SAMPO Analytics Guide

第 1 版
2020 年 1 月

日本電気株式会社

目次

第1章 はじめに

本ガイドではデータサイエンティストを対象として異種混合学習(SAMPO/FAB)を利用した検証フェーズの分析の流れをいくつかの分析のシナリオ例を用いて説明します。

第2章 異種混合学習技術

本章では、異種混合学習技術の特徴と異種混合学習技術を搭載したソフトウェア SAMPO/FAB の概要を説明します。

2.1 背景

実社会にはさまざまなデータがあり、それを分析することが社会的・経済的価値の創造やサービス向上に役立つと期待されています。一般に、データには多数の規則性が混ざり合っており、そのまま分析すると、単純なモデルでは十分な精度を得られないことや複雑で人間が解釈しづらいモデルが必要になることが多くあります。人間が解釈可能なかたちで十分な精度を得るには、同じ規則性をもつ集団ごとにデータを分割し、それぞれで単純なモデルを学習する必要があります。また、良いモデルを得るためには、いくつかの代表的なアルゴリズムにより分析対象となる元データを変換（属性生成）し、それを用いて分析することが効果的です。これらのデータの規則性ごとへの分割や属性生成方法の最適化には、知識・経験と試行錯誤が必要になります。しかし、データ分析を担うデータサイエンティストが世界的に不足している現状が、上述の期待を達成するための大きな課題となっています。

2.2 異種混合学習技術の特徴

異種混合学習技術は、データの規則性ごとへの分割を解決するために NEC が独自に開発した技術で、因子化漸近ベイズ (Factorized Asymptotic Bayesian; FAB) 推論を用いたアルゴリズムのことです。FAB 推論は、データ分割とそれに基づくモデル推定を、自動的かつ高速に行うことができる機械学習の理論です。FAB 推論では、複数の規則性が混ざり合ったデータに対するモデルの良さを適切に評価できるように、独自に考案された因子化情報量基準（Factorized Information Criteria; FIC）が用いられています。

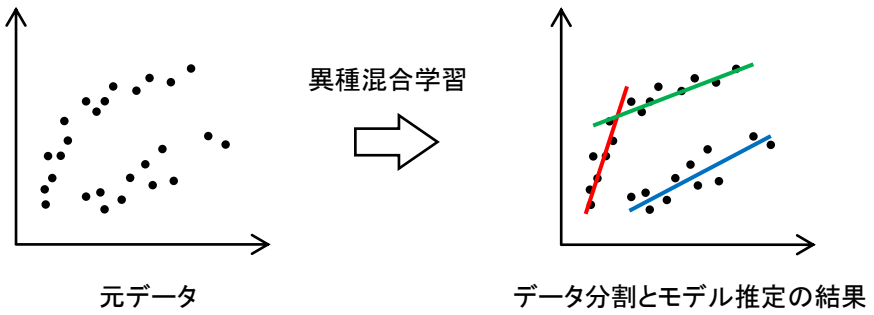


図 2-1 異種混合学習技術によるデータ分割とモデル推定

1. Ryohei Fujimaki, Satoshi Morinaga: Factorized Asymptotic Bayesian Inference for Mixture Modeling. AISTATS 2012: 400-408.
2. Riki Eto, Ryohei Fujimaki, Satoshi Morinaga, Hiroshi Tamano: Fully-Automatic Bayesian Piecewise Sparse Linear Models. AISTATS 2014: 238-246.
3. 藤巻 遼平、森永 聡、江藤 力、本橋 洋介、菅野 亨太: 異種混合学習技術とビッグデータ分析ソリューションの研究開発. 第 29 回先端技術大賞 「フジサンケイ ビジネスアイ賞」受賞論文.

2.3 SAMPO/FAB の概要

異種混合学習技術を用いた分析エンジンを搭載したソフトウェアが SAMPO/FAB です。SAMPO/FAB では、説明変数と目的変数からなる構造化データを扱うことができます。属性生成を簡単に行える属性生成器が複数搭載されています。また、異種混合学習技術を用いた予測器も複数搭載されています。

第3章 検証フェーズの分析の流れ

本章では、検証フェーズの分析の流れと各分析タスクの役割を説明します。

検証フェーズの分析の流れを図 3-1 に示します。

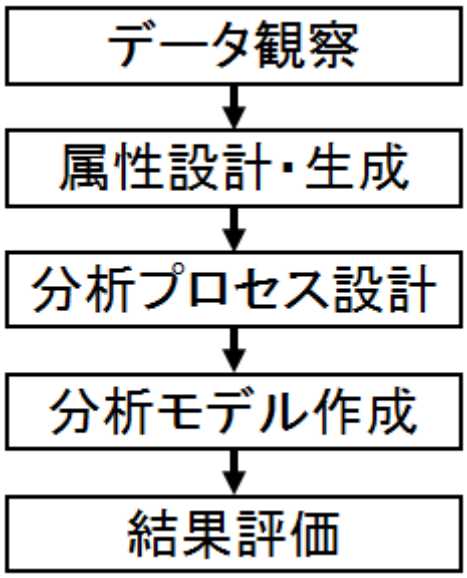


図 3-1 検証フェーズの分析の流れ

3.1 データ観察

データの特徴や傾向を確認するためにデータを観察します。たとえばデータ前処理を実施した後、Jupyter Notebook 上で Pandas DataFrame を用いてデータをグラフィカルに表示したり、SQL を利用してデータの傾向を調べたり、matplotlib を利用してデータをグラフで表示したりすることで確認ができます。

3.2 属性設計・生成

分析を行うために属性を設計し、必要に応じて属性を生成します。設計を基に SAMPO/FAB が対応するデータ形式で入力データを作成します。対応するデータ形式は、CSV ファイル、データベース（PostgreSQL）、Pandas DataFrame のいずれかです。また、各属性の情報(名前、データ型を表すスケールなど)を定義する属性スキーマを作成します。

3.3 分析プロセス設計

より良いモデルを学習するため、分析プロセスを設計します。分析プロセスは 1 つの「プロセス記述」と 1 つの「プロセス実行設定」から成ります。プロセス記述（SPD）[1] には「コンポーネント[2]」間のデータの流れと「コンポーネント」のパラメーターを定義します。プロセス実行設定（SRC）[3]には入力データと属性スキーマなどを定義します。

---

[1] SPD の詳細については『SAMPO Reference』の「SPD Specification」を参照してください。

[2] コンポーネントの一覧と各コンポーネントの詳細については『SAMPO Reference』を参照してください。

[3] SRC の詳細については『SAMPO Reference』の「SRC Specification」を参照してください。

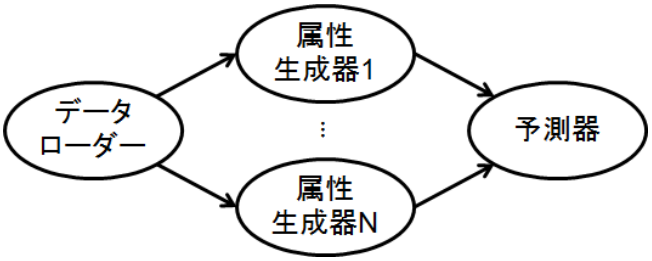


図 3-2 分析プロセスのイメージ

SPD に定義するコンポーネントの種類には、入力データを読み込む「データロードコンポーネント」、属性データを変換して新たな属性データを生成する「属性生成コンポーネント」、さらに学習や予測を行う「予測コンポーネント」などがあります。属性生成コンポーネントには標準化を行うコンポーネントや二値展開するコンポーネントなどがあり、予測コンポーネントには異種混合学習技術を実装した FAB コンポーネントなどがあります。

