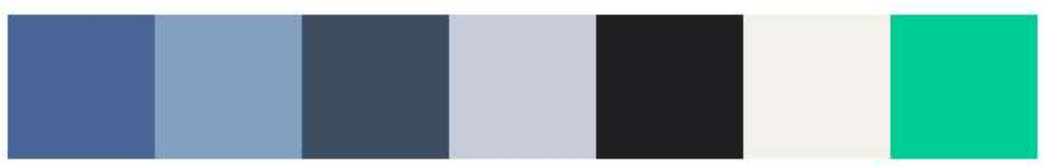
Auto Machine Learning Case Study

In [10]:
```
import plotly.express as px
import plotly.graph_objects as go
import plotly.figure_factory as ff
from plotly.subplots import make_subplots
import matplotlib.pyplot as plt
from colorama import Fore

from pandas_profiling import ProfileReport
import seaborn as sns
from sklearn import metrics
from scipy import stats
import math

from tqdm.notebook import tqdm
from copy import deepcopy

# Installed libraries
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score, f1_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
```

In [11]:
```
# Defining all our palette colours.
primary_blue = "#496595"
primary_blue2 = "#85a1c1"
primary_blue3 = "#3f4d63"
primary_grey = "#c6ccd8"
primary_black = "#202022"
primary_bgcolor = "#f4f0ea"

primary_green = px.colors.qualitative.Plotly[2]

plt.rcParams['axes.facecolor'] = primary_bgcolor

colors = [primary_blue, primary_blue2, primary_blue3, primary_grey, primary_black, primary_bgcolor, primary_green]
sns.palplot(sns.color_palette(colors))
```



In [12]:
```
plt.rcParams['figure.dpi'] = 120
plt.rcParams['axes.spines.top'] = False
plt.rcParams['axes.spines.right'] = False
plt.rcParams['font.family'] = 'serif'
```

In [13]:
```
train_df = pd.read_csv('/input/train.csv')
train_df.columns = [column.lower() for column in train_df.columns]
```

```
# train_df = train_df.drop(columns=['passengerid'])

test_df = pd.read_csv('/input/test.csv')
test_df.columns = [column.lower() for column in test_df.columns]

train_df.head()
```

Out[13]:

	passengerid	survived	pclass	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked
0	0	1	1	Oconnor, Frankie	male	NaN	2	0	209245	27.14	C12239	S
1	1	0	3	Bryan, Drew	male	NaN	0	0	27323	13.35	NaN	S
2	2	0	3	Owens, Kenneth	male	0.33	1	2	CA 457703	71.29	NaN	S
3	3	0	3	Kramer, James	male	19.00	0	0	A. 10866	13.04	NaN	S
4	4	1	3	Bond, Michael	male	25.00	0	0	427635	7.76	NaN	S

In [14]:
```
feature_cols = train_df.drop(['survived', 'passengerid'], axis=1).columns
target_column = 'survived'

## Getting all the data that are not of "object" type.
numerical_columns = ['age', 'fare']
categorical_columns = train_df[feature_cols].drop(columns=numerical_columns).columns

pure_num_cols = train_df[feature_cols].select_dtypes(include=['int64','float64']).columns
pure_cat_cols = train_df[feature_cols].select_dtypes(exclude=['int64','float64']).columns

print(len(numerical_columns), len(categorical_columns))
```

```
2 8
```

H2O AutoML

The H2O AutoML interface is designed to have as few parameters as possible, so all the user needs to do is point to their dataset, identify the response column, and optionally specify a time constraint or limit on the number of total models trained.

In both the R and Python API, AutoML uses the same data-related arguments (x, y, training_frame, validation_frame) as the other H2O algorithms. Most of the time, all you'll need to do is specify the data arguments. You can then configure values for max_runtime_secs and/or max_models to set explicit time or number-of-model limits on your run.

