Summary Dataset Report For Single Dataset

March 13, 2023

[44]:
```
# imports

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np


from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, ParameterGrid
import sklearn.metrics as metrics
from sklearn.metrics import plot_roc_curve
from sklearn.metrics import classification_report
from sklearn.model_selection import StratifiedKFold, KFold
from sklearn.metrics import auc

from tqdm import tqdm # pretty progress bar
import weles as ws

import seaborn as sn
import matplotlib.cm as cm

# setup
plt.style.use('default')
```

1 Load Dataset

[45]:
```
# Set label column name in the dataset -> it must be lowercase due to FET module
LABEL="is_doh"
# Set if your dataset is multiclass or not
MULTICLASS=False
```

[46]:
```
df_dataset = "/srv/data/qod/paper/test2/combined-doh-http.csv"
```

[47]:
```
df_dataset = pd.read_csv(df_dataset, delimiter=",")
```

[48]:

```
df_dataset = df_dataset.sample(frac=1).reset_index(drop=True)
```

[49]:

```
# NOTE: label column must be lowercase due to FET module
#df_dataset.rename(columns = {'Label':'label'}, inplace = True)

df_dataset.replace([np.inf, -np.inf], np.nan, inplace=True)
df_dataset = df_dataset.dropna()
```

[50]:

```
# NOTE: change label in the dataset
y_1 = df_dataset [LABEL]
X_1 = df_dataset.drop(columns=[LABEL])
y_1 = y_1.astype('category')
y_1 = y_1.cat.codes
```

2 Get Init Info

[51]:

```
print(df_dataset[LABEL].value_counts())
```

```
True     5000
False    5000
Name: is_doh, dtype: int64
```

[52]:

```
# Stats
print("Num. Features:",len(X_1.columns))
print("Total Size:",len(X_1))
print("Duplicated Flows:",X_1[X_1.duplicated()].shape[0])
```

```
Num. Features: 24
Total Size: 10000
Duplicated Flows: 1
```

[53]:

```
from fet.explorer import Explorer
# NOTE: label column must be lowercase
e = Explorer(y=LABEL)
```

[54]:

```
e.df = df_dataset
e.feature_cols = list(X_1.columns) #feature_cols
```

[55]:

