In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

Packages

In [ ]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import time
import warnings
import importlib
from fastprogress import master_bar, progress_bar

from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.naive_bayes import BernoulliNB
from sklearn.metrics import roc_curve, roc_auc_score, classification_r
eport
from sklearn.metrics import confusion_matrix, accuracy_score, f1_scor
e, precision_score
from sklearn.metrics import recall_score, balanced_accuracy_score
from sklearn.metrics import average_precision_score, log_loss
from sklearn.metrics import precision_recall_curve, auc, average_preci
sion_score
from sklearn.neural_network import MLPClassifier
from sklearn import ensemble, preprocessing
from sklearn.preprocessing import StandardScaler
from scipy.stats import ttest_rel, ttest_ind

%config InlineBackend.figure_format = 'retina'
pd.options.mode.chained_assignment = None
pd.options.display.max_columns = None
warnings.filterwarnings('ignore')
warnings.filterwarnings("ignore", category=FutureWarning)
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

Utility functions:

In [ ]:

```
# instantiate a model using the parameters and import source
def get_model(model_name:str, import_module:str, model_params:dict):
    model_class = getattr(importlib.import_module(import_module), mode
l_name)
    model = model_class(**model_params) # Instantiates the model
    return model

# returns the p_values for the each value except the best for each col
umn
def p_stats(raw_data, raw_mean, rows, cols, per_data=False):
    # get index of algo with highest performance for each metric or da
taset (in each column)
    best_algo_in_cols = raw_mean.to_numpy().argmax(axis=0)
    algo_raw = []

    for alg in raw_data:
        if per_data:
            # due to the way data is store
            algo_raw.append(np.split(alg.flatten(), cols))
        else:
            algo_raw.append(np.split(alg.flatten('F'), cols))

    p_val_test = np.ones_like(raw_mean)

    # for each metric, get the best performing algo first
    for col in range(cols):
        idx = best_algo_in_cols[col]
        # get raw data for best algo and metric(col)
        best_raw = algo_raw[idx][col]
        # run t-test between this algo(idx) and other algos
        for id_alg in range(rows):
            if (id_alg == idx):
                continue
            else:
                # run t-test between this and the best
                this_raw = algo_raw[id_alg][col]
                t_stat, p_stat = ttest_ind(best_raw, this_raw, nan_pol
icy='omit')
                p_val_test[id_alg][col] = p_stat
    return p_val_test

def plot_roc_pr(real_y, prob_y, algo_name, data_name):
    fig, axes = plt.subplots(1,2, figsize=(12,10))
    # for roc_auc_curve
    # roc_score = roc_auc_score(real_y, prob_y)

    fpr, tpr, _ = roc_curve(real_y, prob_y)
    lab1 = 'ROC Area=%.2f' % (auc(fpr, tpr))
    axes[0].plot([1, 0], [1, 0], color='red', lw=2, linestyle='--', la
bel='Diagonal')
    axes[0].step(fpr, tpr, label=lab1, lw=2, color='black')
    axes[0].set_title((algo_name)+ ", Data: " + str(data_name))
    axes[0].set_xlabel('False Positive Rate')
    axes[0].set_ylabel('True Positive Rate')
    axes[0].legend(loc='lower left')
    # set same size for two subplots
    asp0 = np.diff(axes[0].get_xlim())[0] / np.diff(axes[0].get_ylim())[0]
    axes[0].set_aspect(asp0)
    # for precision-recall
    precision, recall, _ = precision_recall_curve(real_y, prob_y)
    ave_PR = average_precision_score(real_y, prob_y)
    lab2 = 'PR AUC=%.2f' % (ave_PR)
    # add diagonal line
    axes[1].plot([0.0, 1.0], [1.0, 0.0], color='red', lw=2, linestyle='--', label='Diagonal')
    axes[1].step(recall, precision, label=lab2, lw=2, color='black')
    axes[1].set_title((algo_name)+ ", Data: " + str(data_name))
    axes[1].set_xlabel('Recall')
    axes[1].set_ylabel('Precision')
    axes[1].legend(loc='lower left')
    asp1 = np.diff(axes[1].get_xlim())[0] / np.diff(axes[1].get_ylim())[0]
    axes[1].set_aspect(asp1)
    plt.show()
    fig.tight_layout()
```