3.4 モデル作成

設計した分析プロセスを用いてモデルを作成します。SAMPO/FAB のモデルは内部でランダムに生成する初期状態[4]に依存するため、データサイエンティストが分析プロセスを複数回実行して、最良なモデルを選択することをよく行います。この複数回実行することをランダムリスタートと呼びます。

分析結果は独自の形式でプロセスストアに保存されます。プロセスストアはモデルを含む分析結果を保存するための領域です。

3.5 結果評価

最良のモデルを選択するために、人間が理解できるテキスト形式や画像形式に可読化、可視化します。分析結果にはモデル（門木と予測式）の情報、また RMSE などの予測精度に関する値が含まれており、モデルの解釈や評価指標に基づいて結果評価を行い、最良なモデルを選択します。

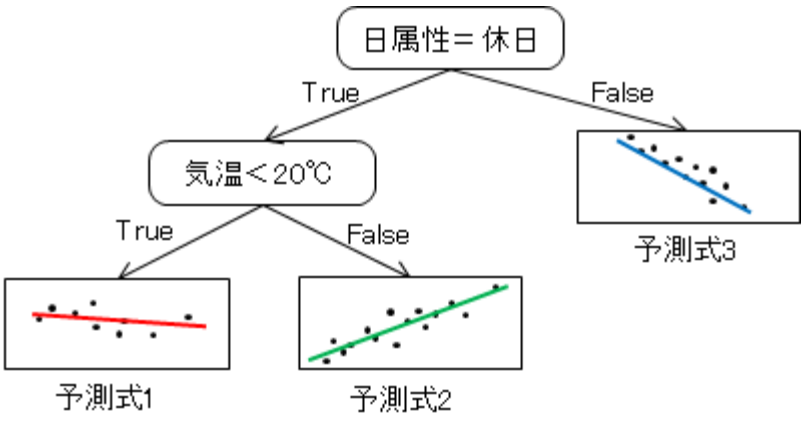


図 3-3 FAB コンポーネントの門木と予測式

例えば、FAB コンポーネントのモデルは、図 3-3 のように門木と予測式で構成されます。デー

[4] NumPy の np.random.rand モジュールを使用して生成しています。
(https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rand.html#numpy.random.rand)

タと門木の分岐条件を示す門関数に従って、サンプルごとに予測式が割り当てられて、それぞれの予測式が予測を行います。FAB コンポーネントでは、学習の中で予測に必要と判断された少数の属性のみのモデルで表現されます。従って、膨大な属性数からなるデータで学習したモデルであっても、門木と予測式を読み解くことで、どのように目的変数の値が生成されるかを解釈できます。

結果評価の内容を確認し、必要に応じて属性設計・生成や分析プロセス設計などの分析タスクへ戻り、分析の流れを繰り返し、分析結果や可視化したモデルなどを基にレポートを作成します。

第4章 分析を対話的に実行する

本章では、分析のシナリオ例を用いて、第 3 章 で示した分析の流れの各タスクとその中での重要な概念を説明します。本ガイドでは Jupyter Notebook[5]を使用した例を用いて説明します。API とコマンドの外部仕様の詳細については『SAMPO Reference』を参照してください。

各節で使用するデータ[6]は同梱の data ディレクトリに格納しています。同じく notebook ファイルは notebooks ディレクトリに格納しています。

4.1 回帰問題のモデル作成

本節では、数値型の属性を予測する回帰問題の例として、中古車の価格を予測するモデルを得た後、予測するシナリオを説明します。

- 4.1.1 データ観察 /4.1.2 属性設計・生成
  CSV ファイル形式の入力データを利用します。
- 4.1.5 結果評価
  分析プロセスで得られた複数のモデルに対して、学習とは別の区間のデータに対する RMSE（予測誤差）を評価して、最良なモデルを選択します。また、最良なモデルを可視化してモデル特性を確認します。
- 4.1.6 最良なモデルで予測
  最良なモデルで予測を行い、予測結果に対する予実グラフを出力します。

notebook は notebooks/regression.ipynb を確認してください。

4.1.1 データ観察

Notebook Dashboard の Files タブで Jupyter Notebook のルートディレクトリに automobile フォルダを作成します。このフォルダに移動し、入力データ（data/automobile_learn.csv、data/automobile_predict.csv）をアップロードします。新しい notebook を作成し、学習データの上から 5 件のデータを確認します。

In [1]:

```
import pandas as pd

learn_data = pd.read_csv('./automobile_learn.csv', na_values='?')
learn_data.head(5)
```

[5] 詳細については https://jupyter.org/を参照してください。

[6] データは UCI のオープンデータを基に作成しています（https://archive.ics.uci.edu/ml/datasets/Automobile, https://archive.ics.uci.edu/ml/datasets/Automobile+Marketing）。

Out[1]:

```
symboling normalized-losses make num-of-doors wheel-base length width height price
0  3 145.0   dodge  two  95.9 173.2 66.3 50.2 12964
1  0   NaN renault four  96.1 181.5 66.5 55.2  9295
2  0  89.0  subaru four  97.0 173.5 65.4 53.0 10198
3  0 115.0   mazda four  98.8 177.8 66.5 55.5 11245
4 -1   NaN  toyota four 104.5 187.8 66.5 54.1 15750
```

4.1.2 属性設計・生成

データ観察結果から値のないサンプル（NaN のサンプル）があることでわかるので、削除します。また、ユニークなサンプル ID である'_sid'の属性を生成します。最後に意図通りになっているか確認します。

In [2]:

```
learn_data.dropna(inplace=True)
learn_data['_sid'] = list(range(learn_data.shape[0]))
learn_data.head(5)
```

Out[2]:

```
symboling normalized-losses make num-of-doors wheel-base length width height price
_sid
0  3 145.0         dodge  two  95.9 173.2 66.3 50.2 12964 0
2  0  89.0        subaru four  97.0 173.5 65.4 53.0 10198 1
3  0 115.0         mazda four  98.8 177.8 66.5 55.5 11245 2
5  0 108.0        nissan four 100.4 184.6 66.5 56.1 14399 3
7 -1  93.0 mercedes-benz four 110.0 190.9 70.3 56.5 25552 4
```

属性スキーマ記述（ASD）[7]を生成し、意図した属性スキーマになっていることを確認します。

In [3]:

```
import logging
from sampo.api import sampo_logging
from sampotools.api import gen_asd_from_pandas_df

sampo_logging.configure(logging.INFO, filename='./automobile_rg.log')
asd = gen_asd_from_pandas_df(learn_data)
pd.DataFrame(asd).T[['scale', 'domain']]
```

Out[3]:

```
                    scale                                                domain
symboling         INTEGER                                                   NaN
normalized-losses    REAL                                                   NaN
make              NOMINAL  [toyota, nissan, mazda, honda, subaru, mitsubi...
num-of-doors      NOMINAL                                          [four, two]
wheel-base           REAL                                                   NaN
length               REAL                                                   NaN
width                REAL                                                   NaN
height               REAL                                                   NaN
price             INTEGER                                                   NaN
_sid               INTEGER                                                   NaN
```