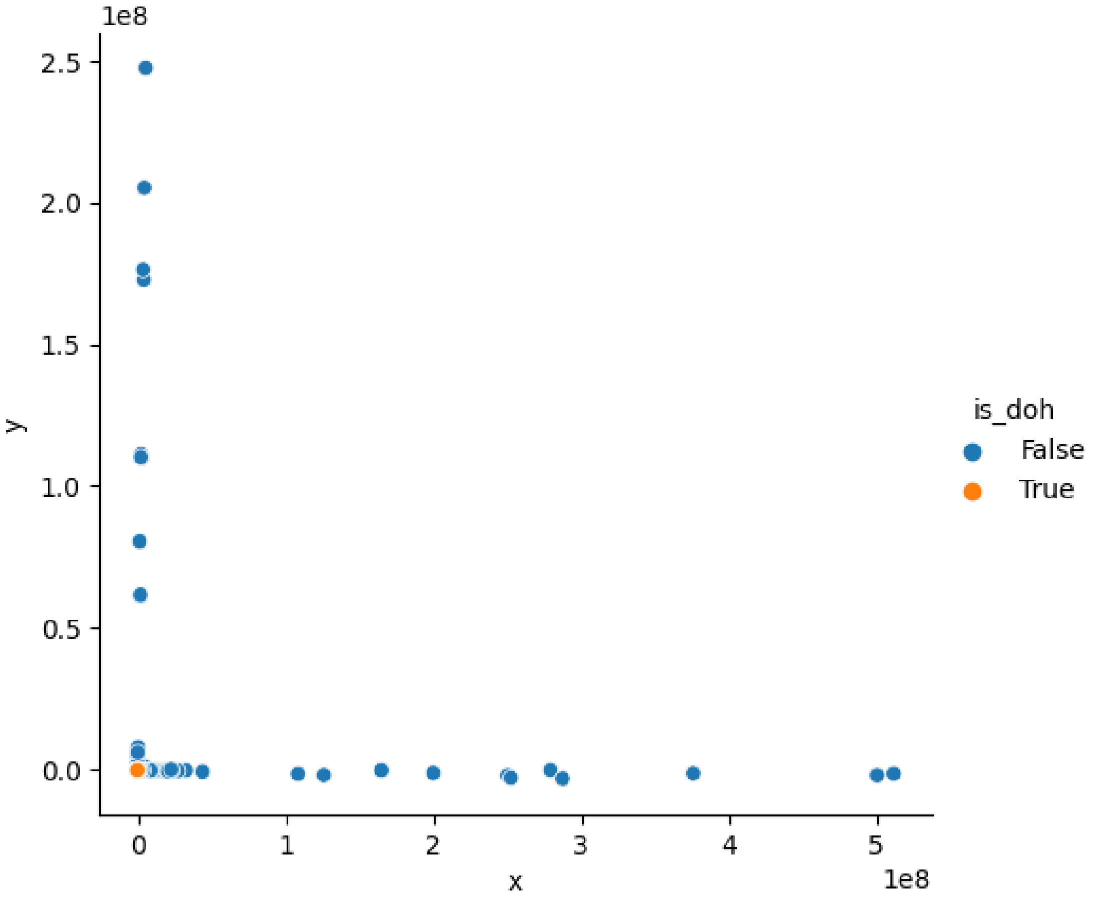
```
e.feature_scores()[:10]
```

```
/home/netmon/.local/lib/python3.6/site-
packages/sklearn/feature_selection/_univariate_selection.py:115: UserWarning:
Features [17 18] are constant.
  UserWarning)
/home/netmon/.local/lib/python3.6/site-
packages/sklearn/feature_selection/_univariate_selection.py:116: RuntimeWarning:
```

```
invalid value encountered in true_divide
  f = msb / msw
```

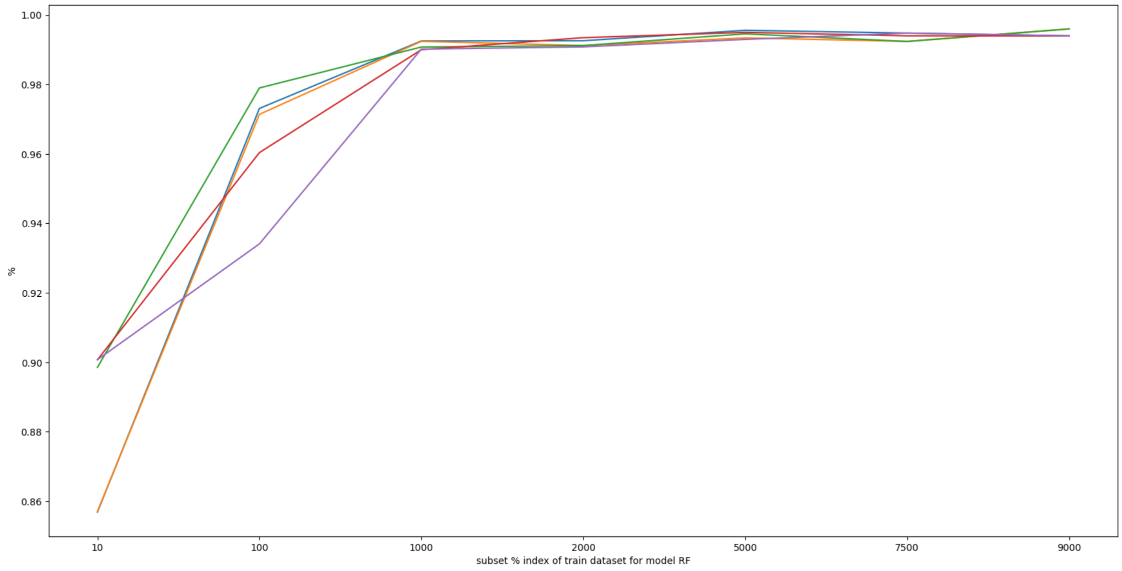
```
[55]: [('var_pkt_size', 2752.500279430559),
       ('median_pkt_size', 1684.8132130723964),
       ('av_pkt_size', 1660.3294836274442),
       ('av_pkt_size_rev', 1099.7064546623874),
       ('var_pkt_size_rev', 745.1159149408735),
       ('median_pkt_size_rev', 587.3424346051345),
       ('bytes_ration', 64.88735001004468),
       ('packets_sum', 16.471173618360513),
       ('packets', 15.802626477813774),
       ('packets_rev', 13.284961913188633)]
```

```
[19]: e.plot_pca()
```