Load cleaned data:

In [ ]:

```
ADULT = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/data/cleaned/adult.csv').drop('Unnamed: 0', axis=1)
LETTER1 = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/data/cleaned/letter_p1.csv').drop('Unnamed: 0', axis=1)
LETTER2 = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/data/cleaned/letter_p2.csv').drop('Unnamed: 0', axis=1)
COVTYPE = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/data/cleaned/covtype5.csv').drop('Unnamed: 0', axis=1)
```

Algorithms and their parameters:

In [ ]:

```
LOGIT = {'name' : 'LogisticRegression()',
            'name_str' : 'LogisticRegression',
            'module' : 'sklearn.linear_model',
            'hyperparameters' : {
                 'solver': ['liblinear'],
                 'C' : [10**i for i in range(-8, 4)],
                 }
            }

TREES = {'name' : 'DecisionTreeClassifier()',
          'name_str' : 'DecisionTreeClassifier',
          'module' : 'sklearn.tree',
          'hyperparameters' : {
              'criterion': ['gini', 'entropy'],
              'max_depth': [i for i in range(1, 26, 3)],
              'min_samples_split': [i for i in range(1, 10, 2)],
              'min_samples_leaf': [1,3,5,7],
              }
         }

FOREST = {'name' : 'RandomForestClassifier()',
          'name_str' : 'RandomForestClassifier',
          'module' : 'sklearn.ensemble',
          'hyperparameters' : {
              'n_estimators' : [2**i for i in range(1, 11)],
              'max_features' : [1,2,4,6,8,12,16,20],
              }
          }

GBOOST = {'name' : 'GradientBoostingClassifier()',
             'name_str' : 'GradientBoostingClassifier',
             'module' : 'sklearn.ensemble',
             'hyperparameters' : {
                 'n_estimators' : [2,4,8,16,32,64,128,256,512,1024,204
8],
                 'max_depth': [i for i in range(1, 10, 2)],
                 }
              }

MLP_ADAM = {'name' : 'MLPClassifier()',
       'name_str' : 'MLPClassifier',
       'module' : 'sklearn.neural_network',
       'hyperparameters' : {
           'hidden_layer_sizes' : [1,2,4,8,32,128,256],
           }
            }

NB = {'name' : 'BernoulliNB()',
             'name_str' : 'BernoulliNB',
             'module' : 'sklearn.naive_bayes',
             'hyperparameters' : {
                 'alpha' : [10**i for i in range(-4, 4)]
                 }}
```

