Lab21

November 8, 2022

1 Boosting

1.1 AdaBoost

[ ]:
```
import sklearn.datasets as data
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier,␣
↪RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import cross_val_score, cross_validate
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

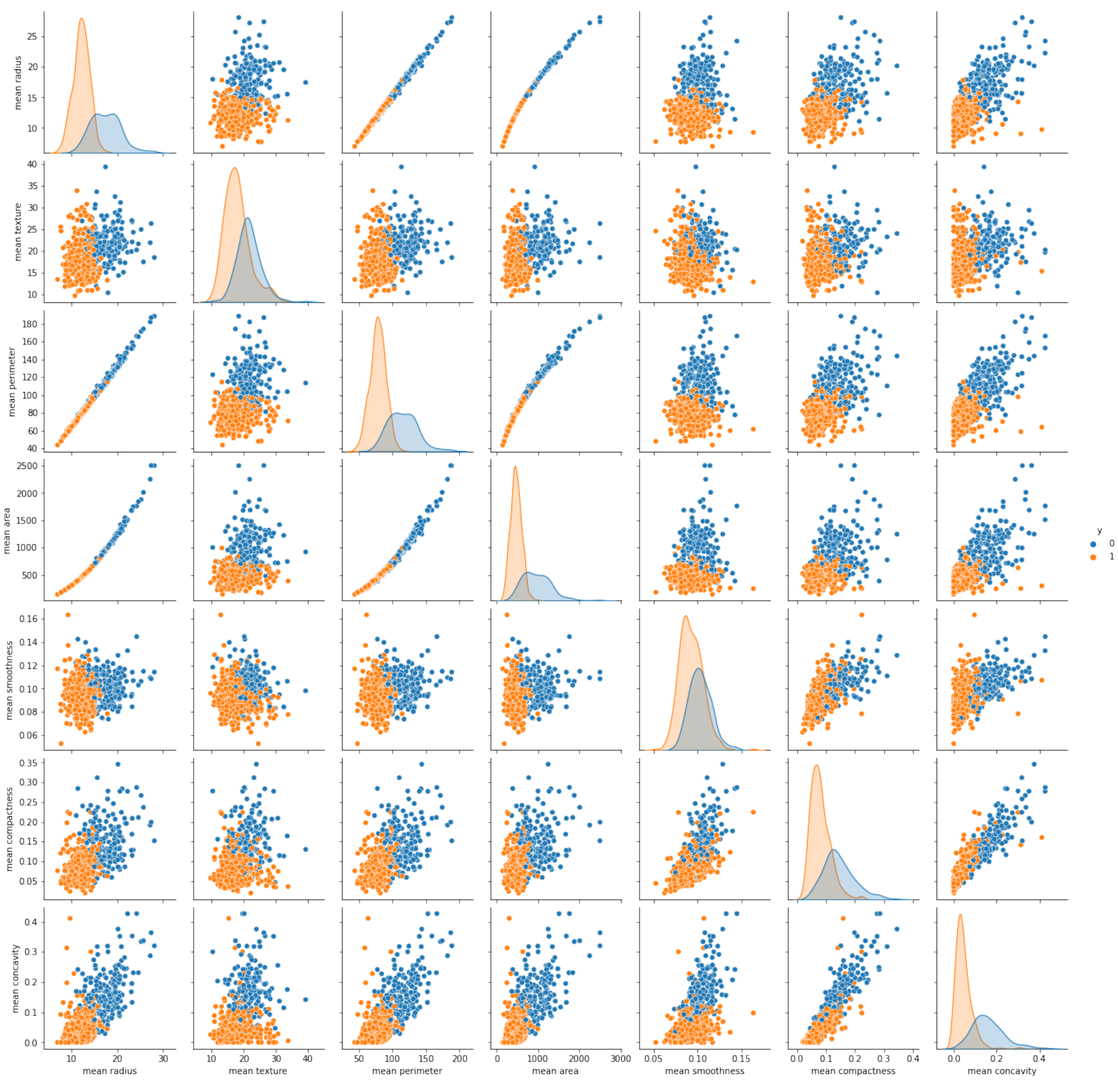
[ ]:
```
# Load breast cancer data
breastCancerFr = data.load_breast_cancer(as_frame=True).data
X = data.load_breast_cancer().data
y = data.load_breast_cancer(as_frame=True).target
breastCancerFr['y'] = y
breastCancerFr
```

[ ]:
```
     mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0          17.99         10.38          122.80     1001.0          0.11840
1          20.57         17.77          132.90     1326.0          0.08474
2          19.69         21.25          130.00     1203.0          0.10960
3          11.42         20.38           77.58      386.1          0.14250
4          20.29         14.34          135.10     1297.0          0.10030
..           …            …              …         …              …
564        21.56         22.39          142.00     1479.0          0.11100
565        20.13         28.25          131.20     1261.0          0.09780
```

```
566        16.60         28.08          108.30       858.1           0.08455
567        20.60         29.33          140.10      1265.0           0.11780
568         7.76         24.54           47.92       181.0           0.05263

     mean compactness  mean concavity  mean concave points  mean symmetry  \
0             0.27760         0.30010              0.14710         0.2419
1             0.07864         0.08690              0.07017         0.1812
2             0.15990         0.19740              0.12790         0.2069
3             0.28390         0.24140              0.10520         0.2597
4             0.13280         0.19800              0.10430         0.1809
..                ...             ...                  ...            ...
564           0.11590         0.24390              0.13890         0.1726
565           0.10340         0.14400              0.09791         0.1752
566           0.10230         0.09251              0.05302         0.1590
567           0.27700         0.35140              0.15200         0.2397
568           0.04362         0.00000              0.00000         0.1587

     mean fractal dimension  ...  worst texture  worst perimeter  worst area  \
0                   0.07871  ...          17.33           184.60      2019.0
1                   0.05667  ...          23.41           158.80      1956.0
2                   0.05999  ...          25.53           152.50      1709.0
3                   0.09744  ...          26.50            98.87       567.7
4                   0.05883  ...          16.67           152.20      1575.0
..                      ...  ...            ...              ...         ...
564                 0.05623  ...          26.40           166.10      2027.0
565                 0.05533  ...          38.25           155.00      1731.0
566                 0.05648  ...          34.12           126.70      1124.0
567                 0.07016  ...          39.42           184.60      1821.0
568                 0.05884  ...          30.37            59.16       268.6

     worst smoothness  worst compactness  worst concavity  \
0             0.16220            0.66560           0.7119
1             0.12380            0.18660           0.2416
2             0.14440            0.42450           0.4504
3             0.20980            0.86630           0.6869
4             0.13740            0.20500           0.4000
..                ...                ...              ...
564           0.14100            0.21130           0.4107
565           0.11660            0.19220           0.3215
566           0.11390            0.30940           0.3403
567           0.16500            0.86810           0.9387
568           0.08996            0.06444           0.0000

     worst concave points  worst symmetry  worst fractal dimension  y
0                  0.2654          0.4601                  0.11890  0
1                  0.1860          0.2750                  0.08902  0
2                  0.2430          0.3613                  0.08758  0
```

