data-dashboard

Release 0.1.1

Maciej Dowgird

May 20, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

`data_dashboard` library allows you to build HTML Dashboard visualizing not only the data and relationships between features but also automatically search for the best 'baseline' sklearn compatible Model.

You can install `data_dashboard` with pip:

```
pip install data-dashboard
```

Note: Please keep in mind that package name is data-dashboard (with hyphen: '-') whereas module to import from is called data_dashboard (with underscore: '_').

To create a Dashboard you need the data: `X`, `y` and the `output_directory` where the HTML files will be placed. You can use toy datasets from *Examples* (e.g. `iris` dataset) included in the library as well:

```
from data_dashboard import Dashboard
from data_dashboard.examples import iris
output_directory = "your_path/dashboard_output"
X, y, descriptions = iris()  # descriptions is additional argument described further in␣
↪docs

dsh = Dashboard(X, y, output_directory, descriptions)
dsh.create_dashboard()
```

Note: Depending on the size of your data, fitting process might take some time. Please be patient!

Created HTML Dashboard will contain 3 subpages for you to investigate:

- `Overview` with summary statistics of the data;
- `Features` where you can dig deeper into each feature in the data;
- `Models` showing search results and models performances.

CHAPTER

TWO

EXAMPLE DASHBOARD

You can access an example Dashboard with Titanic dataset here: https://example-data-dashboard.herokuapp.com

2.1 Dashboard

2.1.1 Creating Dashboard instance

Features Descriptions Dictionary

`Dashboard` instance can be created with just `X`, `y` and `output_directory` arguments, but it doesn't mean it cannot be customized. The most notable optional argument that can augment the HTML Dashboard is `feature_descriptions_dict` dictionary-like object. Structure of the dict should be:

```
feature_descriptions_dict = {

    "Feature1": {

        "description": "description of feature1, e.g. height in cm",
        "category": "cat" OR "num",
        "mapping": {
            "value1": "better name or explanation for value1",
            "value2": "better name or explanation for value2"
        }
    },

    "Feature2": {

        (...)

    }

}
```

- `"description"` describes what the feature is about - what's the story behind numbers and values.
- `"category"` defines what kind of a data you want the feature to be treated as:
  - `"num"` stands for Numerical values;
  - `"cat"` is for Categorical variables;

- "mapping" represents what do every value in your data mean - e.g. 0 - "didn't buy a product", 1 - "bought a product".

This external information is fed to the Dashboard and it will be included in HTML files (where appropriate) for your convenience. Last but not least, by providing a specific "category" you are also forcing the Dashboard to interpret a given feature the way you want, e.g.: you could provide "num" to a binary variable (only 0 and 1s) and Dashboard will treat that feature as Numerical (which means that, for example, Normal Transformations will be applied).

Note: Please keep in mind that every argument in feature_descriptions_dict is optional: you can provide all Features or only one, only "category" for few of the Features and "description" for others, etc.

Other Dashboard instance arguments

already_transformed_columns can be a list of features that are already transformed and won't need additional transformations from the Dashboard:

```
Dashboard(X, y, output_directory,
          already_transformed_columns=["Feature1", "pre-transformed Feature4"]
          )
```

classification_pos_label forces the Dashboard to treat provided label as a positive (1) label (*classification* problem type):

```
Dashboard(X, y, output_directory,
          classification_pos_label="Not Survived"
          )
```

force_classification_pos_label_multiclass is a bool flag useful when you also provide classification_pos_label in a *multiclass* problem type (essentially turning it into *classification problem*) - without it, classification_pos_label will be ignored:

```
Dashboard(X, y, output_directory,
          classification_pos_label="Iris-Setosa",
          force_classification_pos_label_multiclass=True
          )
```

random_state can be provided for results reproducibility:

```
Dashboard(X, y, output_directory,
          random_state=13,
          )
```

Example

```
dsh = Dashboard(
    X=your_X,
    y=your_y,
    output_directory="path/output",
    features_descriptions_dict={"petal width (cm)": {"description": "width of petal (in␣
↪cm)"}},
    already_transformed_columns=["sepal length (cm)"],
```

(continues on next page)

(continued from previous page)

```
    classification_pos_label=1,
    force_classification_pos_label=True,
    random_state=10
    )
```

2.1.2 Creating HTML Dashboard

To create HTML Dashboard from `Dashboard` instance, you need to call `create_dashboard` method:

```
dsh.create_dashboard()
```

You can customize the process further by providing appropriate arguments to the method (see below).

Models

`models` stands for collection of sklearn Models that will be fit on provided data. They can be provided in different ways:

- `list` of Models instances;
- `dict` of Model class: param_grid attributes to do GridSearch on;
- `None` - in which case default Models will be used.

```
# list of Models
models = [DecisionTreeClassifier(), SVC(C=100.0), LogisticRegression()]

# dict for GridSearch
models = {
    DecisionTreeClassifier: {"max_depth": [1, 5, 10], "criterion": ["gini", "entropy"]},
    SVC: {"C": [10, 100, 1000]}
}

# None
models = None
```

Scoring

`scoring` should be a sklearn scoring function appropriate for a given problem type (e.g. `roc_auc_score` for *classification*). It can also be `None`, in which case default scoring for a given problem will be used:

```
scoring = precision_score
```

Note: Some functions might not work for some type of problems (e.g. `roc_auc_score` for *multiclass*)

Mode

`mode` should be provided as either `"quick"` or `"detailed"` string literal. Argument is useful only when `models=None`.