In [15]:
```
train_df = pd.read_csv('/input/train.csv')
train_df.columns = [column.lower() for column in train_df.columns]
# train_df = train_df.drop(columns=['passengerid'])

test_df = pd.read_csv('/input/test.csv')
test_df.columns = [column.lower() for column in test_df.columns]

train_df.head()
```

Out[15]:

	id	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	...	feature_41	feature_42	feature_43	feature_44	feature_45	feature_46	feature_47	feature_48	feature_49	target
0	0	0	0	1	0	1	0	0	0	0	...	0	0	21	0	0	0	0	0	0	Class_2
1	1	0	0	0	0	2	1	0	0	0	...	0	0	0	0	0	0	0	0	0	Class_1
2	2	0	0	0	0	0	0	0	0	0	...	0	1	0	0	0	0	13	2	0	Class_1
3	3	0	0	0	0	0	0	0	3	0	...	0	0	0	0	0	0	0	1	0	Class_4
4	4	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	1	0	Class_2

5 rows × 52 columns

In [16]:
```
feature_columns = train_df.iloc[:, 1:-1].columns.values
target_column = 'target'
feature_columns
```

Out[16]:
```
array(['feature_0', 'feature_1', 'feature_2', 'feature_3', 'feature_4',
       'feature_5', 'feature_6', 'feature_7', 'feature_8', 'feature_9',
       'feature_10', 'feature_11', 'feature_12', 'feature_13',
       'feature_14', 'feature_15', 'feature_16', 'feature_17',
       'feature_18', 'feature_19', 'feature_20', 'feature_21',
       'feature_22', 'feature_23', 'feature_24', 'feature_25',
       'feature_26', 'feature_27', 'feature_28', 'feature_29',
       'feature_30', 'feature_31', 'feature_32', 'feature_33',
       'feature_34', 'feature_35', 'feature_36', 'feature_37',
       'feature_38', 'feature_39', 'feature_40', 'feature_41',
       'feature_42', 'feature_43', 'feature_44', 'feature_45',
       'feature_46', 'feature_47', 'feature_48', 'feature_49'],
      dtype=object)
```

In [17]:
```
import h2o
from h2o.automl import H2OAutoML
```

In [18]:
```
h2o.init()
```

```
Checking whether there is an H2O instance running at http://localhost:54321 ..... not found.
Attempting to start a local H2O server...
  Java Version: openjdk version "11.0.10" 2021-01-19; OpenJDK Runtime Environment (build 11.0.10+9-Ubuntu-0ubuntu1.18.04); OpenJDK 64-Bit Server VM (build 11.0.10+9-Ubuntu-0ubuntu1.18.04, mixed mode, sharing)
  Starting server from /opt/conda/lib/python3.7/site-packages/h2o/backend/bin/h2o.jar
  Ice root: /tmp/tmpfdrq6t1n
  JVM stdout: /tmp/tmpfdrq6t1n/h2o_unknownUser_started_from_python.out
  JVM stderr: /tmp/tmpfdrq6t1n/h2o_unknownUser_started_from_python.err
  Server is running at http://127.0.0.1:54321
Connecting to H2O server at http://127.0.0.1:54321 ... successful.
Warning: Your H2O cluster version is too old (2 years, 7 months and 12 days)! Please download and install the latest version from http://h2o.ai/download/
```

H2O_cluster_uptime:	02 secs
H2O_cluster_timezone:	Etc/UTC
H2O_data_parsing_timezone:	UTC
H2O_cluster_version:	3.32.1.1
H2O_cluster_version_age:	2 years, 7 months and 12 days !!!
H2O_cluster_name:	H2O_from_python_unknownUser_87vj8r
H2O_cluster_total_nodes:	1
H2O_cluster_free_memory:	7.840 Gb
H2O_cluster_total_cores:	4
H2O_cluster_allowed_cores:	4
H2O_cluster_status:	accepting new members, healthy
H2O_connection_url:	http://127.0.0.1:54321
H2O_connection_proxy:	{"http": null, "https": null}
H2O_internal_security:	False
H2O_API_Extensions:	Amazon S3, XGBoost, Algos, AutoML, Core V3, TargetEncoder, Core V4
Python_version:	3.7.9 final