Function and variables for training classifiers

In [ ]:

```
TRIALS = 5
CV_NUM = 5

# add all datasets/algos to a list
data_list = [ADULT, LETTER1, LETTER2, COVTYPE]
algorithms = [LOGIT, TREES, FOREST, GBOOST, MLP_ADAM, NB]

# string names of variables used for printing
dataset_names =  ['ADULT', 'LETTER1', 'LETTER2', 'COVTYPE']
algo_names = ['LOGIT', 'TREES', 'FOREST', 'GBOOST', 'MLP_ADAM', 'NB']
metric_names = ['ACC','APR','ROC']
scorings = {'ACC':'accuracy', 'APR': 'average_precision','ROC' : 'roc_
auc'}

# metrics used
scorers = [accuracy_score, average_precision_score, roc_auc_score]

# scores of each algorithm by data set (averaged over all metrics)
algo_data_df = pd.DataFrame(0.0, index=algo_names, columns=dataset_nam
es)

# scores for each algorithm by metric (average over all data sets)
algo_metric_df_train = pd.DataFrame(0.0, index=algo_names, columns=met
ric_names)
algo_metric_df_test = pd.DataFrame(0.0, index=algo_names, columns=metr
ic_names)

# for each algo, store raw train/test scores for each dataset
# number_of_dataset by trails by number_of_metrics
raw_train = np.zeros((len(algorithms), len(dataset_names), TRIALS, len
(metric_names)))
raw_test = np.zeros((len(algorithms), len(dataset_names), TRIALS, len
(metric_names)))


# work on one algorithm
def learn(idx_algo, algo):
    start = time.time()
    print('\n
--------------------------------------------------------------\n')
    print(f'Started training {algo["name"]}')

    # for this algo, store training metrics averaged across all datase
ts after the loop over data_list
    metric_across_data_train = np.zeros((len(dataset_names), len(metri
c_names)))

    # for this algo, store testing metrics averaged across all dataset
s after the loop over data_list
    metric_across_data_test = np.zeros((len(dataset_names), len(metric
_names)))

    # for this algo, store raw training metrics
    # for each dataset, there is 3 metrics. For each metric, there is
5 trials
```

```
    raw_metric_data_train = np.zeros((len(dataset_names), TRIALS, len(metric_names)))

    # for this algo, store raw training metrics
    raw_metric_data_test = np.zeros((len(dataset_names), TRIALS, len(metric_names)))

    # loop over all datasets
    for idx_data, data in enumerate(progress_bar(data_list)):
        start_data = time.time()
        print(f'Started on {dataset_names[idx_data]} dataset')

        # for each algo/data combo, store testing metrics averaged across 5 trials
        metric_across_trials_test = np.zeros((TRIALS, len(metric_names)))
        # for each algo/data combo, store training metrics averaged across 5 trials
        metric_across_trials_train = np.zeros((TRIALS, len(metric_names)))

        # data for precision-recall curve
        auc_real_y = []
        auc_prob_y = []

        # loop over all trials
        for trial in progress_bar(range(TRIALS)):
            start_trial = time.time()
            print("Started trial: ", trial+1)

            # pick 5000 samples for training
            X = data.drop('y', axis=1)
            Y = data['y']
            X_train, X_test, Y_train, Y_test = train_test_split(X, Y, train_size=5000, random_state=trial)

            # only scale data for MLP's or LOGREG
            if (algo['name_str'] in ['MLPClassifier', 'LogisticRegression']):
                scaler = StandardScaler()
                X_train = pd.DataFrame(scaler.fit_transform(X_train), columns = X.columns)
                X_test = pd.DataFrame(scaler.transform(X_test), columns = X.columns)

            clf = eval(algo['name'])
            param = algo['hyperparameters']

            # Get each parameter that has best performance on validation set

            CV = GridSearchCV(clf, param, cv=CV_NUM, n_jobs=-1, scoring=scorings, refit='ACC')
            CV.fit(X_train, Y_train)
            result_cv = CV.cv_results_
            print("\nBest param for ACC:", CV.best_params_)

            # get parameters for best models for each metric
            param_list = []
            for metric in metric_names:
                best_id = pd.Series(result_cv['rank_test_'+str(metri
c)]).idxmin()
                param_list.append(result_cv['params'][best_id])

            # Train n models using the 5000 samples and each of the n
best parameters
            # and test on test set
            clf_name = algo['name_str']
            module = algo['module']
            train_metrics = []
            test_metrics = []

            for i in range(len(param_list)):
                clf_best = get_model(clf_name, module, param_list[i])
                clf_best.fit(X_train,Y_train)

                # for roc_auc and PR curves
                pred_proba = clf_best.predict_proba(X_test)
                auc_real_y.append(Y_test)
                auc_prob_y.append(pred_proba[:,1])

                X_train_pred = clf_best.predict(X_train)
                X_test_pred = clf_best.predict(X_test)
                X_train_pred_prob = clf_best.predict_proba(X_train)
                X_test_pred_prob = clf_best.predict_proba(X_test)

                # get appropriate scoring function
                scorer = scorers[i]

                # if metric is average precision, need y-score or prob
a
                if (i==1):
                    train_metrics.append(scorer(Y_train, X_train_pred_
prob[:, 1]))
                    test_metrics.append(scorer(Y_test, X_test_pred_pro
b[:, 1]))
                # if scorer is roc_auc, need proba
                elif (i==2):
                    train_metrics.append(scorer(Y_train, X_train_pred_
prob[:, 1]))
                    test_metrics.append(scorer(Y_test, X_test_pred_pro
b[:, 1]))
                else:
                    train_metrics.append(scorer(Y_train, X_train_pre
d))
                    test_metrics.append(scorer(Y_test, X_test_pred))


            # update the row in metric_across_trials_train
            metric_across_trials_train[trial] = train_metrics
            # update the row in metric_across_trials
            metric_across_trials_test[trial] = test_metrics

            finish_trial = time.time()
            print(f'Ended trial {trial+1} in {(finish_trial - start_tr
ial):.3f} seconds')
            print('\n')

        # plot ROC and PR curves
        auc_real_y = np.concatenate(auc_real_y)
        auc_prob_y = np.concatenate(auc_prob_y)
        plot_roc_pr(auc_real_y, auc_prob_y, algo['name_str'], dataset_
names[idx_data])

        # add 5 trails by 3 metrics data to raw list
        raw_metric_data_train[idx_data] = metric_across_trials_train
        raw_metric_data_test[idx_data] = metric_across_trials_test

        # mean of metrics across trials
        mean_across_trials_train = np.mean(metric_across_trials_train,
axis=0)
        mean_across_trials_test = np.mean(metric_across_trials_test, a
xis=0)

        # update algo-data combo with mean of mean_across_trials
        mean_algo_data = np.mean(mean_across_trials_test)
        algo_data_df.iat[idx_algo, idx_data] = mean_algo_data

        # update metric_across_data
        metric_across_data_train[idx_data] = mean_across_trials_train
        metric_across_data_test[idx_data] = mean_across_trials_test

        finish_data = time.time()
        print(f'Ended {dataset_names[idx_data]} in {(finish_data - sta
rt_data):.3f} seconds')

    # update raw_train and raw_test
    raw_train[idx_algo] = raw_metric_data_train
    raw_test[idx_algo] = raw_metric_data_test

    # mean of metrics across data
    mean_across_data_train = np.mean(metric_across_data_train, axis=0)
    mean_across_data_test = np.mean(metric_across_data_test, axis=0)
    algo_metric_df_train.iloc[idx_algo] = mean_across_data_train
    algo_metric_df_test.iloc[idx_algo] = mean_across_data_test

    finish   = time.time()
    print(f'Ended {algo["name"]} in {(finish - start):.3f} seconds')
    print('\n
--------------------------------------------------------------\n')
```

