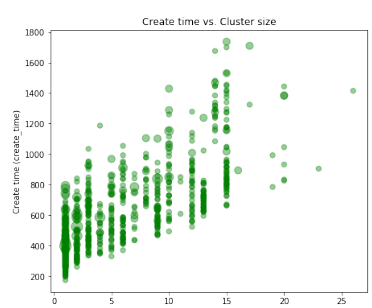
```
Found 60 bad data points, 5.48% of the whole dataset
```

	cluster_name	attempts	failed	vm_count	create_time
6	maximo.20181130172742	1	True	4	220.334
25	sdmitry.20190114185031	6	False	12	2076.067
29	marsels.20181128142358	1	True	5	225.501
35	marsels.20181204093910	10	True	1	1455.075
44	sdmitry.20190114185047	6	False	12	2872.506
70	edrojdina.20181206140532	1	True	1	144.247
71	nikolayk.20181225093155	10	True	1	1631.033
88	abondar.20190204124856	6	False	10	1614.971
135	marsels.20181128202554	10	True	5	1521.896
143	maximo.20190109104914	8	False	1	1129.215

```
We have 1035 good data points now, 94.52% of the original dataset
```







Now let's prepare features.
Features from original dataset to be **used**:

- `vm_count`

- with_os_version: Operating system version (need to be one-hot encoded further)
- with_<X>: Optional software indicators

Features from original dataset to be **ignored**:

- with_flavor and with_image: They are always the same in the dataset according to data analysis
- cluster_name: Just a primary key, no useful information
- build_name: Some unique version identifier, no useful information
- with_<X>_version: Optional software versions. Software installation procedures are almost the same for different sofware version, so we don't expect any significant change of the target variable because of version differene

These features are **ignored** as well as they are actually the **outcome** of cluster provisioning, not something we know before placing a cluster request:

- attempts
- start_ts
- end_ts
- failed
- completed

Full list of features:

```
['vm_count',
 'with_os_version',
 'with_conda',
 'with_docker',
 'with_foundation',
 'with_gemfire',
 'with_kubernetes',
 'with_memcached',
 'with_oracle',
 'with_postgresql',
 'with_tibco',
 'with_ulticom']
```

Example data with one-hot encoded with_os_version column:

```
   vm_count  with_conda  with_docker  with_foundation  with_gemfire  \
0        13       False        False            False         False
1         1       False        False            False          True
2        14       False        False            False          True
3         9       False        False             True          True
4         4       False        False            False          True

   with_kubernetes  with_memcached  with_oracle  with_postgresql  with_tibco  \
0            False           False        False            False       False
```

```
1           False           False         True             False       False
2           False           False        False             False       False
3           False           False         True             False       False
4           False           False         True             False       False

   with_ulticom  with_os_version_rhel6.9  with_os_version_rhel7.3  \
0        False                        0                        1
1        False                        0                        0
2         True                        0                        0
3        False                        0                        0
4        False                        0                        1

   with_os_version_rhel7.4
0                        0
1                        1
2                        1
3                        1
4                        0
```

1.4.2 Implementation

Let's prepare the benchmark model and obtain benchmark metrics.

Name	R^2	MAE	RMSE
Benchmark (mean)	0.7182	116.4822	152.1524
Benchmark (median)	0.6810	118.1614	161.8828

R-squared score of 0.7182 is taken as the benchmark.

Let's check several models available in scikit-learn with their parameters set to defaults:

Name	R^2	MAE	RMSE
Decision Tree	0.8348	87.6430	116.4958
Linear	0.8395	85.1727	114.8359
Ridge	0.8394	85.1669	114.8499
RANSAC	0.8201	85.8679	121.5720
Huber	0.8293	85.6247	118.3986
Theil-Sen	0.7953	89.3923	129.6756
Random Forest	0.8414	85.6550	114.1404
Gradient Boosting	0.8478	83.8335	111.8135

All the selected models have obtained the R-squared score higher than benchmark on the test dataset. Among them **Gradient Boosting** has obtained the highest one. MAE (mean absolute error) and RMSE (root mean squared error) are smallest for Gradient Boosting as well.