In [19]:
```
train_hf = h2o.H2OFrame(train_df.copy())
test_hf = h2o.H2OFrame(test_df.copy())
```

```
Parse progress: |████████████████████████████████████████████████| 100%
Parse progress: |████████████████████████████████████████████████| 100%
```

In [20]:
```
train_hf[target_column] = train_hf[target_column].asfactor()
```

In [21]:
```
%%time
aml = H2OAutoML(
    seed=2021,
    max_runtime_secs=100,
    nfolds = 3,
    exclude_algos = ["DeepLearning"]
)

aml.train(
    x=list(feature_columns),
    y=target_column,
    training_frame=train_hf
)
```

```
AutoML progress: |████████████████████████████████████████████████| 100%
CPU times: user 24.2 s, sys: 425 ms, total: 24.6 s
Wall time: 2min 37s
```

In [22]:
```
lb = aml.leaderboard
lb.head(rows = lb.nrows)
```

	model_id	mean_per_class_error	logloss	rmse	mse	auc	aucpr
	XGBoost_2_AutoML_20231106_230711	0.742573	1.27942	0.719032	0.517007	nan	nan
	DRF_1_AutoML_20231106_230711	0.744102	4.01692	0.657657	0.432513	nan	nan
	XGBoost_1_AutoML_20231106_230711	0.744133	1.27617	0.718288	0.515938	nan	nan
	StackedEnsemble_AllModels_AutoML_20231106_230711	0.744543	1.10319	0.629252	0.395959	nan	nan
	GBM_4_AutoML_20231106_230711	0.745547	1.32745	0.734207	0.53906	nan	nan
	XGBoost_grid__1_AutoML_20231106_230711_model_1	0.745624	1.16831	0.674476	0.454918	nan	nan
	GBM_3_AutoML_20231106_230711	0.747009	1.32768	0.734288	0.539178	nan	nan
	GBM_5_AutoML_20231106_230711	0.747019	1.32731	0.734184	0.539026	nan	nan
	XRT_1_AutoML_20231106_230711	0.747236	3.47468	0.650249	0.422824	nan	nan
	GBM_2_AutoML_20231106_230711	0.748286	1.32794	0.734365	0.539291	nan	nan
	GBM_1_AutoML_20231106_230711	0.748447	1.32825	0.734457	0.539427	nan	nan
	XGBoost_3_AutoML_20231106_230711	0.748685	1.27634	0.718529	0.516283	nan	nan
	StackedEnsemble_BestOfFamily_AutoML_20231106_230711	0.748742	1.10746	0.631129	0.398324	nan	nan
	GLM_1_AutoML_20231106_230711	0.75	1.11666	0.634631	0.402757	nan	nan

Out[22]:

In [23]:
```
%%time

preds = aml.predict(h2o.H2OFrame(test_df[feature_columns].copy()))
preds_df = h2o.as_list(preds)
preds_df

submission[['Class_1', 'Class_2', 'Class_3', 'Class_4']] = preds_df[['Class_1', 'Class_2', 'Class_3', 'Class_4']]
submission.to_csv('h2o_automl_300s.csv', index=False)
submission.head()
```

```
Parse progress: |████████████████████████████████████████████████| 100%
xgboost prediction progress: |████████████████████████████████████| 100%
CPU times: user 1.46 s, sys: 68.6 ms, total: 1.53 s
Wall time: 3 s
```

Out[23]:

	id	Class_1	Class_2	Class_3	Class_4
0	100000	0.207323	0.294891	0.287164	0.210622
1	100001	0.204417	0.363643	0.216650	0.215290
2	100002	0.221544	0.308985	0.253071	0.216401
3	100003	0.190484	0.290091	0.286541	0.232883
4	100004	0.202363	0.367720	0.214957	0.214960

Titanic

In [24]:
```
train_df = pd.read_csv('/input/titanic/train.csv')
train_df.columns = [column.lower() for column in train_df.columns]
# train_df = train_df.drop(columns=['passengerid'])

test_df = pd.read_csv('/input/titanic/test.csv')
test_df.columns = [column.lower() for column in test_df.columns]


feature_columns = train_df.drop(['survived', 'passengerid'], axis=1).columns
target_column = 'survived'

train_hf = h2o.H2OFrame(train_df.copy())
test_hf = h2o.H2OFrame(test_df.copy())

train_df.head()
```

```
Parse progress: |████████████████████████████████████████████████| 100%
Parse progress: |████████████████████████████████████████████████| 100%
```