- if `"quick"`, then the initial search is done only on default instances of Models (for example `SVC()`, `LogisticRegression()`, etc.) as Models are simply scored with scoring function. Top scoring Models are then GridSearched;
- if `"detailed"`, then all available combinations of default Models are GridSearched.

Logging

`logging` is a `bool` flag indicating if you want to have .csv files (search logs) included in your output directory in logs subdirectory.

Disabling Pairplots

Both `seaborn` PairPlot in `Overview` subpage and ScatterPlot Grid in `Features` subpage were identified to be the biggest time/resource bottlenecks in creating HTML Dashboard. If you feel like speeding up the process, set `disable_pairplots=True`.

Note: Pairplots are disabled by default when the number of features in your data crosses certain threshold. See also *Forcing Pairplots*.

Forcing Pairplots

When number of features in X and y crosses a certain threshold, creation of both `seaborn` PairPlot and ScatterPlot Grid is disabled. This was a conscious decision, as not only it extremely slows down the process (and might even lead to raising Exceptions or running out of memory), PairPlots are getting so enormous that the insight gained from them is minuscule.

If you know what you're doing, set `force_pairplot=True`.

Note: If `disable_pairplots=True` and `force_pairplot=True` are both provided, `disable_pairplots` takes precedence and pairplots **will be disabled**.

Example

```
dsh.create_dashboard(
    models=None,
    scoring=sklearn.metrics.precision_score,
    mode="detailed",
    logging=True,
    disable_pairplots=False,
    force_pairplots=True
)
```

2.1.3 Setting Custom Preprocessors in Dashboard

`set_custom_transformers` is a method to provide your own Transformers to `Dashboard` pipeline. `Dashboard` preprocessing is simple, so you are free to change it to your liking. There are 3 arguments (all optional):

- *categorical_transformers*
- *numerical_transformers*
- *y_transformer*

Both `categorical_transformers` and `numerical_transformers` should be list-like objects of instantiated Transformers. As names suggest, `categorical_transformers` will be used to transform Categorical features, whereas `numerical_transformers` will transform Numerical features.

`y_transformer` should be a **single Transformer**.

```
dsh.set_custom_transformers(
    categorical_transformers=[SimpleImputer(strategy="most_frequent")],
    numerical_transformers=[StandardScaler()],
    y_transformer=LabelEncoder()
)
```

Note: Keep in mind that in *regression* problems, `Dashboard` already wraps the target in `TransformedTargetRegressor` object (with `QuantileTransformer` as a transformer). See also sklearn documentation.

2.1.4 Using Dashboard as sklearn pipeline

`Dashboard` can also be used as a simpler version of `sklearn.pipeline` - methods such as `transform`, `predict`, etc. are exposed and available. Please refer to *Documentation* for more information.

2.1.5 Documentation

class data_dashboard.dashboard.**Dashboard**(*X*, *y*, *output_directory*, *feature_descriptions_dict=None*, *already_transformed_columns=None*, *classification_pos_label=None*, *force_classification_pos_label_multiclass=False*, *random_state=None*)

Data Dashboard with Visualizations of original data, transformations and Machine Learning Models performance.

Dashboard analyzes provided data (summary statistics, correlations), transforms it and feeds it to Machine Learning algorithms to search for the best scoring Model. All steps are written down into the HTML output for end-user experience.

HTML output created is a set of 'static' HTML pages saved into provided output directory - there are no server - client interactions. Visualization are still interactive though through the use of Bokeh library.

Note: As files are static, data might be embedded in HTML files. Please be aware when sharing produced HTML Dashboard.

Dashboard object can also be used as a pipeline for transforming/fitting/predicting by using exposed methods loosely following sklearn API.

output_directory
directory where HTML Dashboard will be placed

Type str

already_transformed_columns
list of feature names that are already pre-transformed

Type list

random_state
integer for reproducibility on fitting and transformations, defaults to None if not provided during __init__

Type int, None

features_descriptions
FeatureDescriptor containing external information on features

Type FeatureDescriptor

features
Features object with basic features information

Type Features

analyzer
Analyzer object analyzing and performing calculations on features

Type Analyzer

transformer
Transformer object responsible for transforming the data for ML algorithms, fit to all data

Type Transformer

transformer_eval
Transformer object fit on train data only

Type Transformer

model_finder
ModelFinder object responsible for searching for Models and assessing their performance

Type ModelFinder

X
data to be analyzed

Type pandas.DataFrame, numpy.ndarray, scipy.csr_matrix

y
target variable

Type pandas.Series, numpy.ndarray

X_train
train split of X

Type pandas.DataFrame

X_test
test split of X

Type pandas.DataFrame

y_train
train split of y

Type pandas.Series

y_test
test split of y

Type pandas.Series

transformed_X
all X data transformed with transformer

Type numpy.ndarray, scipy.csr_matrix

transformed_y
all y data transformed with transformer

Type numpy.ndarray

transformed_X_train
X train split transformed with transformer_eval

Type numpy.ndarray, scipy.csr_matrix

transformed_X_test
X test split transformed with transformer_eval

Type numpy.ndarray, scipy.csr_matrix

transformed_y_train
y train split transformed with transformer_eval

Type numpy.ndarray

transformed_y_test
y test split transformed with transformer_eval

Type numpy.ndarray

__init__(*X*, *y*, *output_directory*, *feature_descriptions_dict=None*, *already_transformed_columns=None*, *classification_pos_label=None*, *force_classification_pos_label_multiclass=False*, *random_state=None*)
Create Dashboard object.