3 Test 1: Dataset Redundancy

[ ]:
```
## Description: This test verifies the amount of redundat data for selected␣
↪classification model to provide the comparative results as complete dataset

## Input:
### - X_1: Dataset features part
### - y_1: Dataset label part
### - runs: Number of iterations for random sampling
### - alfa: Level of acceptance
### - metric: Performance metric for ML classifiers
### - clfs: Pool of ML classifiers

## Output:
### - perc: First percentage level that work sufficiently for all "runs"
### - model: List of successful models for selected "perc"
### - score: Inverse of "perc" (final metric)
```



[60]:
```
def eval_dataset(X_1, y_1, frac):
    tmp_results = {}

    for name, clf in clfs.items():
        l = []
        tmp_results[name] = [l.copy() for i in range(runs)]

    for i in range(0,runs):
        X_train_sub, X_test_sub, y_train_sub, y_test_sub =␣
↪train_test_split(X_1, y_1, test_size=1-frac, stratify=y_1, shuffle=True)
        for name, clf in clfs.items():
            clf.fit(X_train_sub, y_train_sub)
            pred = clf.predict(X_test_sub)
            tmp_results[name][i].append(metrics.f1_score(y_test_sub,pred))
    return tmp_results
```

[61]:

```
from sklearn.metrics import precision_score, f1_score, recall_score
from imblearn.metrics import sensitivity_score, specificity_score

runs = 5 # number of iterations
alfa = 0.01 # Lift Value
metric =  f1_score
```

[69]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from xgboost import XGBClassifier

if MULTICLASS:
    clfs = {
            #"KNN": KNeighborsClassifier(),
            #"SVM": SVC(), #long time of processing; SVM is moved to the␣
 ↪separated notebook
            #"GNB": GaussianNB(),
            "DT": DecisionTreeClassifier(criterion = "gini"),
            "RF": RandomForestClassifier(class_weight="balanced",␣
 ↪criterion='gini'),
            #"AB": AdaBoostClassifier(),
            "XGB": XGBClassifier(objective="multi:softmax"),
            #"MLP": MLPClassifier(max_iter=1000, hidden_layer_sizes=5,␣
 ↪batch_size=100)
            #"DT": DecisionTreeClassifier(class_weight="balanced"),
            #"XGB": XGBClassifier(use_label_encoder=False, objective="binary:
 ↪logistic",eval_metric="logloss")
    }
else:
    clfs = {
        #"KNN": KNeighborsClassifier(),
        #"SVM": SVC(), #long time of processing; SVM is moved to the separated␣
 ↪notebook
        #"GNB": GaussianNB(),
        "DT": DecisionTreeClassifier(),
        "RF": RandomForestClassifier(),
        #"AB": AdaBoostClassifier(),
        "XGB": XGBClassifier(use_label_encoder=False, eval_metric="logloss"),
        #"MLP": MLPClassifier(max_iter=1000, hidden_layer_sizes=5,␣
↪batch_size=100)
        #"DT": DecisionTreeClassifier(class_weight="balanced"),
        #"XGB": XGBClassifier(use_label_encoder=False, objective="binary:
↪logistic",eval_metric="logloss")
}
```

[64]:

```
results = eval_dataset(X_1, y_1, 0.9)
#print(results)
max_score = 0
for name, clf in clfs.items():
    for i in range(runs):
        if max_score < results[name][i][0]:
            max_score = results[name][i][0]
print("Max score", max_score)
```

```
Max score 0.9870388833499502
```

[65]:

```
# TODO make paralel
limit = max_score*alfa
low = 0.01
high = 0.99
mid = 0
tmp_redundancy = 0.9

while high - low > alfa:
    print("Testing redundancy with",high, low, tmp_redundancy)
    tmp_redundancy = (high + low) / 2
    tmp_score = eval_dataset(X_1, y_1, tmp_redundancy)
    tmp_high = []
    tmp_low = []
    for name, clf in clfs.items():
        tmp = []
        for item in range(runs):
            if (max_score - tmp_score[name][item][0]) < limit:
                tmp.append(tmp_score[name][item][0])

        if len(tmp) == runs:
            tmp_high.append(tmp_redundancy)
        else:
            tmp_low.append(tmp_redundancy)
    if len(tmp_high) > 0:
        high = tmp_redundancy
    else:
        low = tmp_redundancy
```

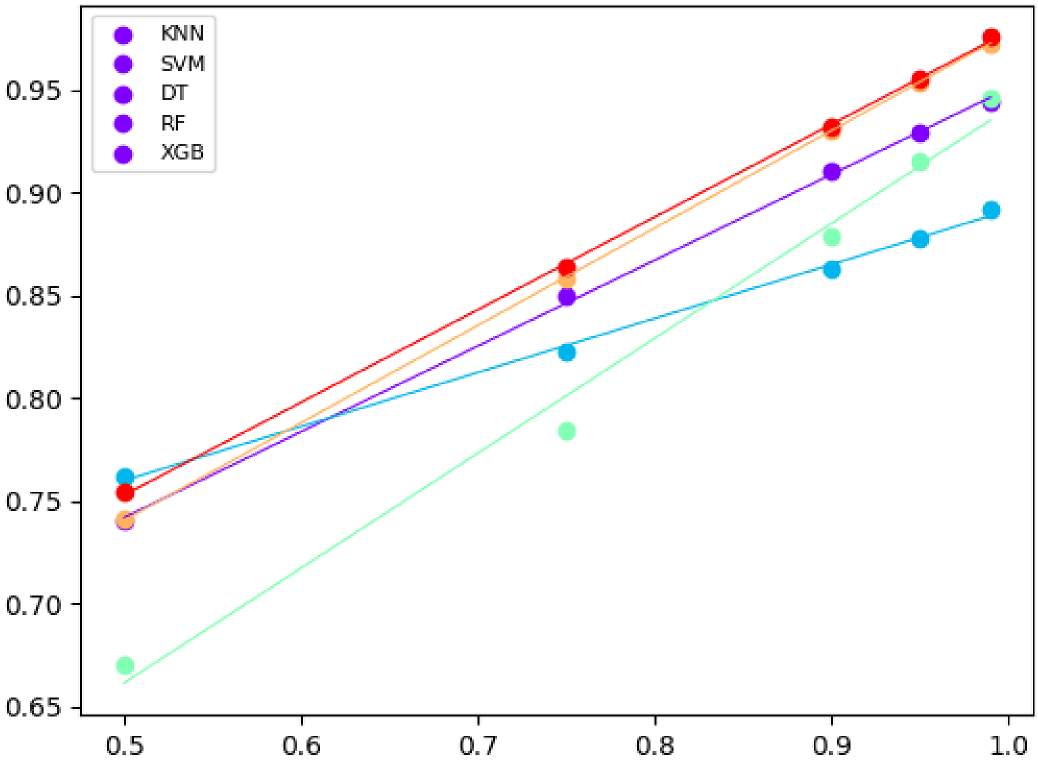
```
print("Dataset redundancy",1-tmp_redundancy)
```

```
Testing redundancy with 0.99 0.01 0.9
Testing redundancy with 0.99 0.5 0.5
Testing redundancy with 0.99 0.745 0.745
Testing redundancy with 0.8674999999999999 0.745 0.8674999999999999
Testing redundancy with 0.8062499999999999 0.745 0.8062499999999999
Testing redundancy with 0.775625 0.745 0.775625
Testing redundancy with 0.775625 0.7603125 0.7603125
Dataset redundancy 0.23203125000000002
```

```
[ ]:
```

4 Test 2: Dataset Association Quality

```
[ ]: ## Description: This test verifes the level of relantionship between feature␣
      ↪data and respective labels. The evaluation is enhanced for␣
      ↪multiclassification and also the final metric is providing more reliable␣
      ↪results.