Out[24]:

	passengerid	survived	pclass	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

In [25]:
```
%%time
aml = H2OAutoML(
    seed=2021,
    max_runtime_secs=100,
    nfolds = 3,
    exclude_algos = ["DeepLearning"]
)

aml.train(
    x=list(feature_columns),
    y=target_column,
    training_frame=train_hf
)
```

```
AutoML progress: |████████████████████████████████████████████████| 100%
CPU times: user 10.3 s, sys: 186 ms, total: 10.5 s
Wall time: 42.8 s
```

In [26]:
```
lb = aml.leaderboard
lb.head(rows = lb.nrows)
```

model_id	mean_residual_deviance	rmse	mse	mae	rmsle
StackedEnsemble_BestOfFamily_AutoML_20231106_230952	0.125677	0.35451	0.125677	0.253959	0.249667
StackedEnsemble_AllModels_AutoML_20231106_230952	0.126337	0.355439	0.126337	0.251808	0.250137
GBM_1_AutoML_20231106_230952	0.129217	0.359468	0.129217	0.253349	0.254382
GBM_4_AutoML_20231106_230952	0.129459	0.359803	0.129459	0.25589	0.252975
GBM_3_AutoML_20231106_230952	0.129721	0.360168	0.129721	0.25219	0.255879
GBM_grid__1_AutoML_20231106_230952_model_10	0.129943	0.360476	0.129943	0.262969	0.254905
GBM_2_AutoML_20231106_230952	0.131887	0.363162	0.131887	0.256498	0.258099
GBM_grid__1_AutoML_20231106_230952_model_21	0.132578	0.364112	0.132578	0.247089	0.256364
GBM_grid__1_AutoML_20231106_230952_model_22	0.133039	0.364745	0.133039	0.254004	0.257425
GBM_grid__1_AutoML_20231106_230952_model_35	0.133793	0.365777	0.133793	0.266688	0.257974
DRF_1_AutoML_20231106_230952	0.134702	0.367018	0.134702	0.253922	0.260008
GBM_grid__1_AutoML_20231106_230952_model_27	0.134707	0.367025	0.134707	0.26481	0.260165
XGBoost_grid__1_AutoML_20231106_230952_model_6	0.135047	0.367488	0.135047	0.252359	0.260699
XGBoost_grid__1_AutoML_20231106_230952_model_8	0.135755	0.36845	0.135755	0.254922	0.260492
XGBoost_grid__1_AutoML_20231106_230952_model_1	0.136158	0.368995	0.136158	0.252117	0.262131
XGBoost_grid__1_AutoML_20231106_230952_model_7	0.136158	0.368995	0.136158	0.259794	0.261302
XGBoost_grid__1_AutoML_20231106_230952_model_4	0.137077	0.370239	0.137077	0.280213	0.261843
GBM_grid__1_AutoML_20231106_230952_model_16	0.137429	0.370714	0.137429	0.237524	0.260993
XGBoost_grid__1_AutoML_20231106_230952_model_3	0.13775	0.371147	0.13775	0.268057	0.263902
GBM_grid__1_AutoML_20231106_230952_model_14	0.137923	0.37138	0.137923	0.294885	0.262581
XGBoost_3_AutoML_20231106_230952	0.138467	0.372111	0.138467	0.255043	0.263709
GBM_grid__1_AutoML_20231106_230952_model_8	0.138483	0.372133	0.138483	0.280151	0.263346
GBM_grid__1_AutoML_20231106_230952_model_20	0.139638	0.373682	0.139638	0.291814	0.264653
GBM_grid__1_AutoML_20231106_230952_model_24	0.139972	0.374129	0.139972	0.298699	0.263499
XRT_1_AutoML_20231106_230952	0.140359	0.374646	0.140359	0.303279	0.271413
GBM_grid__1_AutoML_20231106_230952_model_15	0.14137	0.375991	0.14137	0.293674	0.265485
GBM_grid__1_AutoML_20231106_230952_model_25	0.141377	0.376002	0.141377	0.28426	0.266103
GBM_grid__1_AutoML_20231106_230952_model_30	0.142308	0.377238	0.142308	0.262392	0.265581
GBM_grid__1_AutoML_20231106_230952_model_5	0.143354	0.378621	0.143354	0.302222	0.267025
GBM_grid__1_AutoML_20231106_230952_model_17	0.143691	0.379066	0.143691	0.303773	0.267633
GBM_grid__1_AutoML_20231106_230952_model_6	0.143728	0.379115	0.143728	0.307901	0.268187
GBM_grid__1_AutoML_20231106_230952_model_23	0.144182	0.379713	0.144182	0.304103	0.268073
GBM_grid__1_AutoML_20231106_230952_model_13	0.144886	0.380638	0.144886	0.314363	0.268356
GLM_1_AutoML_20231106_230952	0.145709	0.381718	0.145709	0.301761	0.270041