1.4.3 Refinement

In this section, we would try two approaches to improve Gradient Boosting:

- Grid search (over some of the tunable parameters)
- Transforming target: according to the scikit-learn article, this approach can improve regressors' scores in some cases.

Grid search results:

```
Fitting 10 folds for each of 180 candidates, totalling 1800 fits


[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:   25.6s
[Parallel(n_jobs=-1)]: Done 242 tasks      | elapsed:   33.7s
[Parallel(n_jobs=-1)]: Done 778 tasks      | elapsed:   58.8s
[Parallel(n_jobs=-1)]: Done 1478 tasks      | elapsed:  1.5min
[Parallel(n_jobs=-1)]: Done 1800 out of 1800 | elapsed:  1.8min finished
```

```
Out[25]: GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.3, loss='ls', max_depth=2, max_features=None,
                          max_leaf_nodes=None, min_impurity_decrease=0.0,
                          min_impurity_split=None, min_samples_leaf=1,
                          min_samples_split=2, min_weight_fraction_leaf=0.0,
                          n_estimators=50, n_iter_no_change=None, presort='auto',
                          random_state=None, subsample=0.9, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

```
                              R^2      MAE      RMSE
Name
Optimized Gradient Boosting  0.8562  81.3878  108.6777
```

Optimized Gradient Boosting obtained R-squared score (**0.8562**) that is just slightly different than Gradient Boosting trained with default parameter values (**0.8483**). Looks like GB's default parameters are already good enough for our dataset.

Now, let's try to transform the target variable with `log()` function before applying Gradient Boosting:

```
                     R^2      MAE      RMSE
Name
Transformed Target  0.8486  82.0337  111.5373
```

Now the model obtained slightly higher R-squared score: **0.8486** (as well as lower `MAE` and `RMSE`), but it's not different significantly.

If we recall that `MAE` is measured in seconds, the difference is just ~1.7 seconds between two models. It's not that important from end user point of view.

So after refinement attempts, let's stick to the basic **Gradient Boosting** with default parameter values.

1.5 Results

1.5.1 Model Evaluation and Validation

Let's evaluate dataset feature importances:

Out[28]:

feature	importance
vm_count	0.743361
with_ulticom	0.093252
with_docker	0.068163
with_kubernetes	0.041502
with_gemfire	0.016217
with_foundation	0.014172
with_tibco	0.010548
with_os_version_rhel7.4	0.005574
with_os_version_rhel7.3	0.003646
with_oracle	0.003300

As expected, **`vm_count` is by far the most important feature** to predict target variable.

Let's limit train and test datasets to have less variation of `vm_count` and check Gradient Boosting scores on these limited datasets:

- `vm_count <= 1`
- `vm_count <= 2`
- ... and so on

Out[29]:

Name	R^2	MAE	RMSE
Gradient Boosting: vm_count <= 1	0.1922	73.2785	102.3410
Gradient Boosting: vm_count <= 2	0.2661	72.6078	99.4224
Gradient Boosting: vm_count <= 3	0.4936	81.3403	110.6632
Gradient Boosting: vm_count <= 4	0.4897	80.8451	109.6870
Gradient Boosting: vm_count <= 5	0.5227	79.3627	107.5129
Gradient Boosting: vm_count <= 6	0.5421	80.1081	108.0107
Gradient Boosting: vm_count <= 7	0.5539	79.2495	106.6166
Gradient Boosting: vm_count <= 8	0.6215	80.1723	107.6306
Gradient Boosting: vm_count <= 9	0.6440	81.6106	108.5865
Gradient Boosting: vm_count <= 10	0.6972	83.1530	110.3933
Gradient Boosting: vm_count <= 11	0.6961	83.3636	110.5903
Gradient Boosting: vm_count <= 12	0.7068	82.8967	109.6758
Gradient Boosting: vm_count <= 13	0.7103	82.6586	109.1506
Gradient Boosting: vm_count <= 14	0.8021	83.8723	111.7721
Gradient Boosting: vm_count <= 15	0.8414	83.8869	112.0002
Gradient Boosting: full dataset	0.8480	83.7809	111.7398

As expected, less variation of `vm_count` in the train dataset produces less accurate model, meaning lower ability to capture target variable variance. R-squared score increases with `vm_count` variation.

However, if we look at `MAE`, it's value doesn't change a lot (in interval from 72 to 83 seconds).

1.5.2 Justification

Let's compare the final model with the benchmark:

```
                         R^2        MAE       RMSE
Name
Benchmark (mean)    0.7182  116.4822  152.1524
Gradient Boosting  0.8478   83.8267  111.8036
```

According to the results above, **Gradient Boosting** model explains test dataset variance better and produces better predictions than simple benchmark model describe above.

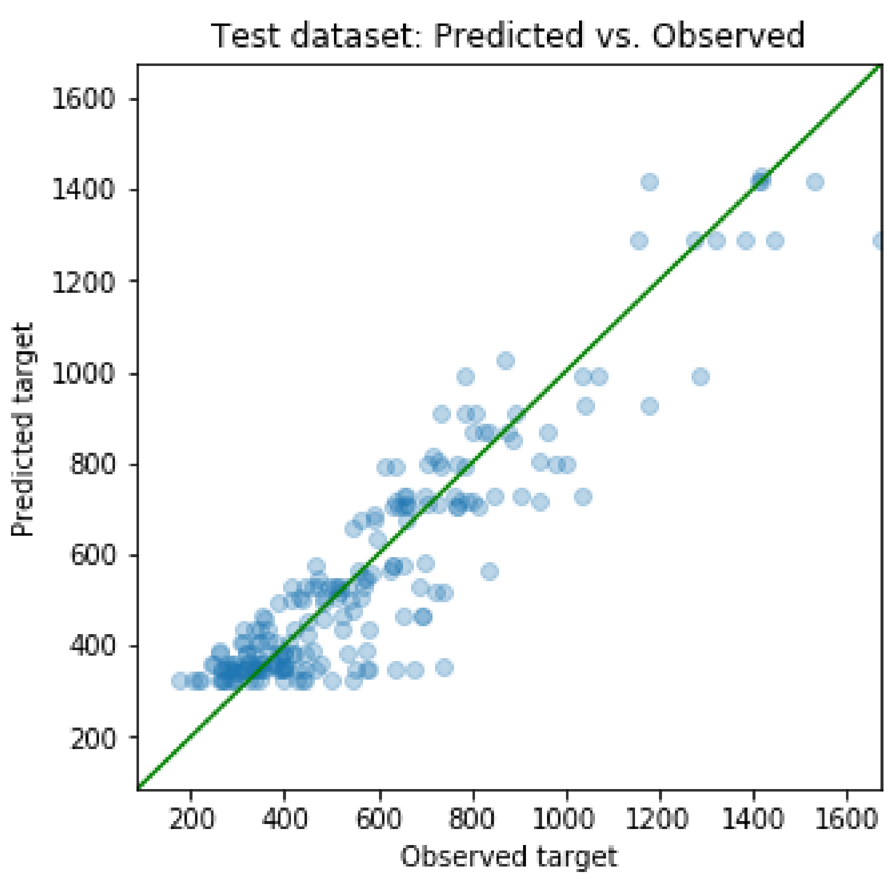
So, this model can be taken as a first solution for the explored problem.

Approaches to build better models are discussed below.

1.6 Conclusion

1.6.1 Free-Form Visualization

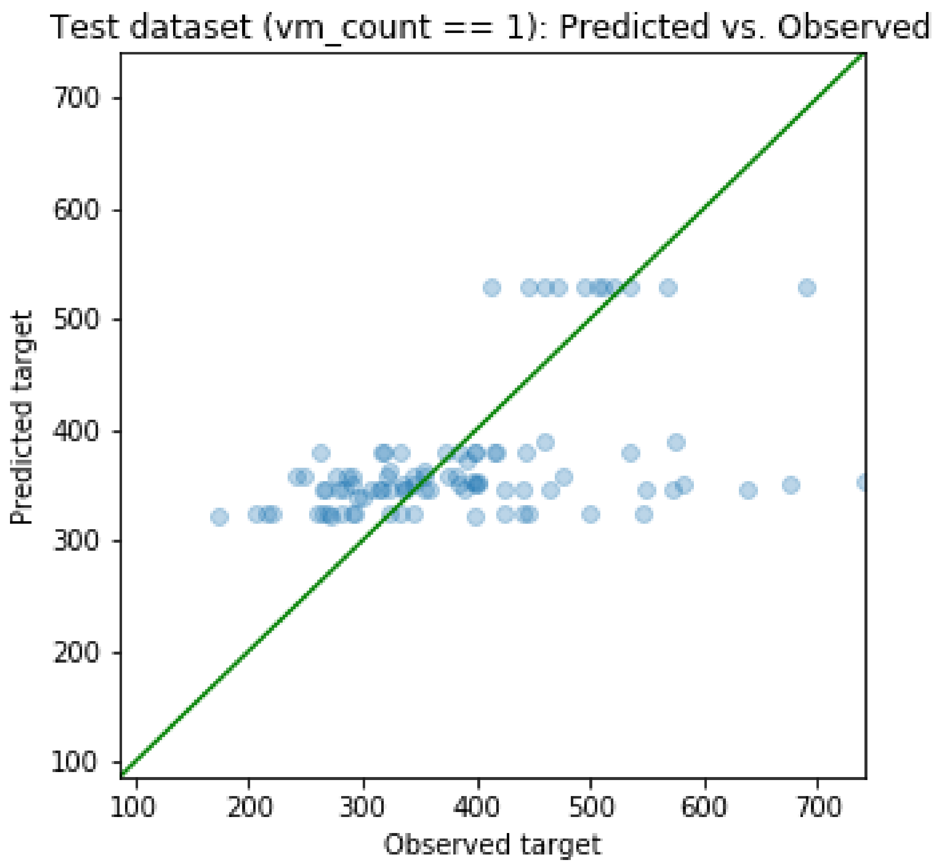