[7] ASD の詳細については『SAMPO Reference』の「ASD Specification」を参照してください。

4.1.3 分析プロセス設計

分析プロセスを下記のとおりに設計します。

SPD はデータローダーの dl コンポーネント、二値展開の bexp コンポーネント[8]、標準化の std コンポーネント[9]、FAB 線形回帰予測器の rg コンポーネント[10]を図 4-1 のようにつなげます。また、rg コンポーネントに対して、属性名が price の属性を目的変数として指定します。

SRC は学習用と予測用の分析プロセスをそれぞれ 5 回、計 10 回分を作成します。テンプレートを利用してプロセス名や入力データを変更しながら作成します。

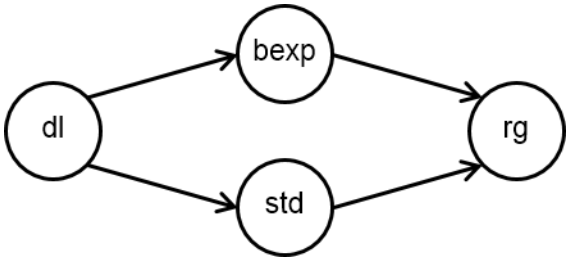


図 4-1 コンポーネント間のデータの流れ

[8] 詳細については『SAMPO Reference』の「BinaryExpandFD Component Specification」を参照してください。

[9] 詳細については『SAMPO Reference』の「StandardizeFD Component Specification」を参照してください。

[10] 詳細については『SAMPO Reference』の「FABHMEBernGateLinearRg Component Specification」を参照してください。

SPD を定義します。

In [4]:

```
from sampo.api import gen_spd

# SPD の定義
spd_content = '''
dl -> std  -> rg
   -> bexp -> rg

---

components:
    dl:
        component: DataLoader

    std:
        component: StandardizeFDComponent
        features: scale == 'real' or scale == 'integer'

    bexp:
        component: BinaryExpandFDComponent
        features: scale == 'nominal'

    rg:
        component: FABHMEBernGateLinearRgComponent
        features: name != 'price'
        target: name == 'price'
        standardize_target: True
        tree_depth: 2

global_settings:
    keep_attributes:
        - price
    feature_exclude:
        - price
'''
spd = gen_spd(template=spd_content)
```

SRC を定義します。

In [5]:

```
from sampo.api import gen_src

# train 用(学習用)の SRC のテンプレート定義
train_src_templ = '''
automobile_train_{{ random_restart }}:
    type: learn
    data_sources:
        dl:
            df: {{ data_df }}
            attr_schema: {{ asd }}
            filters:
                - k_split(10, 0, True)
'''
# validation 用(モデル選択用)の SRC のテンプレート定義
validation_src_templ = '''
automobile_validation_{{ random_restart }}:
    type: predict
    data_sources:
        dl:
            df: {{ data_df }}
            attr_schema: {{ asd }}
            filters:
                - k_split(10, 0, False)
    model_process: automobile_train_{{ random_restart }}
'''

# プロセスリストの作成
process_list = []
num_random_restarts = 5
for idx in range(num_random_restarts):
    src_param = {'random_restart': idx, 'data_df': learn_data, 'asd': asd}
    train_src = gen_src(template=train_src_templ, params=src_param)
    validation_src = gen_src(template=validation_src_templ, params=src_param)
    process_list.append((train_src, spd)) # 学習用の分析プロセスの追加
    process_list.append((validation_src, None)) # 予測用の分析プロセスの追加
```

4.1.4 モデル作成

分析結果を保存するためのプロセスストアを作成します。

In [6]:

```
from sampo.api import process_store

pstore_url = 'pstore_rg'
process_store.create(pstore_url)
```

プロセスを 3 並列で実行し、実行結果を確認します。

In [7]:

```
from sampo.api import process_runner

process_runner.session_run(
    process_list, pstore_url=pstore_url, max_workers=3)

process_store.list_process_metadata(pstore_url)
```

Out[7]:

```
process name version started at running time status
0 automobile_train_0 8a6b27ec… 2018-06-26 10:27:56… 00:00:01… Succeeded
1 automobile_train_1 2a18dcb9… 2018-06-26 10:27:56… 00:00:00… Succeeded
2 automobile_train_2 1623270d… 2018-06-26 10:27:56… 00:00:01… Succeeded
3 automobile_train_3 c32651bb… 2018-06-26 10:27:57… 00:00:00… Succeeded
4 automobile_train_4 38e1ad6e… 2018-06-26 10:27:58… 00:00:00… Succeeded
5 automobile_validation_0 9e4b9db4… 2018-06-26 10:27:58… 00:00:00… Succeeded
6 automobile_validation_1 a5af60a3… 2018-06-26 10:27:58… 00:00:00… Succeeded
7 automobile_validation_2 eae3196b… 2018-06-26 10:27:58… 00:00:00… Succeeded
8 automobile_validation_3 05c6ca01… 2018-06-26 10:27:59… 00:00:00… Succeeded
9 automobile_validation_4 80519965… 2018-06-26 10:27:59… 00:00:00… Succeeded
```

4.1.5 結果評価

得られた分析結果から最良なモデルを選択するために、各モデルの RMSE を表示します。

In [8]:

```
import re

rmse = []
validation_proc_names = [src.name for src, _ in process_list
                         if re.match('automobile_validation.*', src.ptype)]
for validation_proc_name in validation_proc_names:
    with process_store.open_process(pstore_url, validation_proc_name) as prl:
        evaluation = prl.load_comp_output_evaluation('rg')
        rmse.append(evaluation['root_mean_squared_error'][0])

pd.DataFrame(rmse, columns=['rmse'],
             index=validation_proc_names).sort_values('rmse')
```

Out[8]:

```
                              rmse
automobile_validation_2   1776.089590
automobile_validation_4   1952.737155
automobile_validation_3   2063.911535
automobile_validation_0   2156.413447
automobile_validation_1   2342.302751
```

上記の結果から RMSE が最小のモデルを最良モデルとして選択し、可視化します。可視化で門木構造や予測式の詳細を確認します。

In [9]:

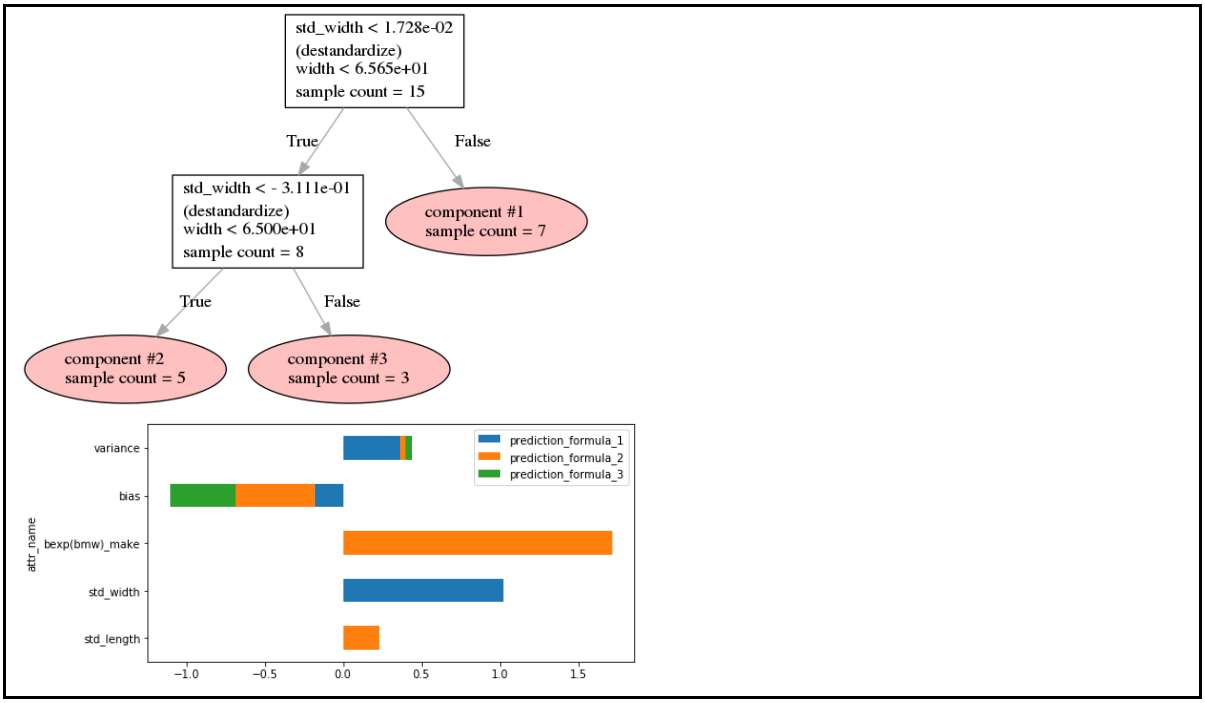
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import display, Image
from sampovis.api.fabhme_vis import save_gate_tree

best_model_proc_name = 'automobile_validation_2'

# モデル情報と予測結果を取得
with process_store.open_process(pstore_url, best_model_proc_name) as prl:
    model_params = prl.load_model('rg')

# 門木の可視化
save_gate_tree(pstore_url, best_model_proc_name, 'output')
display(Image(
    filename='output/{}_rg_fabhme_gate_tree.png'.format(best_model_proc_name)))

# 予測式の詳細
prediction_formulas = model_params['prediction_formulas']
relevant_feature_indices = prediction_formulas.sum(axis=1) != 0
prediction_formulas = prediction_formulas[relevant_feature_indices]
prediction_formulas.plot(kind='barh', figsize=(8,4), stacked=True)
plt.show()
```



上の図はモデルの門木構造を表しており、四角に分岐条件が書かれ、丸に各予測式の番号とその予測式で予測するサンプル数が記載されています。ここでは、width がデータの分割に寄与していることがわかります。また、下の図はステムプロットで表しており、各説明変数の目的変数への寄与度を記載しています。たとえば、予測式 2 の bexp(bmw)_make は正に大きく寄与しており、予測式 1 で予測する価格は width が大きいと価格が大きくなることがわかります。

4.1.6 最良なモデルで予測

最良なモデルで予測を実行します。

まず、学習時と同じように属性設計・生成を行います。

In [10]:

```
predict_data = pd.read_csv('./automobile_predict.csv', na_values='?')
predict_data.dropna(inplace=True)
predict_data['_sid'] = list(range(predict_data.shape[0]))
predict_data.head(5)
```

Out[10]:

```
 symboling  normalized-losses  make  num-of-doors  wheel-base  length  width  height
price _sid
0 1   87.0 toyota two    95.7 158.7 63.6 54.5 6338  0
1 3 150.0   saab two    99.1 186.6 66.5 56.1 18150 1
2 2 134.0 toyota two    98.4 176.2 65.6 53.0 17669 2
3 2 134.0 toyota two    98.4 176.2 65.6 52.0 11549 3
4 0 145.0 jaguar four 113.0 199.6 69.6 52.8 32250 4
```

予測用の SRC を定義します。

In [11]:

```
#  最良なモデルでの予測
predict_src_templ = '''
automobile_predict:
    type: predict
    data_sources:
        dl:
            df: {{ data_df }}
            attr_schema: {{ asd }}
    model_process: {{ model_process }}
'''
src_param = {'model_process': best_model_proc_name, 'data_df': predict_data,
             'asd': asd}
predict_src = gen_src(template=predict_src_templ, params=src_param)
```

予測用の分析プロセスを実行します。

In [12]

```
process_runner.run(src=predict_src, pstore_url=pstore_url)
proc_meta = process_store.list_process_metadata(pstore_url)
proc_meta[proc_meta['process name'] == 'automobile_predict']
```

Out[12]:

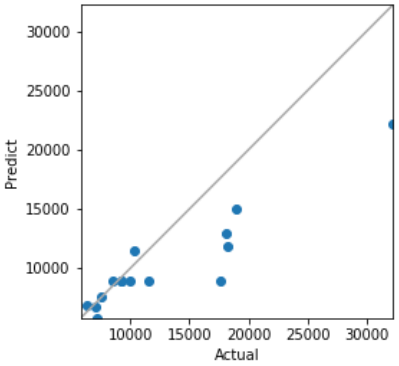
```
process name version started at running time status
0 automobile_predict da4a36cd… 2018-06-29 15:28:03… 00:00:00… Succeeded
```

予測結果を確認するため、予測と実績の散布図を表示します。

In [13]:

```
# 予測と実績の散布図
with process_store.open_process(pstore_url, 'automobile_predict') as prl:
    df = prl.load_comp_output('rg')

plt_df = df.loc[:, ['price', 'rg_predict']]
plt_df.rename(columns={'price': 'actual', 'rg_predict': 'predict'},
              inplace=True)
min_val = int(plt_df.values.min() - 1)
max_val = int(plt_df.values.max() + 1)
plt.scatter(plt_df['actual'], plt_df['predict'])
plt.plot(range(min_val, max_val), range(min_val, max_val),
         color='darkgrey')
plt.xlim(min_val, max_val)
plt.ylim(min_val, max_val)
plt.xlabel("Actual")
plt.ylabel("Predict")
plt.gca().set_aspect("equal", adjustable="box")
plt.show()
```



4.2 判別問題のモデル作成

本節では、カテゴリ型の属性値を予測する判別問題の例として、ダイレクトマーケティングの成功可否を予測するモデルを得た後、予測するシナリオを説明します。

ここでは、Pandas DataFrame 形式の入力データを利用し、4.1 と同様に最良なモデルを選択した後、そのモデルで予測するという分析を行います。

notebook は~/bank-marketing/に作成するものとして説明します。ファイルは notebooks/classification.ipynb を確認してください。

4.2.1 データ観察

Notebook Dashboard の Files タブで Jupyter Notebook のルートディレクトリに bank-marketing フォルダを作成します。このフォルダに移動し、(data/bank_learn.csv, data/bank_predict.csv)をアップロードします。新しい notebook を作成し、データオブジェクト(bank_learn、bank_predict)を作成します。

In [1]:

```
import pandas as pd

learn_data = pd.read_csv('bank_learn.csv')
```

年齢と各職業のサンプルがどの程度あるのかを確認します。

In [2]:

```
# 年齢に対して確認
print(learn_data.groupby('age')['answer'].count())
# 仕事に対して確認
print(learn_data.groupby('job')['answer'].count())
```