model_id	mean_residual_deviance	rmse	mse	mae	rmsle
GBM_grid__1_AutoML_20231106_230952_model_28	0.146182	0.382338	0.146182	0.304743	0.270696
GBM_grid__1_AutoML_20231106_230952_model_4	0.146307	0.382501	0.146307	0.303583	0.270572
GBM_grid__1_AutoML_20231106_230952_model_31	0.14631	0.382504	0.14631	0.305923	0.270706
XGBoost_2_AutoML_20231106_230952	0.147143	0.383592	0.147143	0.276879	0.275316
GBM_grid__1_AutoML_20231106_230952_model_18	0.150136	0.387474	0.150136	0.310458	0.273608
GBM_grid__1_AutoML_20231106_230952_model_29	0.150209	0.387569	0.150209	0.318821	0.273794
GBM_grid__1_AutoML_20231106_230952_model_3	0.150307	0.387695	0.150307	0.316653	0.273923
GBM_5_AutoML_20231106_230952	0.150385	0.387795	0.150385	0.314272	0.274274
GBM_grid__1_AutoML_20231106_230952_model_2	0.150643	0.388128	0.150643	0.309619	0.275236
XGBoost_grid__1_AutoML_20231106_230952_model_2	0.151704	0.389492	0.151704	0.25465	0.275842
GBM_grid__1_AutoML_20231106_230952_model_9	0.151876	0.389713	0.151876	0.318428	0.275713
GBM_grid__1_AutoML_20231106_230952_model_26	0.152377	0.390355	0.152377	0.322933	0.275912
GBM_grid__1_AutoML_20231106_230952_model_34	0.152437	0.390432	0.152437	0.317441	0.276961
GBM_grid__1_AutoML_20231106_230952_model_1	0.152519	0.390536	0.152519	0.328751	0.276047
GBM_grid__1_AutoML_20231106_230952_model_7	0.152742	0.390822	0.152742	0.288468	0.279303
GBM_grid__1_AutoML_20231106_230952_model_11	0.152815	0.390915	0.152815	0.329026	0.276465
GBM_grid__1_AutoML_20231106_230952_model_12	0.153264	0.391489	0.153264	0.328992	0.276906
GBM_grid__1_AutoML_20231106_230952_model_33	0.153782	0.39215	0.153782	0.311488	0.277455
XGBoost_1_AutoML_20231106_230952	0.155705	0.394595	0.155705	0.275155	0.284305
GBM_grid__1_AutoML_20231106_230952_model_19	0.157208	0.396494	0.157208	0.315118	0.280904
XGBoost_grid__1_AutoML_20231106_230952_model_5	0.158321	0.397896	0.158321	0.259294	0.283856
GBM_grid__1_AutoML_20231106_230952_model_32	0.163415	0.404247	0.163415	0.351727	0.284976

Out[26]:

MLJAR AutoML

MLJAR is an Automated Machine Learning framework.