```
3                    0.2575          0.6638                  0.17300  0
4                    0.1625          0.2364                  0.07678  0
..                     …               …                     … ..
564                  0.2216          0.2060                  0.07115  0
565                  0.1628          0.2572                  0.06637  0
566                  0.1418          0.2218                  0.07820  0
567                  0.2650          0.4087                  0.12400  0
568                  0.0000          0.2871                  0.07039  1

[569 rows x 31 columns]
```

[ ]:
```
# Visualize some columns
sns.pairplot(breastCancerFr[['mean radius',
                             'mean texture',
                             'mean perimeter',
                             'mean area',
                             'mean smoothness',
                             'mean compactness',
                             'mean concavity',
                             'y']], hue='y');
```




```
[ ]: # (Stratified) split dataset into training set and test set
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
      ↪stratify=y)
```

```
[ ]: # Create adaboost classifer object
     # It uses decision trees of depth 1 by default but you can change it using the␣
      ↪base_estimator parameter!
     abc = AdaBoostClassifier(n_estimators=10,
                              learning_rate=1)

     # Train Adaboost Classifer
     model = abc.fit(X_train, y_train)

     #Predict the response for test dataset
```

```
y_pred = model.predict(X_test)

# Check performance
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.92      0.91      0.91        64
           1       0.94      0.95      0.95       107

    accuracy                           0.94       171
   macro avg       0.93      0.93      0.93       171
weighted avg       0.94      0.94      0.94       171
```

[ ]:
```
# Using SVM as a base classifier
svc=SVC(probability=True, kernel='rbf')

# Create adaboost classifer object
abc = AdaBoostClassifier(n_estimators=50, base_estimator=svc,learning_rate=1)

# Train Adaboost Classifer
model = abc.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = model.predict(X_test)

# Check performance
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.84      0.92        64
           1       0.91      1.00      0.96       107

    accuracy                           0.94       171
   macro avg       0.96      0.92      0.94       171
weighted avg       0.95      0.94      0.94       171
```

1.2 Gradient Boosting

[ ]:
```
# define the model
gbc = GradientBoostingClassifier(n_estimators = 100)

# Train gradient boosting Classifer
model = gbc.fit(X_train, y_train)
```

```python
#Predict the response for test dataset
y_pred = model.predict(X_test)

# Check performance
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.91      0.91      0.91        64
           1       0.94      0.94      0.94       107

    accuracy                           0.93       171
   macro avg       0.93      0.93      0.93       171
weighted avg       0.93      0.93      0.93       171
```

1.3 Comparison

1.3.1 Performance

[ ]:
```python
abc = AdaBoostClassifier(n_estimators=100)
model = abc.fit(X_train, y_train)
y_pred = model.predict(X_test) #Predict the response for test dataset
print("ADABOOST CLASSIFIER PERFORMANCE + CONFUSION")
print(classification_report(y_test, y_pred)) # Check performance
print(confusion_matrix(y_test, y_pred))
```

```
ADABOOST CLASSIFIER PERFORMANCE + CONFUSION
              precision    recall  f1-score   support

           0       0.97      0.95      0.96        64
           1       0.97      0.98      0.98       107

    accuracy                           0.97       171
   macro avg       0.97      0.97      0.97       171
weighted avg       0.97      0.97      0.97       171

[[ 61   3]
 [  2 105]]
```

[ ]:
```python
gbc = GradientBoostingClassifier(n_estimators=100)
model = gbc.fit(X_train, y_train)
y_pred = model.predict(X_test) #Predict the response for test dataset
print("GRADIENT BOOSTING CLASSIFIER PERFORMANCE")
print(classification_report(y_test, y_pred)) # Check performance
print(confusion_matrix(y_test, y_pred))
```

```
GRADIENT BOOSTING CLASSIFIER PERFORMANCE
              precision    recall  f1-score   support

           0       0.92      0.91      0.91        64
           1       0.94      0.95      0.95       107

    accuracy                           0.94       171
   macro avg       0.93      0.93      0.93       171
weighted avg       0.94      0.94      0.94       171

[[ 58   6]
 [  5 102]]
```

[ ]:
```
rfc = RandomForestClassifier(n_estimators=100)
model = rfc.fit(X_train, y_train)
y_pred = model.predict(X_test) #Predict the response for test dataset
print("RANDOM FOREST CLASSIFIER PERFORMANCE")
print(classification_report(y_test, y_pred)) # Check performance
print(confusion_matrix(y_test, y_pred))
```

```
RANDOM FOREST CLASSIFIER PERFORMANCE
              precision    recall  f1-score   support

           0       0.94      0.92      0.93        64
           1       0.95      0.96      0.96       107

    accuracy                           0.95       171
   macro avg       0.95      0.94      0.94       171
weighted avg       0.95      0.95      0.95       171

[[ 59   5]
 [  4 103]]
```

1.3.2 Feature Importance

[ ]:
```
def get_feature_importance_names(f):
    x = list(zip(f, breastCancerFr.columns))
    x.sort(reverse = True, key = lambda e: e[0])
    return [e[1] for e in x]

print("MOST IMPORTANT FEATURES")
print(f"{'ADABOOST':<30}{'GRADIENT BOOSTING':<30}{'RANDOM FOREST':<30}")
print("\n".join(map(lambda e: f"{e[0]:<30}{e[1]:<30}{e[2]:<30}",
    zip(get_feature_importance_names(abc.feature_importances_),
        get_feature_importance_names(gbc.feature_importances_),
        get_feature_importance_names(rfc.feature_importances_)))))
```