Training Per Algorithm

Logit

In [ ]:

```
# algorithms = [LOGIT, TREES, FOREST, GBOOST, MLP_ADAM, NB]
learn(0, algorithms[0])
```

Decision Trees

In [ ]:

```
# algorithms = [LOGIT, TREES, FOREST, GBOOST, MLP_ADAM, NB]
learn(1, algorithms[1])
```

Random Forest

In [ ]:

```
# algorithms = [LOGIT, TREES, FOREST, GBOOST, MLP_ADAM, NB]
learn(2, algorithms[2])
```

Gradient Boosting Classifier

In [ ]:

```
# algorithms = [LOGIT, TREES, FOREST, GBOOST, MLP_ADAM, NB]
learn(3, algorithms[3])
```

MLP with ADAM

In [ ]:

```
# algorithms = [LOGIT, TREES, FOREST, GBOOST, MLP_ADAM, NB]
learn(4, algorithms[4])
```

Naive Bayes

In [ ]:

```
# algorithms = [LOGIT, TREES, FOREST, GBOOST, MLP_ADAM, NB]
learn(5, algorithms[5])
```

Tables and Output

In [ ]:

```
p_val_test_metric = p_stats(raw_test, algo_metric_df_test, len(algo_names), len(metric_names), per_data=False)
print('p_val_test_metric: for Table 2')
print(p_val_test_metric);print()

p_val_train_metric =  p_stats(raw_train, algo_metric_df_train, len(algo_names), len(metric_names), per_data=False)
print('p_val_train_metric:')
print(p_val_train_metric);print()

p_val_test_data = p_stats(raw_test, algo_data_df, len(algo_names), len(data_list), per_data=True)
print('p_val_test_data: for Table 3')
print(p_val_test_data);print()
```