Provided X and y are checked and converted to pandas object for easier analysis and eventually split and transformed. classification_pos_label is checked if the label is present in y target variable. X is assessed for the number of features and appropriate flags are set. All necessary objects are created. Transformer and transformer_eval are fit to all and train data, appropriately.

X
data to be analyzed
Type pandas.DataFrame, numpy.ndarray, scipy.csr_matrix

y
target variable
Type pandas.Series, numpy.ndarray

output_directory
directory where HTML Dashboard will be placed
Type str

feature_descriptions_dict
dictionary of metadata on features in X and y, defaults to None

Type dict, None

already_transformed_columns
list of feature names that are already pre-transformed
Type list

classification_pos_label
value in target that will be used as positive label
Type Any

force_classification_pos_label_multiclass
flag indicating if provided classification_pos_label in multiclass problem should be forced, de facto changing the problem to classification
Type bool

random_state
integer for reproducibility on fitting and transformations, defaults to None
Type int, None

create_dashboard(*models=None*, *scoring=None*, *mode='quick'*, *logging=True*, *disable_pairplots=False*, *force_pairplot=False*)

Create several Views (Subpages) and join them together to form an interactive WebPage/Dashboard.

Models can be:

- list of initialized models
- dict of 'Model Class': param_grid of a given model to do the GridSearch on
- None - default Models collection will be used

scoring should be a sklearn scoring function. If None is provided, default scoring function will be used.

mode can be:

- **"quick": search is initially done on all models but with no parameter tuning after which top** Models are chosen and GridSearched with their param_grids
- "detailed": GridSearch is done on all default models and their params

Provided mode doesn't matter when models are explicitly provided (not None).

Note: Some functions might not work as of now: e.g. roc_auc_score for multiclass problem as it requires probabilities for every class in comparison to regular predictions expected from other scoring functions.

Depending on logging flag, .csv logs might be created or not in the output directory.

force_pairplot flag forces the dashboard to create Pairplot and ScatterPlot Grid when it was assessed in the beginning not to plot it (as number of features in the data exceeded the limit).

disable_pairplot flag disables creation of Pairplot and ScatterPlot Grid in the Dashboard - it takes precedence over force_pairplot flag.

HTML output is created in output_directory attribute file path and opened in a web browser window.

Parameters

- **models** (*list, dict, optional*) – list of Models or 'Model class': param_grid dict pairs, defaults to None
- **scoring** (*func, optional*) – sklearn scoring function, defaults to None

- **mode** (*"quick", "detailed", optional*) – either "quick" or "detailed" string, defaults to "quick"
- **logging** (*bool, optional*) – flag indicating if .csv logs should be created, defaults to True
- **disable_pairplots** (*bool, optional*) – flag indicating if Pairplot and ScatterPlot Grid in the Dashboard should be created or not, defaults to False
- **force_pairplot** (*bool, optional*) – flag indicating if PairPlot and ScatterPlot Grid in the Dashboard should be created when number of features in the data crossed the internal limit, defaults to False

search_and_fit(*models=None*, *scoring=None*, *mode='quick'*)

Search for the best scoring Model, fit it with all data and return it.

Models can be: - list of initialized models - dict of 'Model Class': param_grid of a given model to do the GridSearch on - None - default Models collection will be used

scoring should be a sklearn scoring function. If None is provided, default scoring function will be used.

mode can be:

- **"quick": search is initially done on all models but with no parameter tuning after which top** Models are chosen and GridSearched with their param_grids
- "detailed": GridSearch is done on all default models and their params

Provided mode doesn't matter when models are explicitly provided (not None).

Note: Some functions might not work as of now: e.g. roc_auc_score for multiclass problem as it requires probabilities for every class in comparison to regular predictions expected from other scoring functions.

Parameters

- **models** (*list, dict, optional*) – list of Models or 'Model class': param_grid dict pairs, defaults to None
- **scoring** (*func, optional*) – sklearn scoring function, defaults to None
- **mode** (*"quick", "detailed", optional*) – either "quick" or "detailed" string, defaults to "quick"

Returns best scoring Model already fit to X and y data

Return type sklearn.Model

set_and_fit(*model*)

Set provided Model as a best scoring Model and fit it to all X and y data.

Parameters **model** (*sklearn.Model*) – instance of ML Model

transform(*X*)

Transform provided X data with Transformer.

Returns transformed X

Return type numpy.ndarray, scipy.csr_matrix

predict(*transformed_X*)

Predict target from provided X with the best scoring Model.

Parameters **transformed_X** (*pandas.DataFrame, numpy.ndarray, scipy.csr_matrix*) – transformed X feature space to predict target variable from

Returns predicted y target variable

Return type numpy.ndarray

best_model()

Return best (chosen) Model used in predictions.

Returns best scoring Model

Return type sklearn.Model

set_custom_transformers(*categorical_transformers=None*, *numerical_transformers=None*, *y_transformer=None*)

Set custom Transformers to be used in the problem pipeline.

Provided arguments should be a list of Transformers to be used with given type of features. Only one type of transformers can be provided.