```
MOST IMPORTANT FEATURES
```

```
ADABOOST                       GRADIENT BOOSTING              RANDOM FOREST
worst area                     mean concave points            worst perimeter
mean texture                   worst area                     worst area
worst smoothness               worst concave points           worst concave points
mean concave points            worst texture                  mean concave points
area error                     worst perimeter                worst radius
worst texture                  worst fractal dimension        mean concavity
concavity error                worst smoothness               worst concavity
worst concavity                radius error                   mean area
smoothness error               worst concavity                mean perimeter
worst concave points           mean concavity                 mean radius
texture error                  worst radius                   area error
compactness error              mean texture                   worst texture
worst perimeter                compactness error              worst smoothness
mean compactness               mean area                      mean texture
mean concavity                 fractal dimension error        mean smoothness
mean symmetry                  smoothness error               mean compactness
radius error                   mean compactness               radius error
concave points error           perimeter error                worst fractal
dimension
fractal dimension error        mean radius                    worst compactness
worst fractal dimension        mean symmetry                  worst symmetry
mean fractal dimension         mean perimeter                 concavity error
perimeter error                area error                     perimeter error
symmetry error                 worst symmetry                 fractal dimension
error
worst symmetry                 concavity error                texture error
mean radius                    worst compactness              compactness error
mean perimeter                 mean fractal dimension         mean symmetry
mean area                      concave points error           smoothness error
mean smoothness                texture error                  concave points error
worst radius                   mean smoothness                mean fractal
dimension
worst compactness              symmetry error                 symmetry error
```

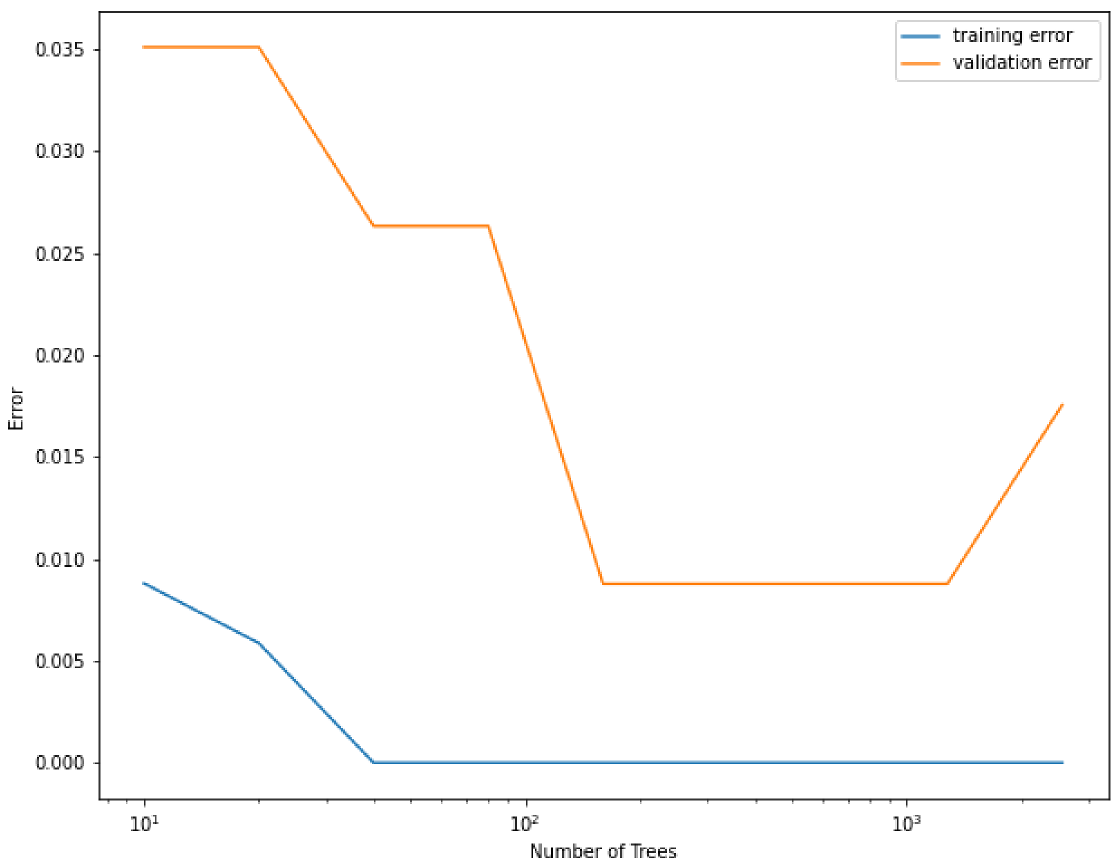
1.4 Hyperparameter tuning

[ ]:
```
# (Stratified) split dataset into training, validation, and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,␣
 ↪stratify=y)
X_val, X_test, y_val, y_test = train_test_split(X_test, y_test, test_size=0.5,␣
 ↪stratify=y_test)
```

[ ]:
```
num_trees = [10, 20, 40, 80, 160, 320, 640, 1280, 2560]
models = map(lambda n: GradientBoostingClassifier(n_estimators=n), num_trees)
scores = {"val_err" : [], "train_err" : []}
for m in models:
```

```
    m.fit(X_train, y_train)
    scores["val_err"].append(1 - m.score(X_val, y_val))
    scores["train_err"].append(1 - m.score(X_train, y_train))

plt.semilogx(num_trees, scores["train_err"])
plt.semilogx(num_trees, scores["val_err"])
plt.legend(["training error", "validation error"])
plt.xlabel("Number of Trees")
plt.ylabel("Error")
```

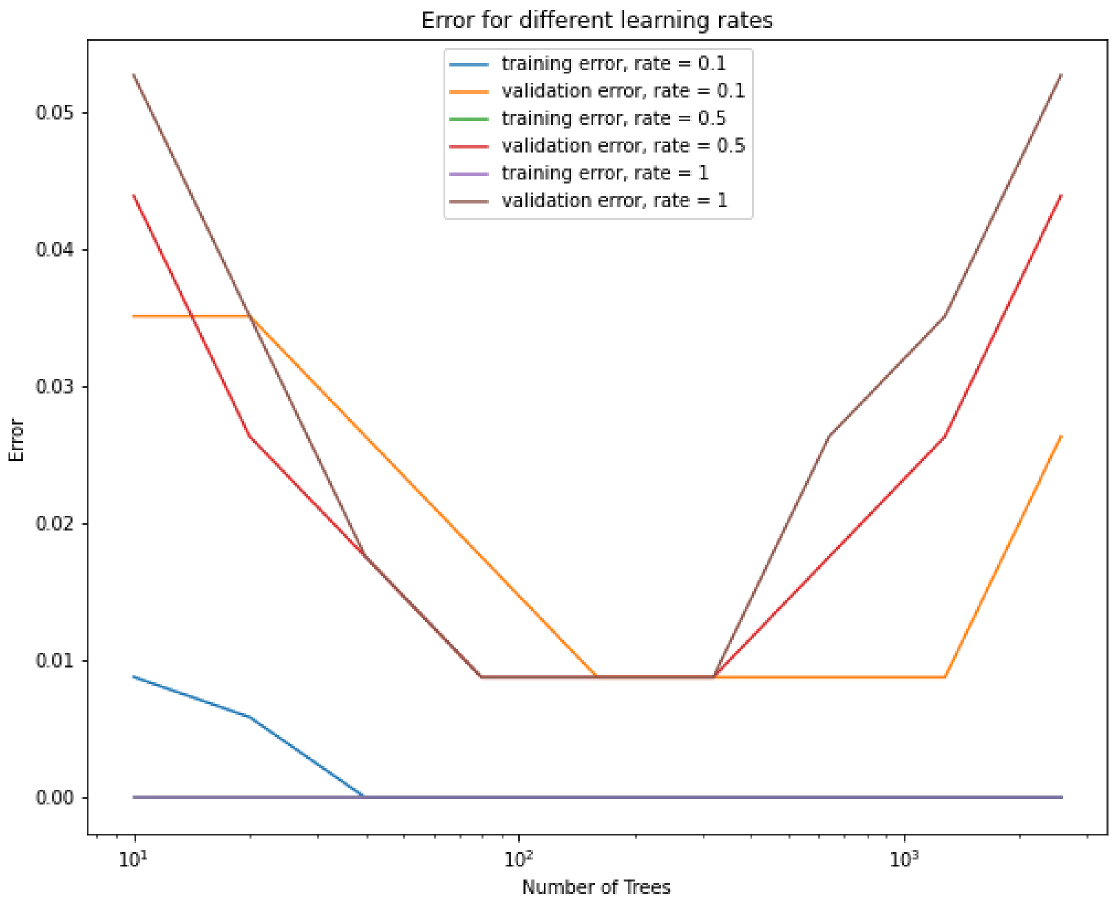
[ ]: Text(0, 0.5, 'Error')



[ ]:
```
num_trees = [10, 20, 40, 80, 160, 320, 640, 1280, 2560]
learning_rates = [0.1, 0.5, 1]
scores = {}
for r in learning_rates: scores[r] = {"val_err" : [], "train_err" : []}