```
age
18       2
19       1
20       2
21       6
22       9
23      16
24      54
25      53
26      60
27      80
28      96
29     125
30     153
31     173
32     195
33     152
34     161
35     147
36     154
37     125
38     135
39     115
40      93
41     128
42     102
43      98
44      92
45      94
46      88
47     100
      ...
54      69
55      63
56      64
57      56
58      61
59      45
60      28
61       6
62       5
63       4
64       5
65       3
```

```
66      6
67      3
68      4
69      2
70      2
71      4
72      2
73      2
74      3
75      2
76      3
77      1
78      3
80      3
81      3
82      2
85      1
86      2
Name: answer, dtype: int64
job
admin.           906
blue-collar      792
entrepreneur     134
housemaid        102
management       299
retired          148
self-employed    141
services         354
student           72
technician       619
unemployed       104
unknown           37
Name: answer, dtype: int64
```

4.2.2 属性設計・生成

データ観察から年齢がばらけているので、年齢層（～29 歳、30～39 歳、40～49 歳、50 歳～）の属性と_sid の属性を生成します。

In [3]:

```
# learn データに対して属性生成
age_groups = learn_data['age'] // 10
age_groups[age_groups <= 2] = 2
age_groups[age_groups >= 5] = 5
age_groups = age_groups.astype(str) # SAMPO/FAB でカテゴリ値として扱うため
age_groups.name = 'age_group'
learn_data = pd.concat([learn_data, age_groups], axis=1)
learn_data['_sid'] = list(range(learn_data.shape[0])) # _sid の属性を追加
```

ASD を生成し、意図した属性スキーマになっていることを確認します。

In [4]:

```
import logging
from sampo.api import sampo_logging
from sampotools.api import gen_asd_from_pandas_df

sampo_logging.configure(logging.INFO, filename='./bank.log')
asd = gen_asd_from_pandas_df(learn_data)
pd.DataFrame(asd).T[['scale', 'domain']]
```

Out[4]:

	scale	domain
age	INTEGER	NaN
job	NOMINAL	[admin., blue-collar, technician, services, ma...
martial	NOMINAL	[married, single, divorced, unknown]
education	NOMINAL	[university.degree, high.school, basic.9y, pro...
contact	NOMINAL	[cellular, telephone]
month	NOMINAL	[may, jul, aug, jun, nov, apr, oct, sep, mar, ...
day_of_week	NOMINAL	[mon, thu, tue, wed, fri]
duration	INTEGER	NaN
previous	INTEGER	NaN
answer	NOMINAL	[no, yes]
age_group	NOMINAL	[3, 4, 5, 2]
_sid	INTEGER	NaN

4.2.3 分析プロセス設計

SPD を定義します。

In [5]:

```
from sampo.api import gen_spd

# SPD の定義
spd_content = '''
dl -> std  -> cl
   -> bexp -> cl

---

components:
    dl:
        component: DataLoader

    std:
        component: StandardizeFDComponent
        features: scale == 'real' or scale == 'integer'

    bexp:
        component: BinaryExpandFDComponent
        features: scale == 'nominal'

    cl:
        component: FABHMEBernGateLinearClComponent
        features: name != 'answer'
        target: name == 'answer'
        positive_label: 'yes'
        tree_depth: 2

global_settings:
    keep_attributes:
        - answer
    feature_exclude:
        - answer
'''
spd = gen_spd(template=spd_content)
```

SRC を定義し、プロセスリストを作成します。

In [6]:

```
from sampo.api import gen_src

# train 用(学習用)の SRC のテンプレート定義
train_src_templ = '''
bank_train_{{ random_restart }}:
    type: learn
    data_sources:
        dl:
            df: {{ data_df }}
            attr_schema: {{ asd }}
            filters:
                - k_split(10, 0, True)
'''

# validation 用(モデル選択用)の SRC のテンプレート定義
validation_src_templ = '''
bank_validation_{{ random_restart }}:
    type: predict
    data_sources:
        dl:
            df: {{ data_df }}
            attr_schema: {{ asd }}
            filters:
                - k_split(10, 0, False)
    model_process: bank_train_{{ random_restart }}

'''

# プロセスリストの作成
process_list = []
num_random_restarts = 5
for idx in range(num_random_restarts):
    src_param = {'random_restart': idx, 'data_df': learn_data, 'asd': asd}
    train_src = gen_src(template=train_src_templ, params=src_param)
    validation_src = gen_src(template=validation_src_templ, params=src_param)
    process_list.append((train_src, spd)) # 学習用の分析プロセスの追加
    process_list.append((validation_src, None)) # 予測用の分析プロセスの追加
```

4.2.4 モデル作成

分析結果を保存するためのプロセスストアを作成します。

In [7]:

```
from sampo.api import process_store

pstore_url = 'pstore_cl'
process_store.create(pstore_url)
```

プロセスを 3 並列で実行し、実行結果を確認します。

In [8]:

```
from sampo.api import process_runner

process_runner.session_run(
    process_list, pstore_url=pstore_url, max_workers=3)

process_store.list_process_metadata(pstore_url)
```

Out[8]:

```
process name version started at running time status
0 bank_train_0 a6e0396c… 2018-07-18 11:44:39… 00:00:07… Succeeded
1 bank_train_1 9524e4da… 2018-07-18 11:44:39… 00:00:05… Succeeded
2 bank_train_2 973db236… 2018-07-18 11:44:39… 00:00:06… Succeeded
3 bank_train_3 8c55ccef… 2018-07-18 11:44:45… 00:00:07… Succeeded
4 bank_train_4 51e71aad… 2018-07-18 11:44:46… 00:00:00… Succeeded
5 bank_validation_0 1722b8a3… 2018-07-18 11:44:49… 00:00:00… Succeeded
6 bank_validation_1 ab2db165… 2018-07-18 11:44:47… 00:00:01… Succeeded
7 bank_validation_2 fa3bf85d… 2018-07-18 11:44:48… 00:00:00… Succeeded
8 bank_validation_3 9522cc6e… 2018-07-18 11:44:52… 00:00:00… Succeeded
9 bank_validation_4 994987d6… 2018-07-18 11:44:53… 00:00:00… Succeeded
```

4.2.5 結果評価

得られた分析結果から最良なモデルを選択するために、各モデルの正解率を表示します。

In [9]:

```
import re

accuracies = []
proc_names = [src.name for src, _ in process_list
              if re.match('bank_validation.*', src.ptype)]
for proc_name in proc_names:
    with process_store.open_process(pstore_url, proc_name) as prl:
        evaluation = prl.load_comp_output_evaluation('cl')
        accuracies.append(evaluation['accuracy'][0])

pd.DataFrame(accuracies, columns=['accuracy'],
             index=proc_names).sort_values('accuracy')
```

Out[9]:

```
                   accuracy
bank_validation_4   0.908356
bank_validation_3   0.916442
bank_validation_0   0.919137
bank_validation_1   0.919137
bank_validation_2   0.924528
```

上記の結果から正解率が最大のモデルを最良モデルとして選択し、可視化します。可視化で門木構造や予測式の詳細を確認します。

In [10]:

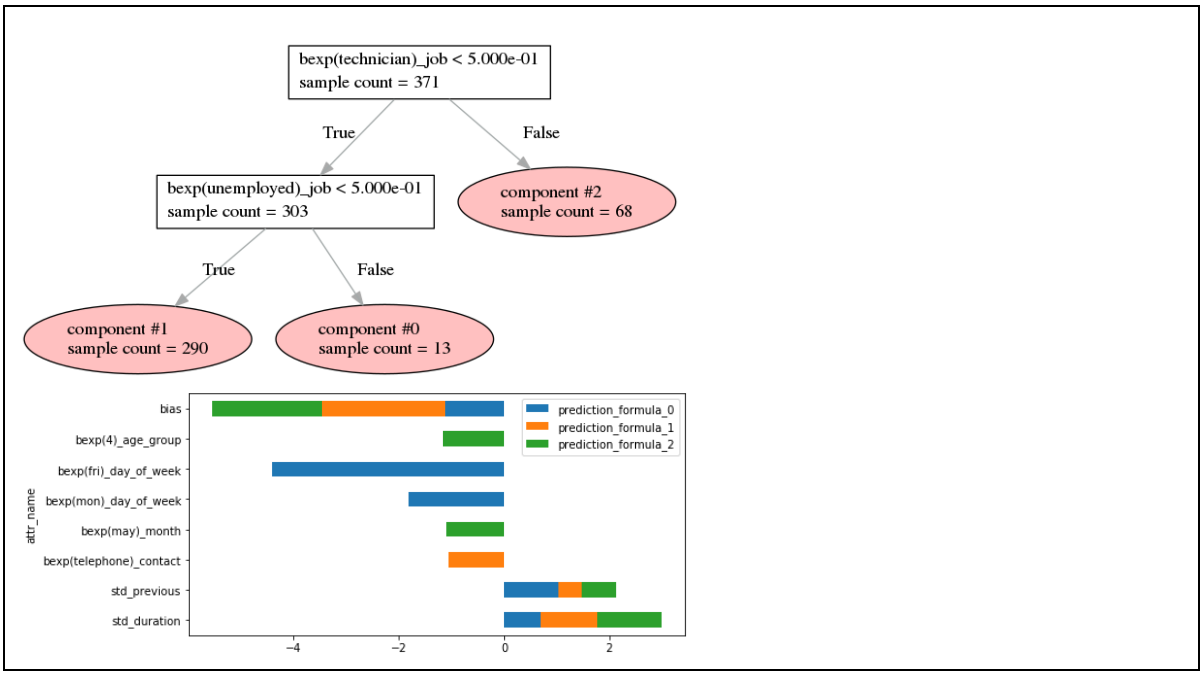
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import display, Image
from sampovis.api.fabhme_vis import save_gate_tree

best_model_proc_name = 'bank_validation_2'

# モデル情報と予測結果を取得
with process_store.open_process(pstore_url, best_model_proc_name) as prl:
    model_params = prl.load_model('cl')

# 門木の可視化
save_gate_tree(pstore_url, best_model_proc_name, 'output')
display(Image(
    filename='output/{}_cl_fabhme_gate_tree.png'.format(best_model_proc_name)))

# 予測式の詳細
prediction_formulas = model_params['prediction_formulas']
relevant_feature_indices = prediction_formulas.sum(axis=1) != 0
prediction_formulas = prediction_formulas[relevant_feature_indices]
prediction_formulas.plot(kind='barh', figsize=(8,4), stacked=True)
plt.show()
```



4.2.6 最良なモデルで予測

最良なモデルで予測を実行します。

まず、学習時と同じように属性設計・生成を行います。

In [11]:

```
# predict データに対して属性生成
predict_data = pd.read_csv('bank_predict.csv')
age_groups = predict_data['age'] // 10
age_groups[age_groups <= 2] = 2
age_groups[age_groups >= 5] = 5
age_groups = age_groups.astype(str)
age_groups.name = 'age_group'
predict_data = pd.concat([predict_data, age_groups], axis=1)
predict_data['_sid'] = list(range(predict_data.shape[0])) # _sid の属性を追加
```

予測用の分析プロセスの SRC を定義します。

In [12]:

```
predict_src_templ = '''
bank_predict:
    type: predict
    data_sources:
        dl:
            df: {{ data_df }}
            attr_schema: {{ asd }}
    model_process: {{ model_process }}
'''

src_param = {'data_df': predict_data, 'model_process': best_model_proc_name,
             'asd': asd}
predict_src = gen_src(template=predict_src_templ, params=src_param)
```

予測用の分析プロセスを実行します。

In [13]:

```
process_runner.run(src=predict_src, pstore_url=pstore_url)
proc_meta = process_store.list_process_metadata(pstore_url)
proc_meta[proc_meta['process name'] == 'bank_predict']
```

Out[13]:

```
process name version started at running time status
0 bank_predict 4f573228… 2018-07-18 11:44:55… 0:00:00… Succeeded
```

得られた分析結果を評価するために、正解率を表示します。

In [14]:

```
with process_store.open_process(pstore_url, predict_src.name) as prl:
    evaluation = prl.load_comp_output_evaluation('cl')
evaluation['accuracy'][0]
```

Out[14]:

```
0.9124087591240876
```

4.3 ホットスタートを用いたモデルの再学習

本節では、ホットスタート機能を用いてモデルの再学習を行う例として、新たなデータを用いて既存のモデルの再学習を行い、予測するシナリオを説明します。

一般的に、モデルは、予測対象のデータの傾向が学習時から変化することで予測精度が低下するため、モデルを運用する過程で再学習が必要となる場合があります。

ホットスタートを利用することで、モデルの門木構造や予測式の属性を保持したままモデルの再学習ができます。ランダムリスタートによるモデル探索の手間を削減できることに加えて、1 回あたりの学習時間も短縮できるため、再学習時間を大幅に削減することができます。

ホットスタートには、大きく以下の 2 種類があります。

- 事後確率ホットスタート
  指定したモデルと同じ門木構造を持つモデルを初期状態とし再学習を行います。
- モデルホットスタート
  指定したモデルを初期状態とし再学習を行います。その際モデル構造の一部を固定することができます。モデル構造をどのように固定するかは、モデルホットスタートのタイプ[11]によって異なります。

実際に、ホットスタートを用いて再学習を行う例を以下のシナリオで説明します。ここでは、店舗の売上を予測するモデルとして 2017 年 1 月から 6 月のデータで学習した現行のモデルがあるものの、2017 年 7 月以降のデータに対しては予測精度が想定する水準を下回っているとして、再学習により予測精度を改善する例を示します。

- 4.3.1 現行のモデルの確認
  使用するデータを準備します。現行のモデルを使って新しいデータに対する売上予測を行い、想定する予測精度を下回っていることを確認します。
- 4.3.2 分析プロセス設計 / 4.3.3 モデル作成
  新しいデータを使って、モデルホットスタートによる現行モデルの再学習を実施します。
- 4.3.4 結果評価
  4.3.1 と同様の確認を行い、再学習によって予測精度が改善していることを確認します。

notebook ファイルは notebooks/regression_with_hotstart.ipynb を確認してください。

4.3.1 現行のモデルの確認

Notebook Dashboard の Files タブで Jupyter Notebook のルートディレクトリに sales_hotstart フォルダを作成します。このフォルダに移動し、入力データ（data/sales_201707-12.csv、data/sales.asd、data/base_model_process.tar.gz）をアップロードします。