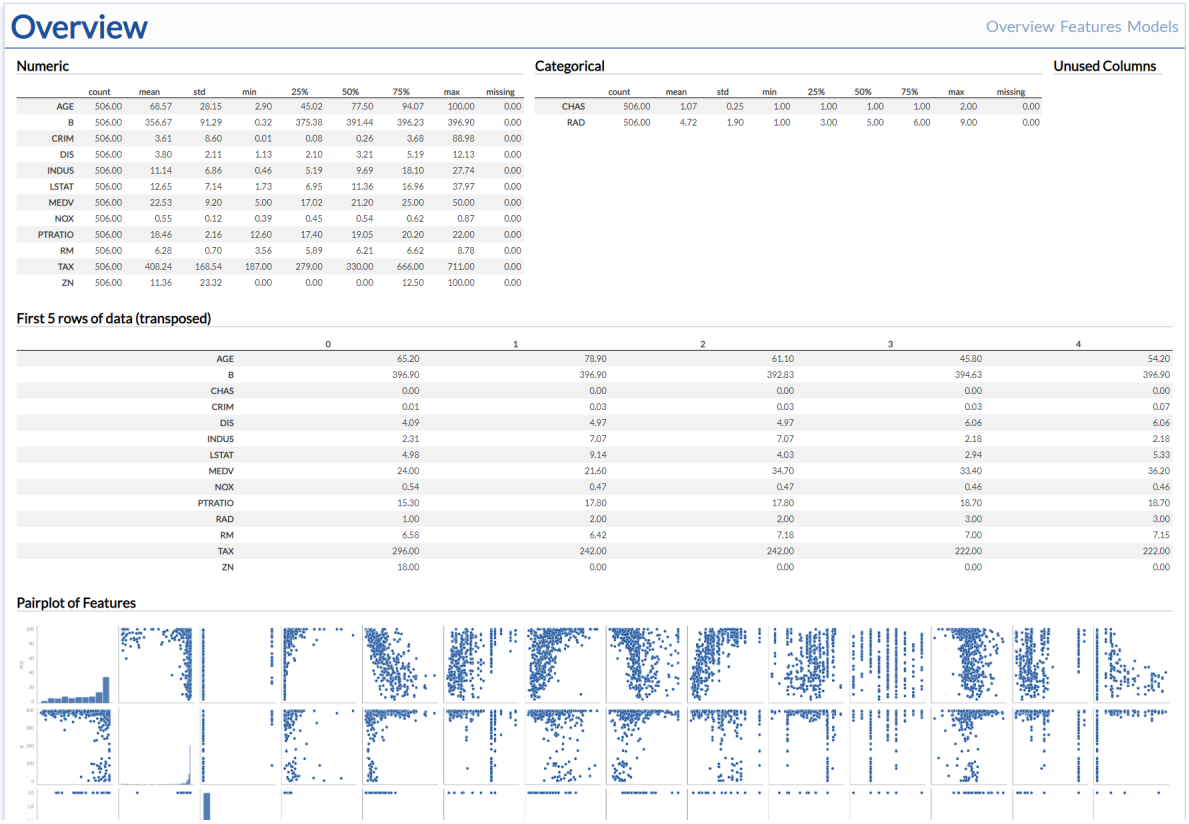
Transformers are updated in both transformer and transformer_eval instances. ModelFinder and Output instances are recreated with the new transformed data and new Transformers.

Parameters

- **categorical_transformers** (*list*) – list of Transformers to be used on categorical features
- **numerical_transformers** (*list*) – list of Transformers to be used on numerical features
- **y_transformer** (*sklearn.Transformer*) – singular Transformer to be used on target variable