legend_strings = []
for r in learning_rates:
    for n in num_trees:
```

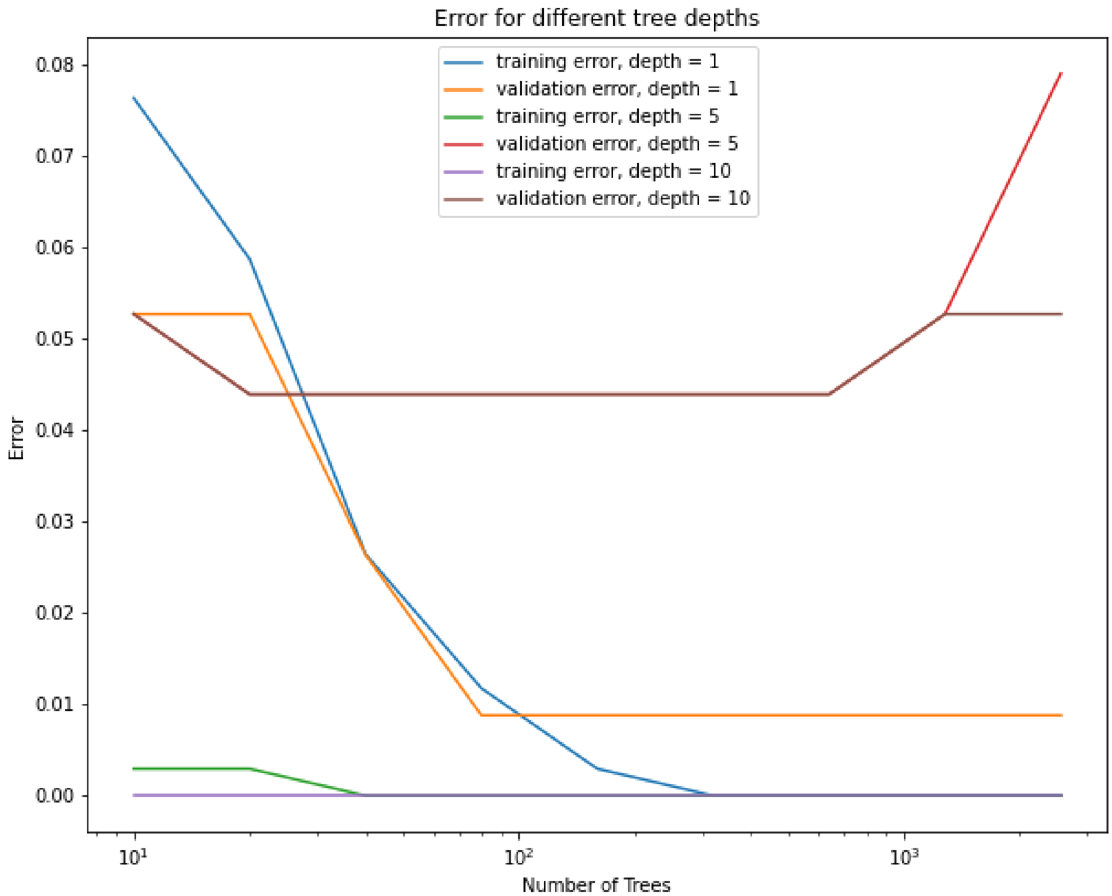
```
        m = GradientBoostingClassifier(n_estimators=n, learning_rate=r)
        m.fit(X_train, y_train)
        scores[r]["val_err"].append(1 - m.score(X_val, y_val))
        scores[r]["train_err"].append(1 - m.score(X_train, y_train))
    plt.semilogx(num_trees, scores[r]["train_err"])
    plt.semilogx(num_trees, scores[r]["val_err"])
    legend_strings.append(f"training error, rate = {r}")
    legend_strings.append(f"validation error, rate = {r}")


plt.legend(legend_strings)
plt.title("Error for different learning rates")
plt.xlabel("Number of Trees")
plt.ylabel("Error")
```

[ ]: Text(0, 0.5, 'Error')



[ ]:
```
num_trees = [10, 20, 40, 80, 160, 320, 640, 1280, 2560]
max_depths = [1, 5, 10]
scores = {}
for d in max_depths: scores[d] = {"val_err" : [], "train_err" : []}

legend_strings = []
for d in max_depths:
    for n in num_trees:
        m = GradientBoostingClassifier(n_estimators=n, max_depth=d)
        m.fit(X_train, y_train)
        scores[d]["val_err"].append(1 - m.score(X_val, y_val))
        scores[d]["train_err"].append(1 - m.score(X_train, y_train))
    plt.semilogx(num_trees, scores[d]["train_err"])
    plt.semilogx(num_trees, scores[d]["val_err"])
    legend_strings.append(f"training error, depth = {d}")
    legend_strings.append(f"validation error, depth = {d}")