The MLJAR AutoML can work in several modes:

- **Explain** - ideal for initial data exploration
- **Perform** - perfect for production-level ML systems
- **Compete** - mode for ML competitions under restricted time budget. By the default, it performs advanced feature engineering like golden features search, kmeans features, feature selection. It does model stacking.
- **Optuna** - uses Optuna to highly tune algorithms: `Random Forest`, `Extra Trees`, `Xgboost`, `LightGBM`, `CatBoost`, `Neural Network`. Each algorithm is tuned with Optuna hperparameters framework with selected time budget (controlled with optuna_time_budget). By the default feature engineering is not enabled (you need to manually swtich it on, in AutoML() parameter).

In [27]:
```
%%script false --no-raise-error
!apt-get install -y build-essential python3-dev
```

In [28]:
```
%%script false --no-raise-error
!pip -q install pip --upgrade
!pip install graphviz --upgrade
!pip install dtreeviz
!pip install mljar-supervised
```

In [29]:
```
%%script false --no-raise-error
from supervised.automl import AutoML # mljar-supervised
```

In [30]:
```
%%script false --no-raise-error
%%time
automl = AutoML(
    mode="Compete",
    eval_metric="f1",
    total_time_limit=300,
    features_selection=False # switch off feature selection
)
automl.fit(
    train[feature_cols],
    train[target_column]
)
```

In [31]:
```
%%script false --no-raise-error
%%time
preds = automl.predict(test[feature_cols])

submission['Survived'] = preds
submission.to_csv('mljar_automl_300s_f1_metric.csv', index=False)
submission.head()
```

PyCaret

In [32]:
```
%%script false --no-raise-error
!pip install pycaret
```

In [33]:
```
%%script false --no-raise-error
from pycaret.classification import *
```

In [34]:
```
%%script false --no-raise-error
from category_encoders.cat_boost import CatBoostEncoder

cat_train_df = train_df.copy()
cat_test_df = test_df.copy()

ce = CatBoostEncoder()

cols_to_encode = ['name', 'sex', 'ticket', 'cabin', 'embarked']
cat_train_df[pure_cat_cols] = ce.fit_transform(cat_train_df[pure_cat_cols], cat_train_df[target_column])
cat_test_df[pure_cat_cols] = ce.transform(cat_test_df[pure_cat_cols])
```

In [35]:
```
%%script false --no-raise-error
setup(
    data = cat_train_df[feature_cols.to_list() + [target_column]],
    target = target_column,
    fold = 3,
```

```
    silent = True,
)
```

In [36]:
```
%%script false --no-raise-error
%%time
best_models = compare_models(
    sort='F1',
    n_select=3,
    budget_time=300,
) # we will use it later
```

In [37]:
```
%%script false --no-raise-error
# select best model
best = automl(optimize = 'F1')
```

In [38]:
```
%%script false --no-raise-error
plot_model(best, plot = 'confusion_matrix')

plot_model(best, plot = 'feature_all')
```

EvalML: AutoML

EvalML is an AutoML library which builds, optimizes, and evaluates machine learning pipelines using domain-specific objective functions.

Key Functionality

- **Automation** - Makes machine learning easier. Avoid training and tuning models by hand. Includes data quality checks, cross-validation and more.
- **Data Checks** - Catches and warns of problems with your data and problem setup before modeling.
- **End-to-end** - Constructs and optimizes pipelines that include state-of-the-art preprocessing, feature engineering, feature selection, and a variety of modeling techniques.
- **Model Understanding** - Provides tools to understand and introspect on models, to learn how they'll behave in your problem domain.
- **Domain-specific** - Includes repository of domain-specific objective functions and an interface to define your own.