2.2 HTML Subpages

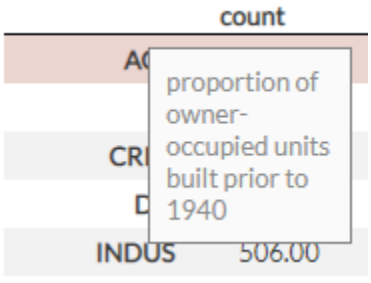
2.2.1 Overview



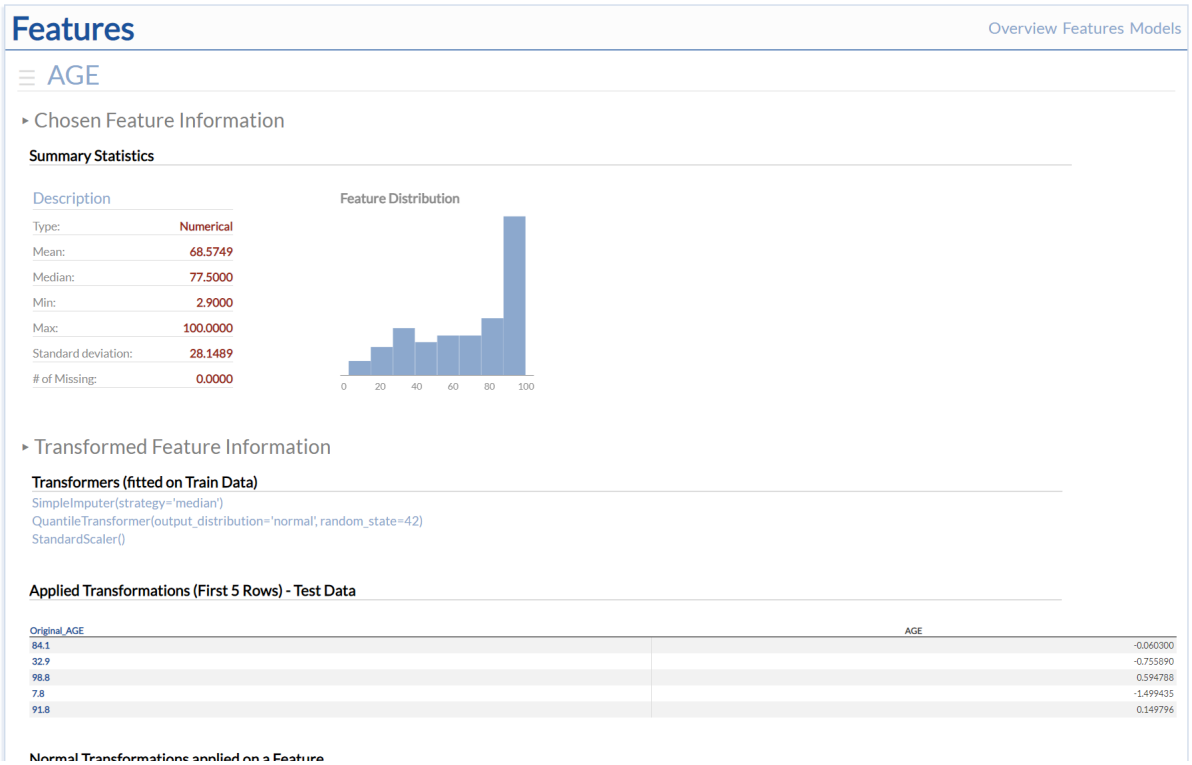
`Overview` subpage gives general information on the Features - their summary statistics, number of missing values, etc. (similar to `describe` method from `pandas.DataFrame`).

If the number of features isn't too big, seaborn pairplot is also included.

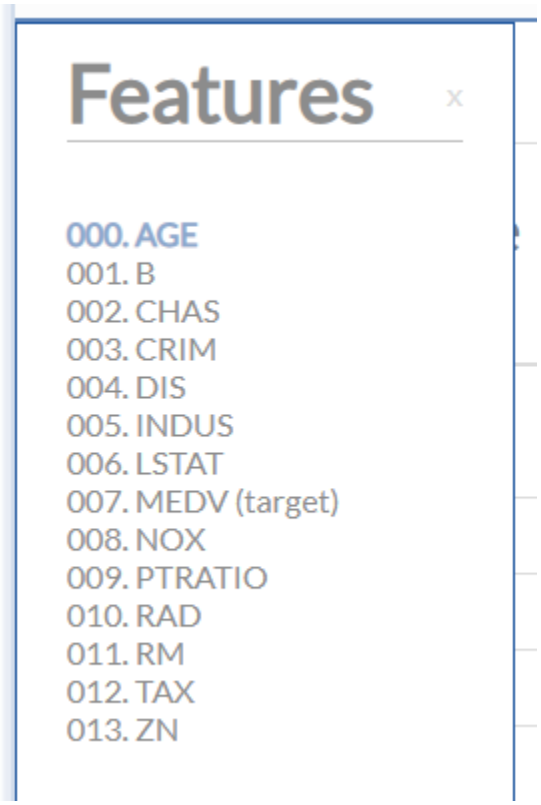
Note: Feature names in all tables are 'hoverable' - upon hovering, corresponding description and mapping will be shown.



2.2.2 Features View



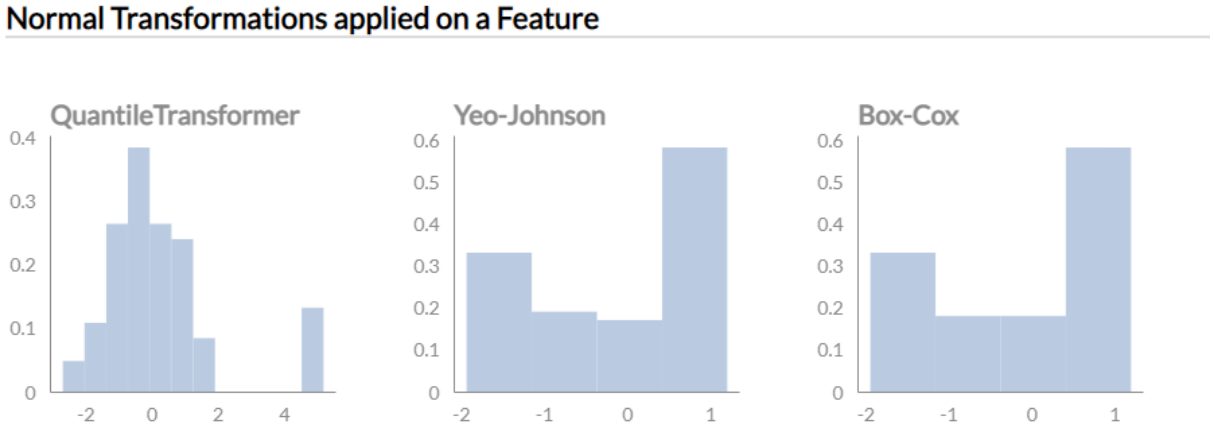
`Features` subpage allows you to dive deeper into each Feature present in your data. Not only summary statistics and underlying distribution are included, but also transformations and correlations with other features - feature engineering might be a little bit easier. To change displayed feature, you can click on the 'burger' button in the upper left corner and choose another one from the opened menu.