plt.legend(legend_strings)
plt.title("Error for different tree depths")
plt.xlabel("Number of Trees")
plt.ylabel("Error")
```

[ ]:
```
Text(0, 0.5, 'Error')
```



1.5 hyperparameter tuning via cross validation

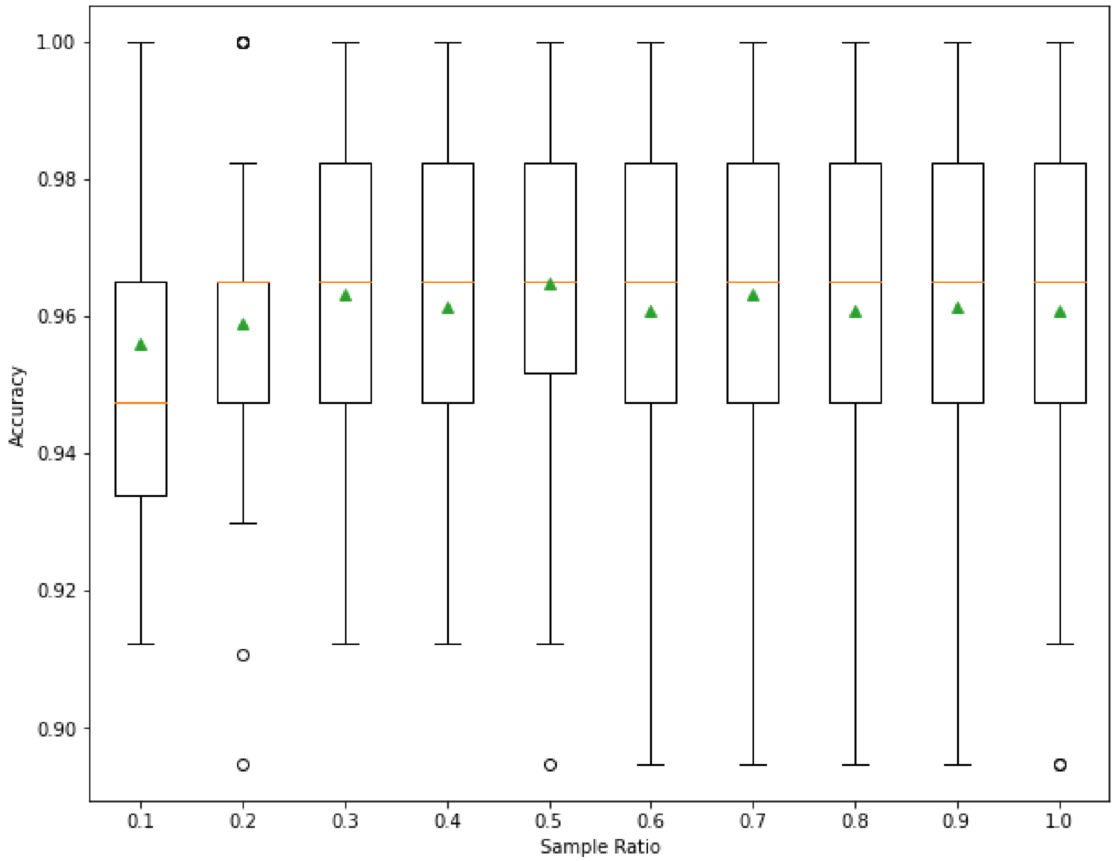
1.5.1 Choosing sample size

[ ]:
```python
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore sample ratio from 10% to 100% in 10% increments
    for i in np.arange(0.1, 1.1, 0.1):
        key = '%.1f' % i
        models[key] = GradientBoostingClassifier(subsample=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
```

```python
    return scores



# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))

# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.xlabel("Sample Ratio")
plt.ylabel("Accuracy")
```

```
>0.1 0.956 (0.023)
>0.2 0.959 (0.025)
>0.3 0.963 (0.023)
>0.4 0.961 (0.024)
>0.5 0.965 (0.024)
>0.6 0.961 (0.023)
>0.7 0.963 (0.025)
>0.8 0.961 (0.024)
>0.9 0.961 (0.026)
>1.0 0.961 (0.027)
```

```
[ ]: Text(0, 0.5, 'Accuracy')
```



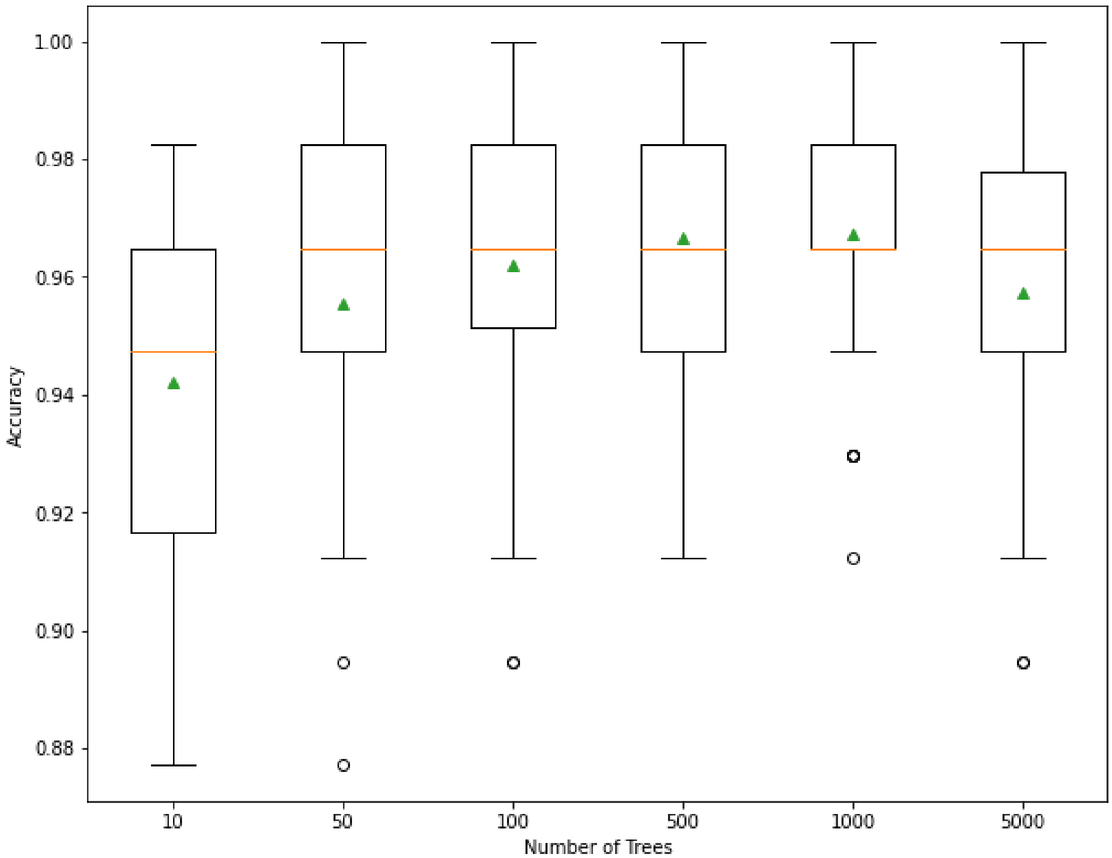
1.5.2 Choosing # trees

[ ]:
```
# get a list of models to evaluate
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [10, 50, 100, 500, 1000, 5000]
    for n in n_trees:
        models[str(n)] = GradientBoostingClassifier(n_estimators=n)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores
```