     ## Input:
     ### - X_1: Dataset features part
     ### - y_1: Dataset label part
     ### - nperm: Number of permutation iterations
     ### - perc: Levels of selections for training part
     ### - metric: Performance metric for ML classifiers
     ### - clfs: Pool of ML classifiers

     # Output:
     ### - ML model with highest slope and AUC ... TBD ...
```



[66]:

```
from sklearn.preprocessing import MinMaxScaler

X_1 = MinMaxScaler().fit_transform(X_1)
datasets = {
    "all": (X_1, y_1)
}
```

[67]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from xgboost import XGBClassifier
from sklearn.metrics import f1_score

if MULTICLASS:
    clfs = {
            #"KNN": KNeighborsClassifier(),
            #"SVM": SVC(), #long time of processing; SVM is moved to the␣
 ↪separated notebook
            #"GNB": GaussianNB(),
            "DT": DecisionTreeClassifier(criterion = "gini"),
            "RF": RandomForestClassifier(class_weight="balanced",␣
 ↪criterion='gini'),
            #"AB": AdaBoostClassifier(),
            "XGB": XGBClassifier(objective="multi:softmax"),
            #"MLP": MLPClassifier(max_iter=1000, hidden_layer_sizes=5,␣
 ↪batch_size=100)
            #"DT": DecisionTreeClassifier(class_weight="balanced"),
            #"XGB": XGBClassifier(use_label_encoder=False, objective="binary:
 ↪logistic",eval_metric="logloss")
    }
else:
    clfs = {
        #"KNN": KNeighborsClassifier(),
        #"SVM": SVC(), #long time of processing; SVM is moved to the separated␣
 ↪notebook
        #"GNB": GaussianNB(),
        "DT": DecisionTreeClassifier(),
        "RF": RandomForestClassifier(),
        #"AB": AdaBoostClassifier(),
        "XGB": XGBClassifier(use_label_encoder=False, eval_metric="logloss"),
        #"MLP": MLPClassifier(max_iter=1000, hidden_layer_sizes=5,␣
 ↪batch_size=100)
        #"DT": DecisionTreeClassifier(class_weight="balanced"),
        #"XGB": XGBClassifier(use_label_encoder=False, objective="binary:
 ↪logistic",eval_metric="logloss")
}


metrics = {
    "F1": f1_score
}
```

[ ]:

```
#TODO Replace weles to speedup and remove python version dependency
#Improve implementation towards ALF
ev = ws.evaluation.Evaluator(datasets=datasets, protocol2=(False, 2, None)).
 ↪process(clfs=clfs, verbose=0)

scores = ev.score(metrics=metrics)
```

[ ]:

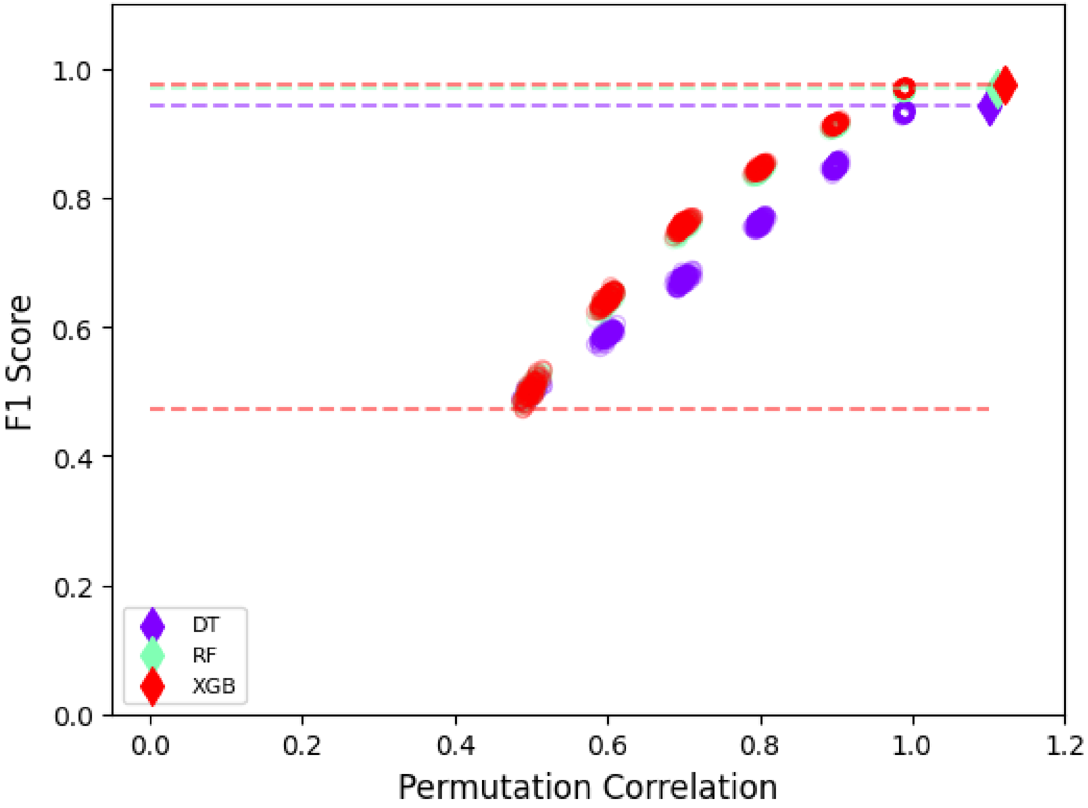
```
nperm = 100 # number of permutations
perc = [50, 40, 30, 20, 10, 1] # percentage of permutation
a=np.shape(ev.scores.mean(axis=2)[:, :, 0]) # true result

perm = np.zeros((nperm,len(perc),a[1]))
corr = np.zeros((nperm,len(perc)))
```

```python
for i in range(nperm):
    for j in range(len(perc)):

        print(i,j)
        t=0
        while True:
            # TODO customize permutation based on input amount of classes
            ind1=np.where(y_1 == 0)
            ind2=np.where(y_1 == 1)
            ind3=np.where(y_1 == 2)
            ind4=np.where(y_1 == 3)
            ind5=np.where(y_1 == 4)
            ind6=np.where(y_1 == 5)

            nperc1 = round(perc[j]*len(ind1[0])/100)
            nperc2 = round(perc[j]*len(ind2[0])/100)
            nperc3 = round(perc[j]*len(ind3[0])/100)
            nperc4 = round(perc[j]*len(ind4[0])/100)
            nperc5 = round(perc[j]*len(ind5[0])/100)
            nperc6 = round(perc[j]*len(ind6[0])/100)

            indP = np.random.permutation(np.concatenate((ind1[0][:nperc1], ind2[0][:nperc2], ind3[0][:nperc3], ind4[0][:nperc4], ind5[0][:nperc5], ind6[0][:nperc6])))
            ind = np.sort(indP);

            y1P = np.copy(y_1);

            y1P[ind] = y_1[indP];

            comparison = y_1 == y1P

            if not comparison.all() or t > 3:
                print(t)
                break
            t += 1

        datasetsP = {
          "all": (X_1, y1P)
        }

        evP = ws.evaluation.Evaluator(datasets=datasetsP,protocol2=(False, 2, None)).process(clfs=clfs, verbose=0)

        scores = evP.score(metrics=metrics)

        perm[i,j,:] = evP.scores.mean(axis=2)[:, :, 0]
        kk = np.corrcoef(y1P,y_1)
        corr[i,j] = kk[0,1]
```