Jupyter の Terminal で以下を実行し、base_model_process.tar.gz を解凍します。

```
$ tar xzvf ~/sales_hotstart/base_model_process.tar.gz -C ~/sales_hotstart/
```

新しい notebook を作成し、売上データの一部を検証データとし、現行のモデルの予測精度、モ

[11] モデルホットスタートのタイプ一覧と各タイプの詳細については 『SAMPO Reference』を参照してください。

デル特性を確認します。

モデルが含まれる分析プロセスをインポートし、売上データ、ASD を読み込みます。また、モデル確認用の SPD、SRC を定義します。

In [1]:

```
import logging

import pandas as pd

from sampo.api import (
    sampo_logging, process_store, gen_spd, gen_src, process_runner
)
from sampotools.api import load_asd

sampo_logging.configure(logging.INFO, filename='./hotstart.log')

pstore_url = 'pstore_hotstart'
process_store.create(pstore_url)
process_store.import_process('./base_model_process', 'sales_train', pstore_url)

data_path = './sales_201707-12.csv'
asd_path = './sales.asd'

# 現行のモデル確認用の SRC のテンプレート定義
validation_src_templ = '''
sales_validation:
    type: predict
    data_sources:
        dl:
            path: {{ data_path }}
            attr_schema: {{ asd }}
            filters:
                - k_split(10, 0, False)
    model_process: sales_train
'''

src_param = {'data_path': data_path, 'asd': asd_path}
validation_src = gen_src(template=validation_src_templ, params=src_param)
```

分析プロセスを実行します。

In [2]:

```
process_runner.run(src=validation_src, pstore_url=pstore_url)
```

Out [2]:

```
sales_validation.337de76c-799e-4935-91af-671da223351c
```

RMSE と散布図を確認します。ここでは、この RMSE の値は、モデルを運用する上での予測精度の水準を下回っており、モデルの予測精度の改善を要することを想定しています。また、散布

図を確認し、実測値(Actual)が大きい領域で予測が外れていることを確認します。

In [3]:

```
process_name='sales_validation'
with process_store.open_process(pstore_url, process_name) as prl:
    evaluation = prl.load_comp_output_evaluation('rg')

evaluation['root_mean_squared_error'][0]
```

Out[3]:

```
12.995691202578689
```

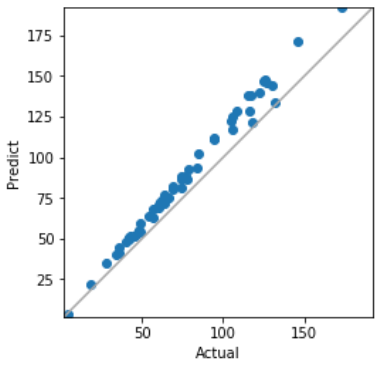
In [4]:

```
%matplotlib inline
import matplotlib.pyplot as plt

with process_store.open_process(pstore_url, process_name) as prl:
    output_df = prl.load_comp_output('rg')

plt_df = output_df.loc[:, ['sales', 'rg_predict']]
plt_df.rename(columns={'sales': 'actual', 'rg_predict': 'predict'},
inplace=True)
min_val = int(plt_df.values.min() - 1)
max_val = int(plt_df.values.max() + 1)
plt.scatter(plt_df['actual'], plt_df['predict'])
plt.plot(range(min_val, max_val), range(min_val, max_val),
color='darkgrey')
plt.xlim(min_val, max_val)
plt.ylim(min_val, max_val)
plt.xlabel("Actual")
plt.ylabel("Predict")
plt.gca().set_aspect("equal", adjustable="box")
plt.show()
```

Out [4]:

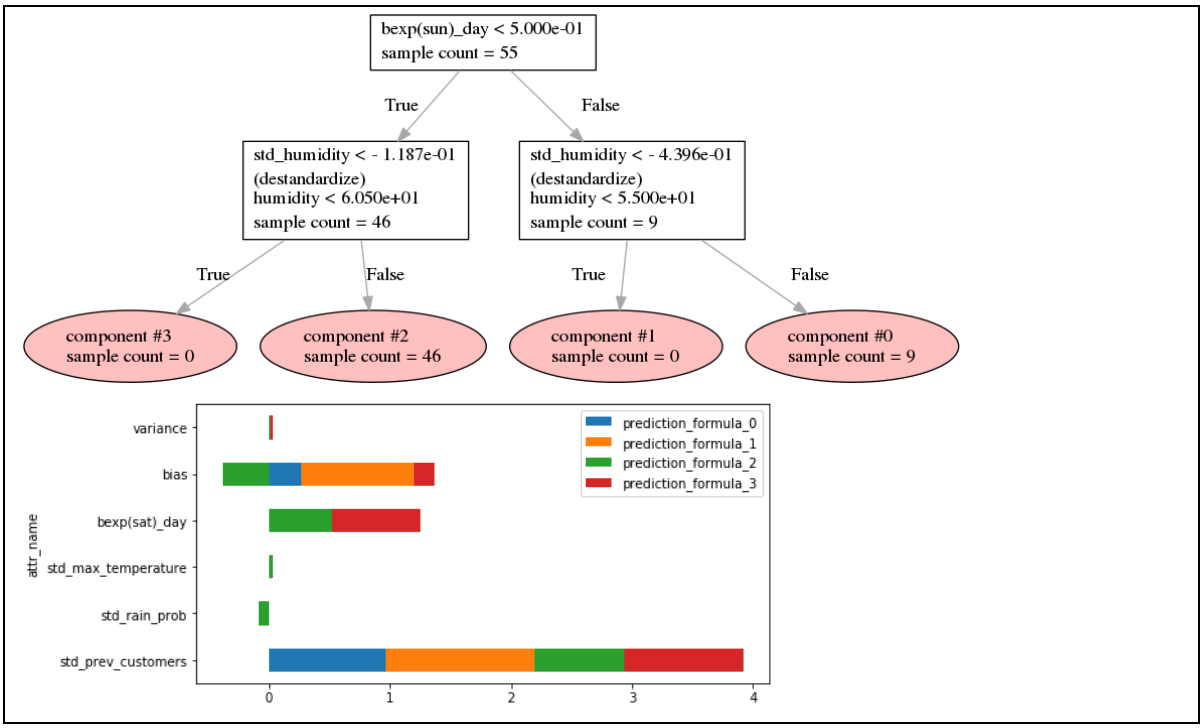


現行のモデルの構造を可視化します。

In [5]:

```
import numpy as np
from IPython.display import display, Image
from sampovis.api.fabhme_vis import save_gate_tree
model_proc_name = 'sales_validation'
# モデル情報と予測結果を取得
with process_store.open_process(pstore_url, model_proc_name) as prl:
    model_params = prl.load_model('rg')

# 門木の可視化
save_gate_tree(pstore_url, model_proc_name, 'output')
display(Image(
    filename='output/{}_rg_fabhme_gate_tree.png'.format(model_proc_name)))
# 予測式の詳細
prediction_formulas = model_params['prediction_formulas']
relevant_feature_indices = prediction_formulas.sum(axis=1) != 0
prediction_formulas = prediction_formulas[relevant_feature_indices]
prediction_formulas.plot(kind='barh', figsize=(8,4), stacked=True)
plt.show()
```