```python
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.xlabel("Number of Trees")
plt.ylabel("Accuracy")
```

```
>10 0.942 (0.030)
>50 0.956 (0.029)
>100 0.962 (0.027)
>500 0.967 (0.021)
>1000 0.967 (0.023)
>5000 0.957 (0.027)
```

[ ]:
```
Text(0, 0.5, 'Accuracy')
```

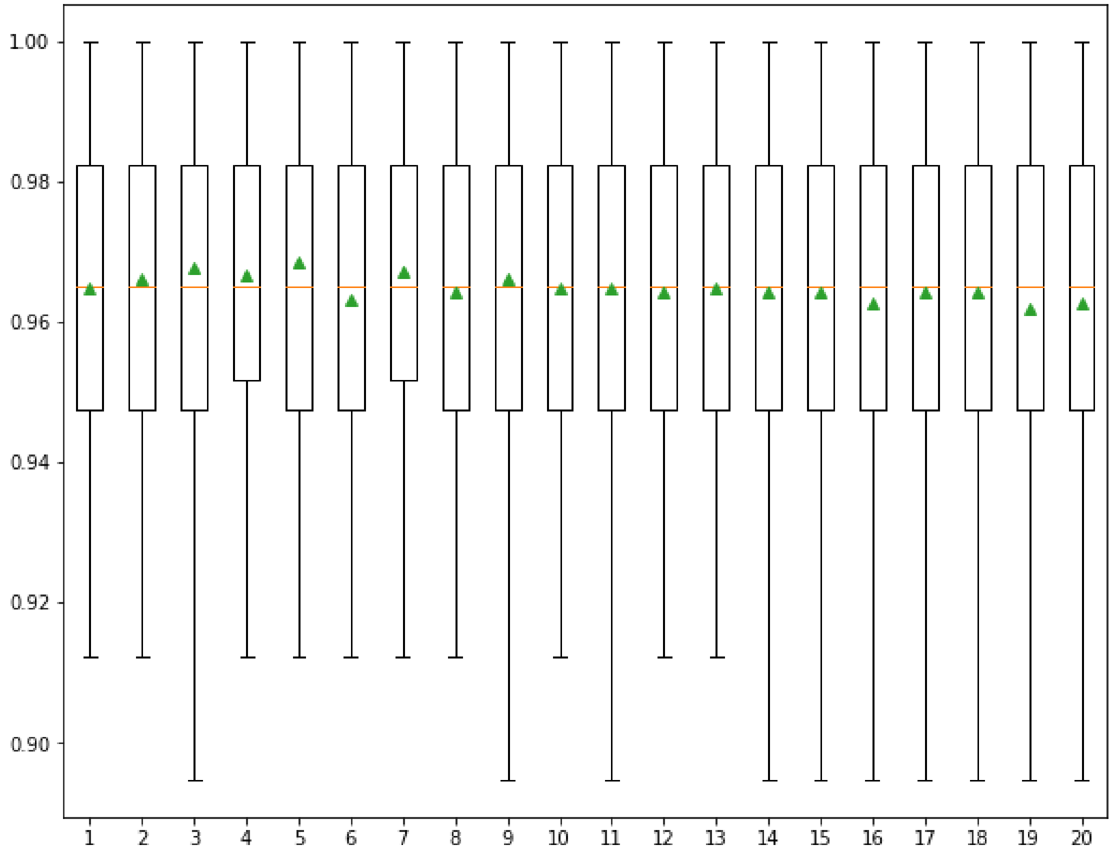


1.5.3 Choosing # features that are used in building a tree

[ ]:
```
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore number of features from 1 to 20
    for i in range(1,21):
        models[str(i)] = GradientBoostingClassifier(max_features=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
```

```
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True);
```

```
>1 0.965 (0.024)
>2 0.966 (0.024)
>3 0.968 (0.024)
>4 0.967 (0.024)
>5 0.968 (0.025)
>6 0.963 (0.023)
>7 0.967 (0.024)
>8 0.964 (0.027)
>9 0.966 (0.026)
>10 0.965 (0.024)
>11 0.965 (0.024)
>12 0.964 (0.022)
>13 0.965 (0.025)
>14 0.964 (0.027)
>15 0.964 (0.027)
>16 0.963 (0.025)
>17 0.964 (0.026)
>18 0.964 (0.026)
>19 0.962 (0.026)
>20 0.963 (0.027)
```

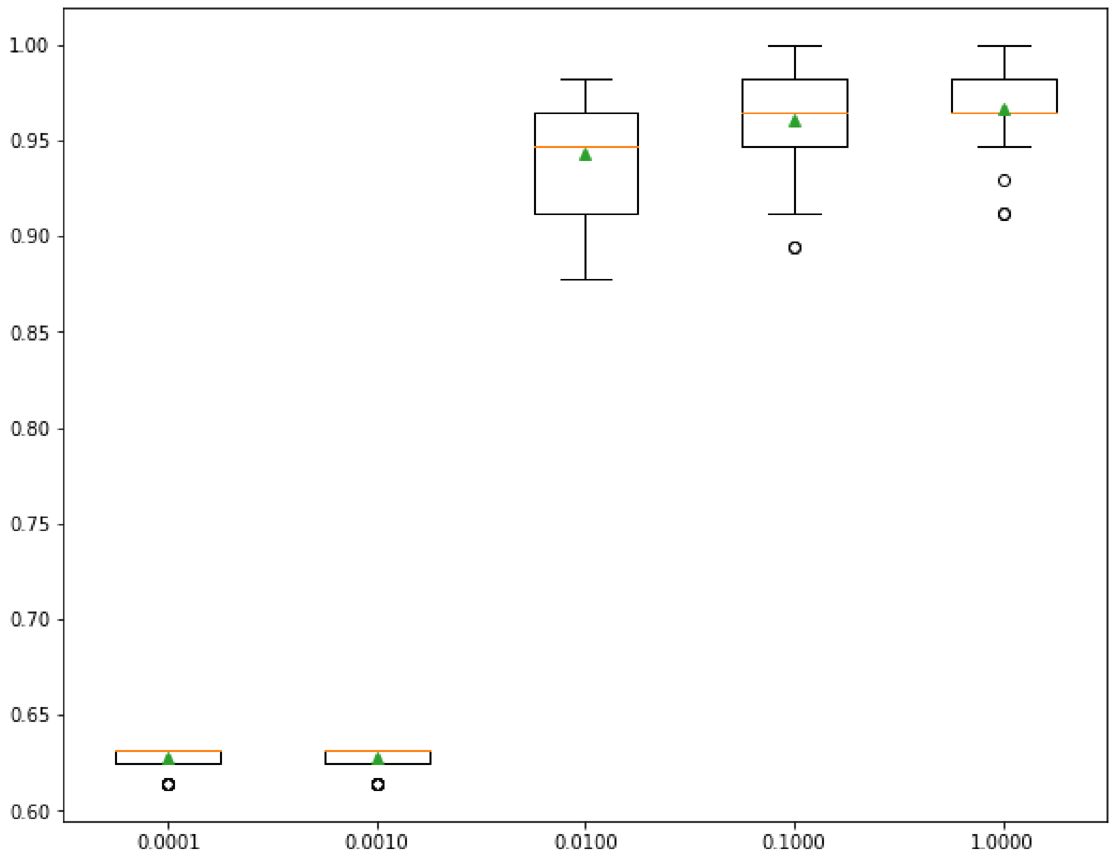


1.5.4 Choosing learning rate