[75]:

```
import matplotlib.cm as cm

pvalues = np.zeros((a[1],len(perc)))

colors = cm.rainbow(np.linspace(0, 1, a[1]))
# plot true values as diamonds
for i, c in zip(range(a[1]),colors):
    plt.scatter(1.1+i*0.01, ev.scores.mean(axis=2)[:, i, 0], s=100, color=c,␣
 ↪marker='d')

plt.legend(("DT","RF","XGB"), prop={'size': 8})
#loc='lower right'

# plot lines for true values
for i, c in zip(range(a[1]),colors):
    plt.plot([0, 1.1+i*0.01], [ev.scores.mean(axis=2)[:, i, 0], ev.scores.
 ↪mean(axis=2)[:, i, 0]], c=c, linestyle='dashed', alpha=0.5)

# plot permutations
colors = cm.rainbow(np.linspace(0, 1, a[1]))
for j in range(len(perc)):
    for i, c in zip(range(a[1]),colors):
        ind = np.where(perm[:,j,i]<ev.scores.mean(axis=2)[:, i, 0])
        plt.scatter((corr[ind,j]), perm[ind,j,i], color="none", edgecolor=c,␣
 ↪alpha=0.3)

for j in range(len(perc)):
    for i, c in zip(range(a[1]),colors):
        ind = np.where(perm[:,j,i]>=ev.scores.mean(axis=2)[:, i, 0])
        plt.scatter((corr[ind,j]), perm[ind,j,i], color=c, edgecolor="black",␣
 ↪alpha=1)
        pvalues[i,j] = ((len(ind[0])+1)*1.0)/(nperm+1);

plt.ylabel('F1 Score', size=12)
plt.xlabel('Permutation Correlation', size=12)

plt.plot([0, 1.1], [perm.min(), perm.min()], color='red', linestyle='dashed',␣
 ↪alpha=0.5)

plt.axis([-0.05, 1.2, 0, 1.1])


plt.show()
```

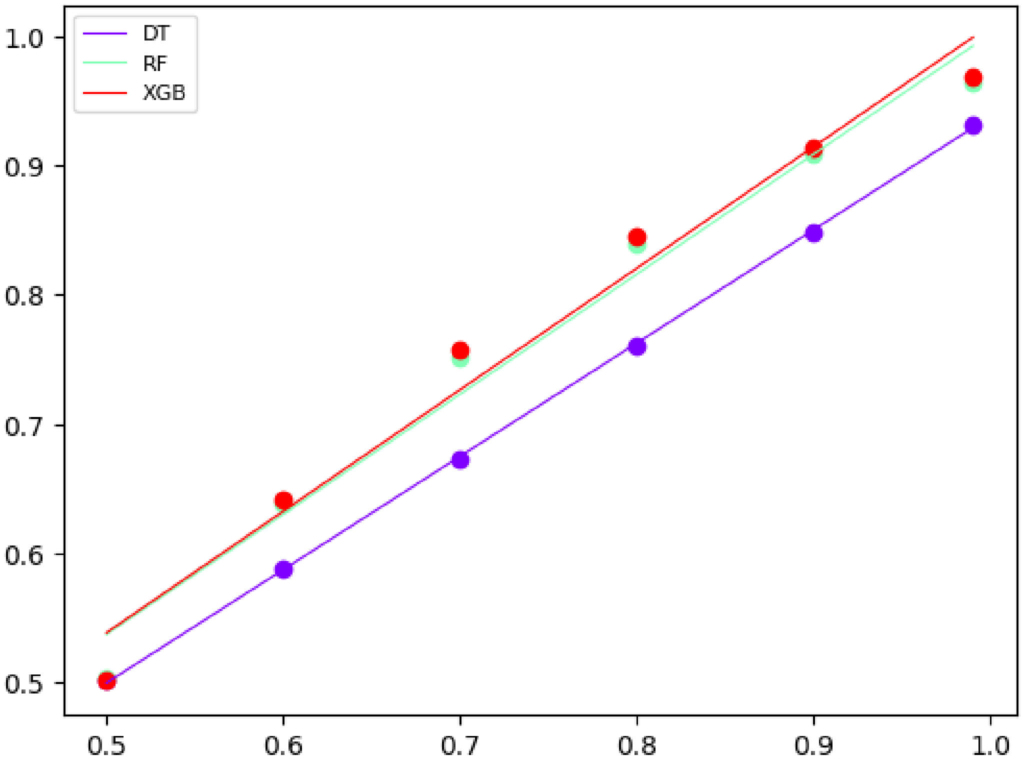
```
pv = pd.DataFrame(data=pvalues, index=["DT","RF","XGB"],␣
 ↪columns=["50%","40%","30%", "20%", "10%", "1%"])

def significant(v):
    return "font-weight: bold; color: red" if v > 0.05 else None

pv.style.applymap(significant)
```



```
[75]: <pandas.io.formats.style.Styler at 0x7f33c46ec8d0>
```

```
[76]: names = ["DT","RF","XGB"]
cor = []
per = []
slopes = []
auc_scores = []
max_perm = [0] * len(perc) # List of values for maximal slopes across all models

for i, c in zip(range(a[1]),colors):
    for j in range(len(perc)):
        plt.scatter(np.mean(corr[:,j]), np.mean(perm[:,j,i]), color=c, alpha=1)
        # Find Maximal values for each correlation level
        if max_perm[j] < np.mean(perm[:,j,i]):
            max_perm[j] = np.mean(perm[:,j,i])
    cor = np.mean(corr[:,:], axis=0)
    per = np.mean(perm[:,:,i], axis=0)
    auc_score = auc(cor,per)
    slope, intercept = np.polyfit(cor, per, 1)
    plt.plot(cor, slope*cor + intercept, color=c, linewidth=0.8)
    print(names[i], '=', slope)
    slopes = np.append(slopes, slope)
    auc_scores = np.append(auc_scores, auc_score)

plt.legend(names, prop={'size': 8})

maxind = np.argmax(abs(slopes))
maxind_auc = np.argmax(abs(auc_scores))

print('Slope:', np.max(abs(slopes)), '-', names[maxind])
print('AUC:', np.max(abs(auc_scores)), '-', names[maxind_auc])
print("Top AUC", auc(cor,max_perm),"- Max F1:",max_perm[-1],"- Final Metric:
 ↪",(0.5-auc(cor,max_perm)/max_perm[-1])/0.125)
```