Note: Depending on the number of features, refreshing the page upon feature selection change might take a while - please be patient!

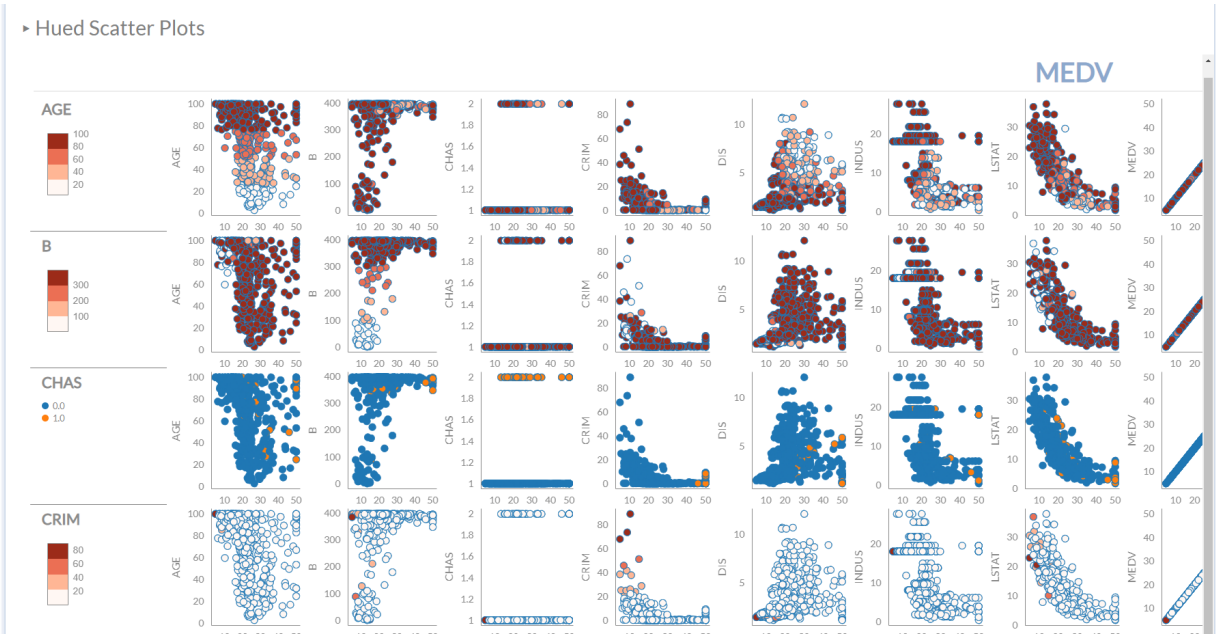
Content in `Features` subpage is divided into subsections that you can easily expand or hide (if the page gets too cluttered). First two sections provide information on a chosen feature (both original and transformed), whereas the third section shows correlations between **all** features (normalized and original), regardless of selection.

Note: When Numerical Feature is selected, Transformations sections will show plots of Normal Transformations. This should help you make a conscious decision which of the Transformers is the best for that feature (per sklearn guidance).

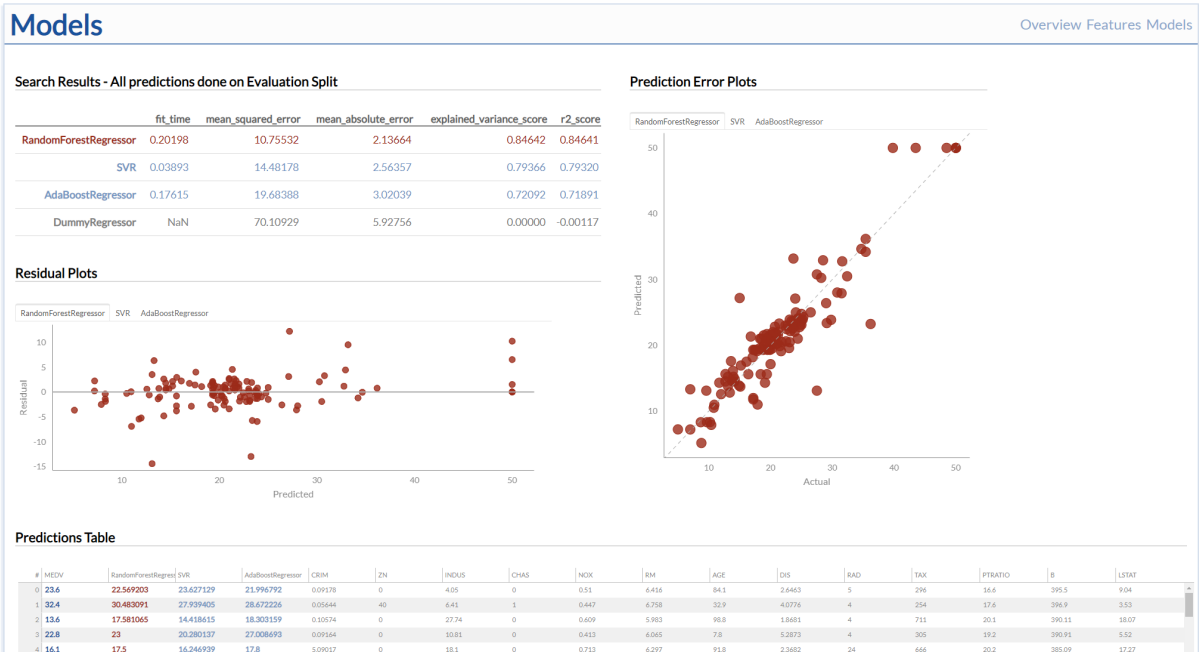


ScatterPlot Grid is included in the last section - `Bokeh` visualization that plots every feature against another as scatter plots in a manner similar to `seaborn` pairplot, but with a twist added - every feature is also used as a hue (coloring). The idea behind it was to provide visual assistance with manual feature engineering (if there are any easily separable groups).