```
[ ]: # get a list of models to evaluate
def get_models():
    models = dict()
    # define learning rates to explore
    for i in [0.0001, 0.001, 0.01, 0.1, 1.0]:
        key = '%.4f' % i
        models[key] = GradientBoostingClassifier(learning_rate=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
```

```python
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True);
```

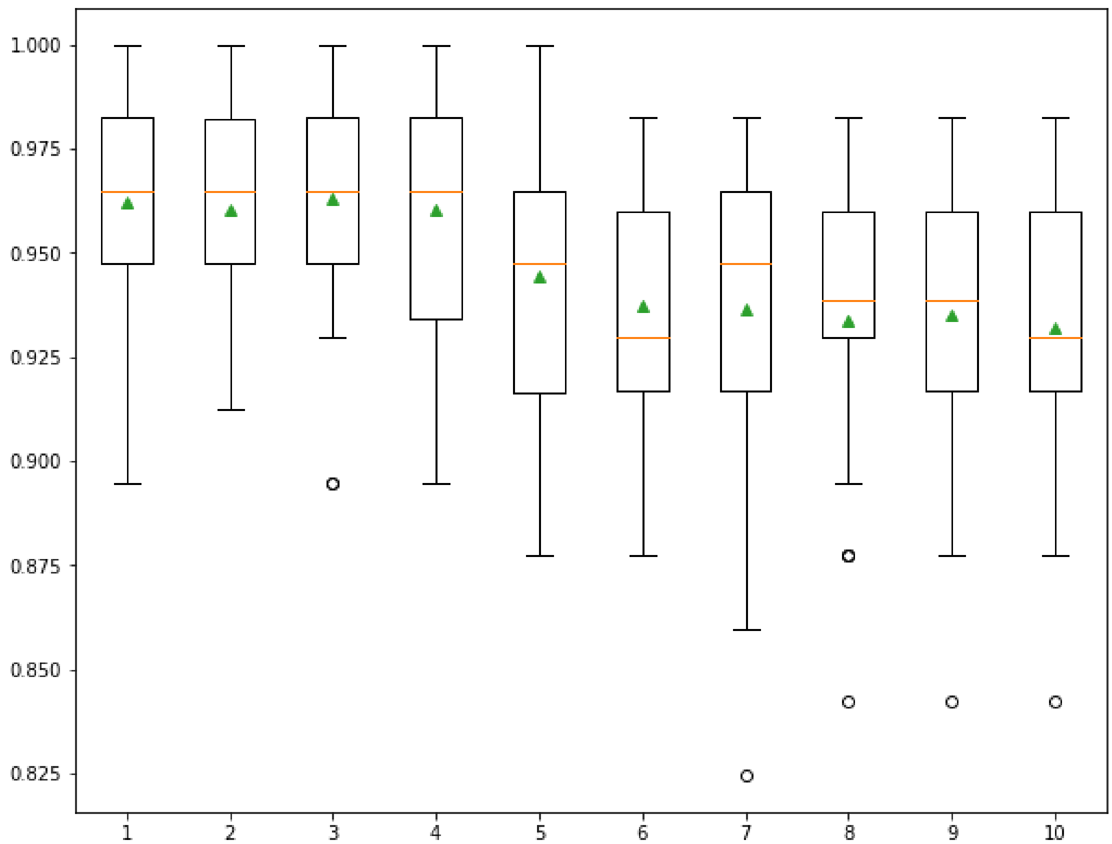
```
>0.0001 0.627 (0.007)
>0.0010 0.627 (0.007)
>0.0100 0.943 (0.030)
>0.1000 0.961 (0.028)
>1.0000 0.967 (0.023)
```



1.5.5 Choosing tree depth

[ ]:
```python
# get a list of models to evaluate
def get_models():
    models = dict()
    # define max tree depths to explore between 1 and 10
    for i in range(1,11):
        models[str(i)] = GradientBoostingClassifier(max_depth=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True);
```

```
>1 0.962 (0.024)
>2 0.960 (0.022)
>3 0.963 (0.026)
>4 0.960 (0.031)
>5 0.944 (0.031)
>6 0.937 (0.027)
>7 0.936 (0.038)
>8 0.934 (0.033)
>9 0.935 (0.031)
>10 0.932 (0.033)
```



1.5.6 Grid search for hyperparameters (The following cell takes very long to compute!)

[ ]:
```
# define the model with default hyperparameters
model = GradientBoostingClassifier()
# define the grid of values to search
grid = dict()
grid['n_estimators'] = [10, 50, 100, 500]
grid['learning_rate'] = [0.0001, 0.001, 0.01, 0.1, 1.0]
grid['subsample'] = [0.5, 0.7, 1.0]
grid['max_depth'] = [3, 7, 9]
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv,␣
↪scoring='accuracy')
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```python
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.970155 using {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 500,
'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 10, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 10, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 10, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 50, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 50, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 50, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 100, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 100, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 100, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 500, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 500, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 3,
'n_estimators': 500, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 10, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 10, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 10, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 50, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 50, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 50, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 100, 'subsample': 0.5}
```