In this section, we'll explore our model predictions in comparison to ground truth test values:



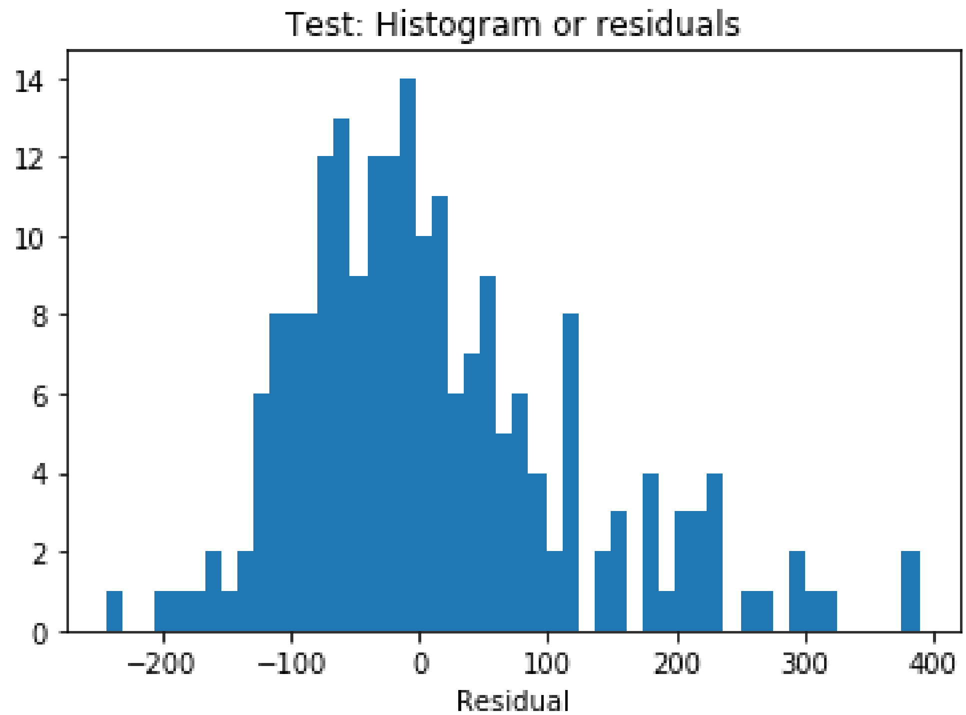
Green line on the plot above is where the observed target equals to the predicted value. Markers with positive residuals are plotted to the right of green line, negative residuals - to the left.

As one can see, markers on the left are more "condensed", while markers on the right have some outliers.

Let's filter our test data by `vm_count == 1` and plot it again for a closer look:



Additionally, let's plot histogram and collect basic statistics about residuals obtained on the test dataset:




```
Out[33]: count     207.000000
         mean       11.199518
         std       111.498447
         min      -242.976831
         25%       -65.884104
         50%        -9.044797
         75%        61.011066
         max       388.816646
         Name: create_time, dtype: float64
```

The histogram above (as well as calculated median) shows that our model is biased towards negative residuals.

It means it would rather predict target value greater than actual observation. However, it's not that bad from end-user point of view if a cluster would actually get prepared earlier than predicted.

On the other hand, high positive residual outliers (right tail of histogram above) represent the opposite situation: end user would wait for his cluster longer than he was "promised" by the model. The difference could be up to ~6.5 minutes (`max = 388 seconds`). This attribute of the model is to be considered and addressed somehow when the model will be placed in production.

1.6.2 Reflection

Few things I've noticed in the course of this project:

1) The most important and interesting part is the data exploration: starting from initial dataset analysis up to feature importances and residual statistics. I've also noticed how important are visualization techniques, and I wish to improve them.

2) Data exploration helped to find an issue in the cluster creation scripts, so it can be useful not only for ML purposes.

3) Feature importances detected by the model are different from what I expected intuitively. Before starting this project, I thought that 3drparty software selection is more important than it actually is.

4) Now I think that more data is needed in order to improve the model further. Few examples of additional features that could be useful:

- current CPU/RAM/HDD load of underlying Openstack hypervisors
- VM distribution among hypervisors

Finally, let's get back to the high level problem described in the beginning:

> For the higher cloud capacity utilization and better user experience we would like to have a model capable to predict overall > test execution time. This time is a sum of three:
>
> - **Queue time (`t1`)**: time spent by an execution request in the queue waiting for free cloud capacity
> - **VM preparation time (`t2`)**: time to create a cluster of VMs and provision it with the requested 3rdparty software
> - **Actual test execution time (`t3`)**: time to execute selected tests on the prepared VM cluster till the final report
>
> In this project, we would focus on `t2` estimation only, leaving `t1` and `t3` for future.

It's time now to put the model in production, and continue with `t1` and `t3` prediction problems.

1.6.3 Improvement

I as noted in the section above, more features need to be collected to improve the model.

Another improvement is to train the model with "asymmetric" loss function: to penalize model more for predictions lower than actual observation.

This is because the problem we've explored is an ETA problem, so the residual sign is important from end-user point of the view: it's better to predict target value with negative residual.

The first approach to try could be `quantile` loss function with `alpha = .75` instead of `least squares` to train Gradient Boosting.

Besides the model improvement, we may use data exploration techniques to dig into cluster creation failures to find what caused them and how cluster creation scripts can be adjusted to avoid such failures or detect them earlier.