Note: Row with a chosen Feature on X axis and coloring by the same Feature is greyed out to minimize confusion (ideally separate colored groups would be present). Greying out was included as an alternative to removing that row, as it was technically difficult to not break the whole structure of the Grid while doing that.



2.2.3 Models



`Models` subpage gives information on provided Models performance. Shown results and visualizations will always correspond to the 3 best scoring Models chosen during HTML Dashboard creation (`search` methods). Even though provided plots will be different based on a problem type, scoring table in the upper left corner and predictions table at the bottom stay the same.

Note: Model names are 'hoverable' in a similar manner to Feature Names in `Overview` subpage - the difference being that instead of descriptions, used *params* are shown.

Classification

`Models` subpage in *classification* problem generates Plots of performance curves (`ROC curve`, `precision-recall curve` and `detection error tradeoff curve`) and Confusion Matrices for every model (see sklearn roc, precision-recall, detection error tradeoff).

Search Results - All predictions done on Evaluation Split

	fit_time	accuracy_score	balanced_accuracy_score	f1_score	roc_auc_score
SVC #1	0.00698	0.96503	0.96827	0.97143	0.99480
SVC #2	0.00399	0.96503	0.96099	0.97207	0.99397
SVC #3	0.00300	0.96503	0.96099	0.97207	0.99334
DummyClassifier	NaN	0.53846	0.50177	0.63736	0.50177

Confusion Matrices

SVC #1

	Predicted Negative	Predicted Positive
Actual Negative	53	1
Actual Positive	4	85

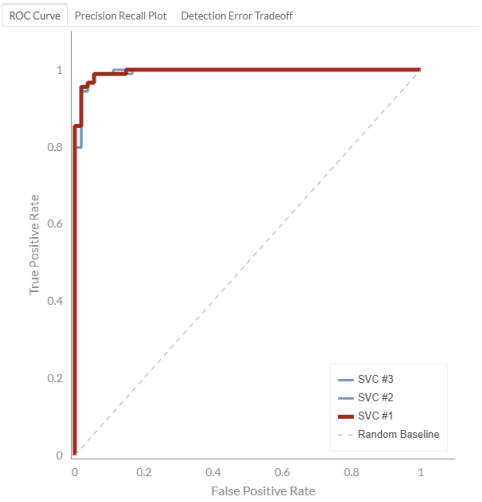
SVC #2

	Predicted Negative	Predicted Positive
Actual Negative	51	3
Actual Positive	2	87

SVC #3

	Predicted Negative	Predicted Positive
Actual Negative	51	3
Actual Positive	2	87

Result Curves Comparison



Note: Legend is interactive - you can mute lines by clicking at a particular legend entry.

Regression

`Models` subpage in *regression* problem generates two Plots for every Model: `prediction errors` and `residuals`.

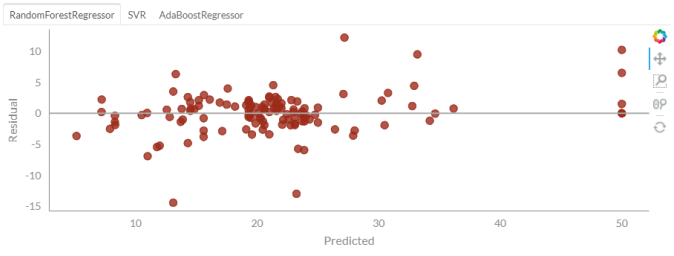
- `prediction errors` plot shows Actual Target on X axis and Predicted Target on Y Axis;
- `residuals` plot shows Actual Target on X axis and the difference between Predicted and Actual on Y Axis.

Models

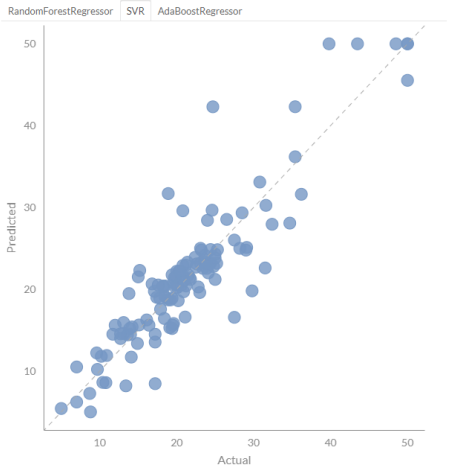
Search Results - All predictions done on Evaluation Split

	fit_time	mean_squared_error	mean_absolute_error	explained_variance_score	r2_score
RandomForestRegressor	0.20198	10.75532	2.13664	0.84642	0.84641
SVR	0.03893	14.48178	2.56357	0.79366	0.79320
AdaBoostRegressor	0.17615	19.68388	3.02039	0.72092	0.71891
DummyRegressor	NaN	70.10929	5.92756	0.00000	-0.00117