4.3.2 分析プロセス設計

再学習を行うための分析プロセスを設計します。

SPD を定義します。

In [6]:

```
spd_content = '''
dl -> std  -> rg
   -> bexp -> rg

---

components:
    dl:
        component: DataLoader

    std:
        component: StandardizeFDComponent
        features: scale == 'real' or scale == 'integer'

    bexp:
        component: BinaryExpandFDComponent
        features: scale == 'nominal'

    rg:
        component: FABHMEBernGateLinearRgComponent
        features: name != 'sales'
        target: name == 'sales'
        standardize_target: True

global_settings:
    keep_attributes:
        - sales
    feature_exclude:
        - sales
'''
spd = gen_spd(template=spd_content)
```

SRC を定義します。ここでは、モデルホットスタートのタイプとして、予測式の係数のみを再学習する mh_refit _comp を選択します。

In [7]:

```
# 再学習用の SRC のテンプレート定義
retrain_src_templ = '''
sales_retrain:
    type: learn
    data_sources:
        dl:
            path: {{ data_df }}
            attr_schema: {{ asd }}
            filters:
                - k_split(10, 0, True)
    hotstart:
        rg:
            base_model: sales_train
            type: mh_refit_comp
'''
# validation 用(検証用)の SRC のテンプレート定義
validation_src_templ = '''
sales_revalidation:
    type: predict
    data_sources:
        dl:
            path: {{ data_df }}
            attr_schema: {{ asd }}
            filters:
                - k_split(10, 0, False)
    model_process: sales_retrain
'''
src_param = {'data_df': data_path , 'asd': asd_path}
retrain_src = gen_src(template=retrain_src_templ, params=src_param)
validation_src = gen_src(template=validation_src_templ, params=src_param)
```

4.3.3 モデル作成

分析プロセスを実行します。

In [8]:

```
process_runner.run(spd=spd, src=retrain_src, pstore_url=pstore_url)
process_runner.run(src=validation_src, pstore_url=pstore_url)
```

Out [8]:

```
sales_revalidation.2828ab7a-0808-4e76-983b-b31355317e23
```

4.3.4 結果評価

RMSE を確認し、再学習後のモデルでは予測精度が向上していることを確認します。また、散布図から、実測値(Actual)が大きい領域でも予測誤差が小さくなっていることを確認します。

In [9]:

```
process_name='sales_revalidation'
with process_store.open_process(pstore_url, process_name) as prl:
    evaluation = prl.load_comp_output_evaluation('rg')

evaluation['root_mean_squared_error'][0]
```

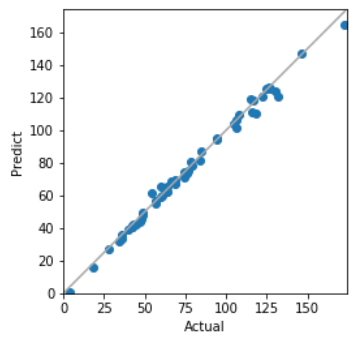
Out [9]:

```
3.26746217655499
```

In[10]:

```
with process_store.open_process(pstore_url, process_name) as prl:
    output_df = prl.load_comp_output('rg')

plt_df = output_df.loc[:, ['sales', 'rg_predict']]
plt_df.rename(columns={'sales': 'actual', 'rg_predict': 'predict'},
inplace=True)
min_val = int(plt_df.values.min() - 1)
max_val = int(plt_df.values.max() + 1)
plt.scatter(plt_df['actual'], plt_df['predict'])
plt.plot(range(min_val, max_val), range(min_val, max_val),
color='darkgrey')
plt.xlim(min_val, max_val)
plt.ylim(min_val, max_val)
plt.xlabel("Actual")
plt.ylabel("Predict")
plt.gca().set_aspect("equal", adjustable="box")
plt.show()
```

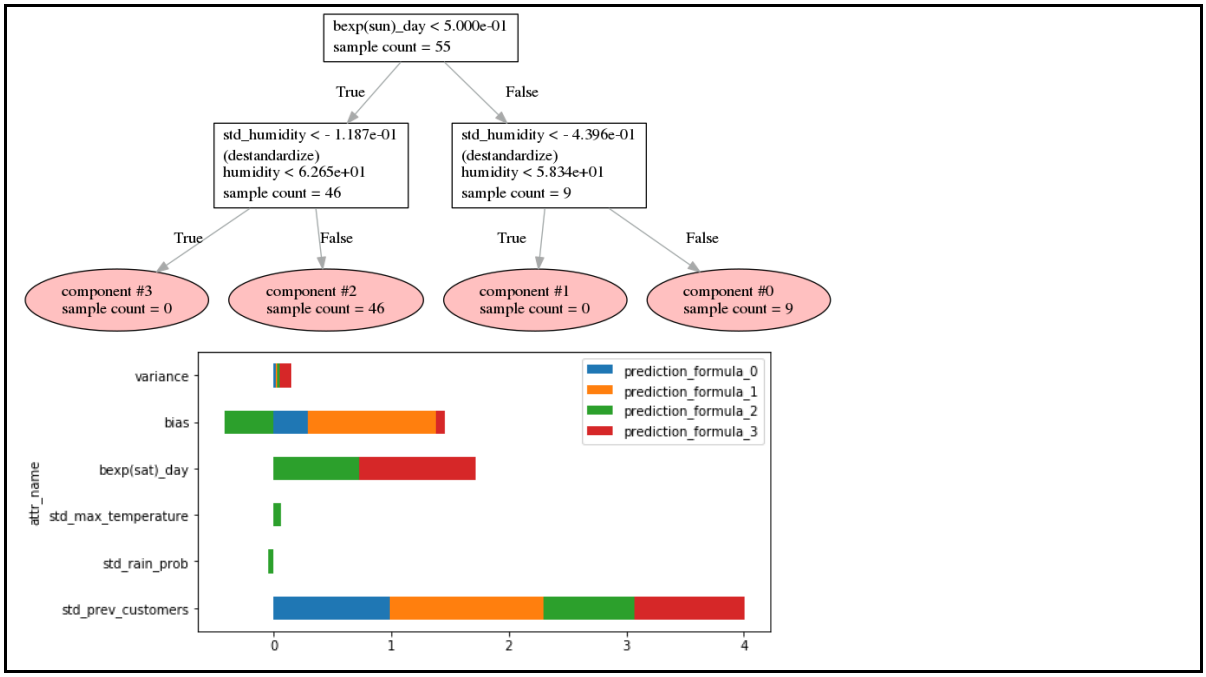


再学習後のモデルを可視化し、門木の構造が変更されずに、予測式の係数のみが変更されていることを確認します。

In [11]:

```
model_proc_name = 'sales_revalidation'
# モデル情報と予測結果を取得
with process_store.open_process(pstore_url, model_proc_name) as prl:
    model_params = prl.load_model('rg')

# 門木の可視化
save_gate_tree(pstore_url, model_proc_name, 'output')
display(Image(
    filename='output/{}_rg_fabhme_gate_tree.png'.format(model_proc_name)))
# 予測式の詳細
prediction_formulas = model_params['prediction_formulas']
relevant_feature_indices = prediction_formulas.sum(axis=1) != 0
prediction_formulas = prediction_formulas[relevant_feature_indices]
prediction_formulas.plot(kind='barh', figsize=(8,4), stacked=True)
plt.show()
```



SAMPO Analytics Guide

2020 年 1 月

日本電気株式会社