In [39]:
```
%%script false --no-raise-error
!pip install evalml
```

In [40]:
```
%%script false --no-raise-error
from evalml.automl import AutoMLSearch
```

In [41]:
```
%%script false --no-raise-error
X = train_df.drop(columns=[target_column, 'passengerid'])
y = train_df[target_column]

X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.2)
```

In [42]:
```
%%script false --no-raise-error
%%time
automl = AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    problem_type='binary',
    random_seed=2021,
    max_time=300,
)
```

In [43]:
```
%%script false --no-raise-error
automl.search()
```

In [44]:
```
%%script false --no-raise-error
automl.rankings
```

In [45]:
```
%%script false --no-raise-error
%%time
pipeline = automl.best_pipeline
pipeline.fit(X, y)
```

In [46]:
```
%%script false --no-raise-error
preds = pipeline.predict(test_df.drop([target_column, 'passengerid'], axis=1))

submission['Survived'] = preds.to_series().astype(int)
submission.to_csv('evalml_automl_300s_f1_metric.csv', index=False)
submission.head()
```

TPOT: Genetic Approach

In [47]:
```
%%script false --no-raise-error
!pip install tpot
```

In [48]:
```
%%script false --no-raise-error
from category_encoders.cat_boost import CatBoostEncoder

cat_train_df = train_df.copy()
cat_test_df = test_df.copy()

ce = CatBoostEncoder()

cols_to_encode = ['name', 'sex', 'ticket', 'cabin', 'embarked']
cat_train_df[pure_cat_cols] = ce.fit_transform(cat_train_df[pure_cat_cols], cat_train_df[target_column])
cat_test_df[pure_cat_cols] = ce.transform(cat_test_df[pure_cat_cols])
```

In [49]:
```
%%script false --no-raise-error
X = cat_train_df.drop(columns=[target_column, 'passengerid'])
y = cat_train_df[target_column]

X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.2, random_state=2021)
```

In [50]:
```
%%script false --no-raise-error

from tpot import TPOTClassifier
from sklearn.model_selection import train_test_split

tpot = TPOTClassifier(generations=5, population_size=50, verbosity=2, random_state=42)
tpot.fit(X_train, y_train)
print(tpot.score(X_test, y_test))
tpot.export('tpot_digits_pipeline.py')
```

FLAML: Fast and Lightweight AutoML

FLAML is a lightweight Python library that automatically, efficiently, and economically discovers accurate machine learning models. It liberates users from the burdens of selecting learners and their associated hyperparameters. Notably fast and cost-effective, its uncomplicated, lightweight design allows for easy extension, such as incorporating customized learners or metrics. Powered by a novel, cost-effective hyperparameter optimization and learner selection method developed by Microsoft Research, FLAML utilizes the search space's structure to select an order optimized for both cost and error.

For instance, the system initially suggests economical configurations in the search process but swiftly transitions to more complex configurations with larger sample sizes as needed in the later stages. Additionally, it prioritizes inexpensive learners at the outset but penalizes them later if there is slow error improvement. This cost-bounded search and cost-based prioritization significantly enhance search efficiency when operating under budget constraints.

In [51]:
```
%%script false --no-raise-error
!pip install flaml
```

In [52]:
```
%%script false --no-raise-error
from flaml import AutoML
from sklearn.datasets import load_boston
```

In [53]:
```
%%script false --no-raise-error
# Initialize an AutoML instance
automl = AutoML()

# Specify automl goal and constraint
automl_settings = {
    "time_budget": 300,  # in seconds
    "metric": 'accuracy',
    "task": 'classification',
}
```

In [54]:
```
%%script false --no-raise-error
# Train with labeled input data
automl.fit(
    X_train=train_df[feature_cols],
    y_train=train_df[target_column],
    **automl_settings
)
```

In [55]:
```
%%script false --no-raise-error
# Predict
print(automl.predict_proba(train_df[feature_cols]))
# Export the best model
print(automl.model)
```

Simple regression problem

In [56]:
```
%%script false --no-raise-error

# Initialize an AutoML instance
automl = AutoML()

# Specify automl goal and constraint
automl_settings = {
    "time_budget": 10,  # in seconds
    "metric": 'r2',
    "task": 'regression',
    "log_file_name": "test/boston.log",
}
```

```
X_train, y_train = load_boston(return_X_y=True)
# Train with labeled input data
automl.fit(X_train=X_train, y_train=y_train,
                        **automl_settings)

# Predict
print(automl.predict(X_train))
# Export the best model
print(automl.model)
```