Residual Plots

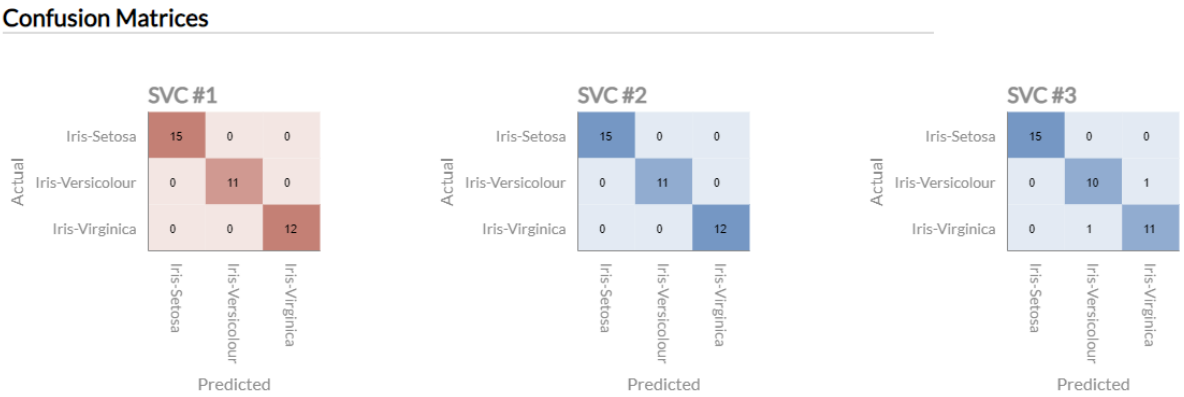


Prediction Error Plots



Multiclass

`Models` subpage in *multiclass* problem shows you Confusion Matrices for every Model assessed.



2.3 Examples

2.3.1 Library examples

Library comes with few defined *toy datasets* that you can use if you don't have any of your data handy.

```
from data_dashboard.examples import iris, boston, diabetes, digits, wine, breast_cancer
X, y, descriptions = iris()  # multiclass
X, y, descriptions = boston()  # regression
X, y, descriptions = diabetes()  # regression
X, y, descriptions = digits()  # multiclass
X, y, descriptions = wine()  # multiclass
X, y, descriptions = breast_cancer()  # classification
```

2.3.2 Example Dashboard

Deployed Dashboard example can be found here.

2.3.3 Documentation

`data_dashboard.examples.examples.`**`iris`**`()`
Return iris dataset with custom descriptions.

Returns X, y, descriptions

Return type tuple

`data_dashboard.examples.examples.`**`boston`**`()`
Return boston dataset with custom descriptions.

Returns X, y, descriptions

Return type tuple

data_dashboard.examples.examples.**diabetes**()
Return diabetes dataset with custom descriptions.

Returns X, y, descriptions

Return type tuple

data_dashboard.examples.examples.**digits**(*n_class=10*)
Return digits dataset. Descriptions are None.

Parameters **n_class** (*int, optional*) – number of classes to return, defaults to 10

Returns X, y, descriptions

Return type tuple

data_dashboard.examples.examples.**wine**()
Return wine dataset. Descriptions are None.

Returns X, y, descriptions

Return type tuple

data_dashboard.examples.examples.**breast_cancer**()
Return breast cancer dataset. Descriptions are None.

Returns X, y, descriptions

Return type tuple

2.4 Questions and Answers

- *What types of Machine Learning problems does data_dashboard work on?*

  Currently *classification*, *regression* and *multiclass* problems are viable. *Multilabel* problem is **not available**.

- *How are the Model results calculated?*

  Provided data is split into train and tests splits. Every time any Models search is initiated, Models are fit on a train portion of the data, but the scores are calculated on the test portion. However, the chosen Model (or the one set by you with `set_and_fit` method) is fit on **all** data.

- *What are the default models used for searches?*

  You can inspect default models used here.

- *What are the default Transformers used?*

  For Categorical Features:

  - `SimpleImputer(strategy="most_frequent")`
  - `OneHotEncoder(handle_unknown="ignore")`

  For Numerical Features:

  - `SimpleImputer(strategy="median")`
  - `QuantileTransformer(output_distribution="normal")`
  - `StandardScaler()`

- *Can I provide my own default models to GridSearch instead of those defined in models.py module?*

Yes! You can do it with `models` argument in `create_dashboard` method - you just need to provide your own dictionary object with *Model class: param_grid* pairs.

- *What to do when I am working on a NLP-related problem?*

  At the time of building this library, I wasn't experienced enough to tackle the NLP problem in any simplified, automated way. The best way to approach it would be to transform the data on your end and then provide transformed `X` and `y` to the `Dashboard` (with `already_transformed_columns` argument as needed).

- *Why did you decide on HTML static files instead of server-client architecture (e.g. with bokeh server)?*

  Two main reasons:

  - I wanted to have a lightweight HTML output that doesn't rely on any server processes (either localhost or regular ones);
  - I have done something similar in my other project and I wanted to try something new.

2.4.1 Known Issues / Problems

- *Multilabel* problem type is currently not supported.
- `Features` subpage might get laggy when trying to change feature selection and when number of all features in the data is high.
- CSS/HTML might get *wonky* sometimes.

2.5 License

2.5.1 Contact

Question? Please contact dowgird.maciej@gmail.com

2.6 Help

Question? Please contact dowgird.maciej@gmail.com or open an issue on github: https://github.com/maciek3000/data_dashboard/issues

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

W

X

Y