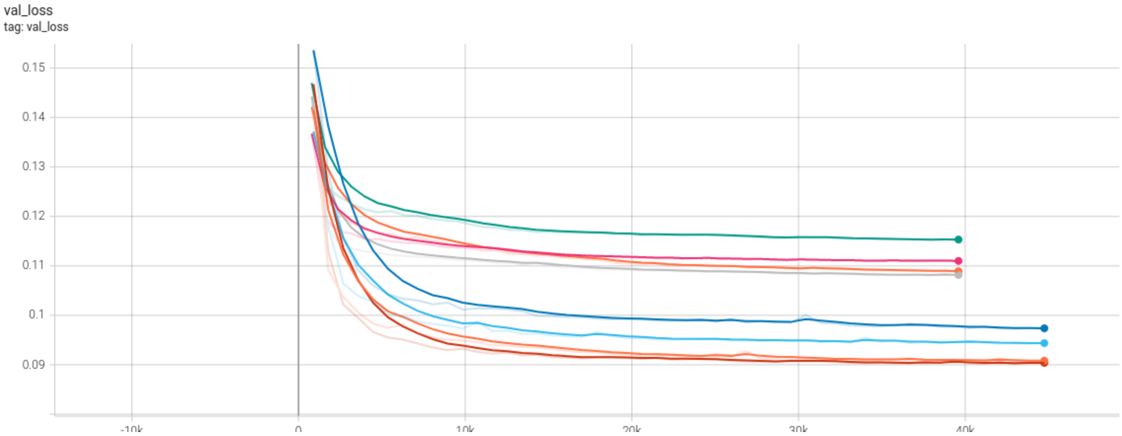
```
DT = 0.8782781857307009
RF = 0.9308594164102943
XGB = 0.941785711074947
Slope: 0.941785711074947 - XGB
AUC: 0.3798507743962079 - XGB
Top AUC 0.3798957282620485 - Max F1: 0.969534817750935 - Final Metric:
0.8653360658604807
```



5 Test 3: Dataset Complexity

[ ]:

```
## Description:

## Input:

## Output:
```



[ ]:

```
```