```
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 100, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 100, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 500, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 500, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 7,
'n_estimators': 500, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 10, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 10, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 10, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 50, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 50, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 50, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 100, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 100, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 100, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 500, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 500, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.0001, 'max_depth': 9,
'n_estimators': 500, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 10, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 10, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 10, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 50, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 50, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 50, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 100, 'subsample': 0.5}
```

```
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 100, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 100, 'subsample': 1.0}
0.945520 (0.021934) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 500, 'subsample': 0.5}
0.943181 (0.023866) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 500, 'subsample': 0.7}
0.932665 (0.030049) with: {'learning_rate': 0.001, 'max_depth': 3,
'n_estimators': 500, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 10, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 10, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 10, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 50, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 50, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 50, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 100, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 100, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 100, 'subsample': 1.0}
0.947274 (0.022243) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 500, 'subsample': 0.5}
0.950219 (0.024885) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 500, 'subsample': 0.7}
0.937364 (0.030561) with: {'learning_rate': 0.001, 'max_depth': 7,
'n_estimators': 500, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 10, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 10, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 10, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 50, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 50, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 50, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 100, 'subsample': 0.5}
```

```
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 100, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 100, 'subsample': 1.0}
0.946690 (0.023358) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 500, 'subsample': 0.5}
0.950219 (0.024885) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 500, 'subsample': 0.7}
0.933271 (0.031124) with: {'learning_rate': 0.001, 'max_depth': 9,
'n_estimators': 500, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 10, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 10, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 10, 'subsample': 1.0}
0.941426 (0.024972) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 50, 'subsample': 0.5}
0.946126 (0.023098) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 50, 'subsample': 0.7}
0.932665 (0.032666) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 50, 'subsample': 1.0}
0.956662 (0.025541) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 100, 'subsample': 0.5}
0.953164 (0.028044) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 100, 'subsample': 0.7}
0.943807 (0.029799) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 100, 'subsample': 1.0}
0.961362 (0.026563) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 500, 'subsample': 0.5}
0.959607 (0.026848) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 500, 'subsample': 0.7}
0.954334 (0.029192) with: {'learning_rate': 0.01, 'max_depth': 3,
'n_estimators': 500, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 10, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 10, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 10, 'subsample': 1.0}
0.947274 (0.024441) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 50, 'subsample': 0.5}
0.947880 (0.025449) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 50, 'subsample': 0.7}
0.936779 (0.031524) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 50, 'subsample': 1.0}
0.960182 (0.024796) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 100, 'subsample': 0.5}
```

```
0.957843 (0.027412) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 100, 'subsample': 0.7}
0.938544 (0.031902) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 100, 'subsample': 1.0}
0.961351 (0.028084) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 500, 'subsample': 0.5}
0.959618 (0.030762) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 500, 'subsample': 0.7}
0.936790 (0.036644) with: {'learning_rate': 0.01, 'max_depth': 7,
'n_estimators': 500, 'subsample': 1.0}
0.627412 (0.006966) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 10, 'subsample': 0.5}
0.627412 (0.006966) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 10, 'subsample': 0.7}
0.627412 (0.006966) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 10, 'subsample': 1.0}
0.949029 (0.023745) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 50, 'subsample': 0.5}
0.949635 (0.025554) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 50, 'subsample': 0.7}
0.933271 (0.032730) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 50, 'subsample': 1.0}
0.957843 (0.026265) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 100, 'subsample': 0.5}
0.954334 (0.028864) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 100, 'subsample': 0.7}
0.932696 (0.032634) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 100, 'subsample': 1.0}
0.963116 (0.027322) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 500, 'subsample': 0.5}
0.959618 (0.029397) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 500, 'subsample': 0.7}
0.933866 (0.030558) with: {'learning_rate': 0.01, 'max_depth': 9,
'n_estimators': 500, 'subsample': 1.0}
0.950230 (0.028647) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
10, 'subsample': 0.5}
0.950230 (0.027935) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
10, 'subsample': 0.7}
0.940288 (0.030885) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
10, 'subsample': 1.0}
0.960182 (0.022633) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
50, 'subsample': 0.5}
0.959597 (0.029394) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
50, 'subsample': 0.7}
0.954334 (0.029886) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
50, 'subsample': 1.0}
0.960171 (0.026475) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
100, 'subsample': 0.5}
```

```
0.964275 (0.026232) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
100, 'subsample': 0.7}
0.961947 (0.027567) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
100, 'subsample': 1.0}
0.966614 (0.019922) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
500, 'subsample': 0.5}
0.965445 (0.020989) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
500, 'subsample': 0.7}
0.967220 (0.022974) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators':
500, 'subsample': 1.0}
0.957832 (0.022037) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
10, 'subsample': 0.5}
0.955462 (0.026768) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
10, 'subsample': 0.7}
0.936779 (0.032168) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
10, 'subsample': 1.0}
0.963116 (0.027682) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
50, 'subsample': 0.5}
0.961362 (0.027338) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
50, 'subsample': 0.7}
0.937375 (0.034654) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
50, 'subsample': 1.0}
0.961372 (0.028060) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
100, 'subsample': 0.5}
0.963701 (0.026400) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
100, 'subsample': 0.7}
0.931527 (0.041309) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
100, 'subsample': 1.0}
0.970155 (0.020339) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
500, 'subsample': 0.5}
0.968975 (0.021091) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
500, 'subsample': 0.7}
0.937959 (0.034373) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators':
500, 'subsample': 1.0}
0.950219 (0.029081) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
10, 'subsample': 0.5}
0.949050 (0.029879) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
10, 'subsample': 0.7}
0.930357 (0.031151) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
10, 'subsample': 1.0}
0.960777 (0.031613) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
50, 'subsample': 0.5}
0.956088 (0.032903) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
50, 'subsample': 0.7}
0.932112 (0.031236) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
50, 'subsample': 1.0}
0.961936 (0.028334) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
100, 'subsample': 0.5}
```

```
0.959597 (0.027246) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
100, 'subsample': 0.7}
0.933292 (0.032739) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
100, 'subsample': 1.0}
0.964871 (0.023082) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
500, 'subsample': 0.5}
0.967231 (0.019609) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
500, 'subsample': 0.7}
0.931537 (0.036571) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estimators':
500, 'subsample': 1.0}
0.923872 (0.035449) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
10, 'subsample': 0.5}
0.943807 (0.032165) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
10, 'subsample': 0.7}
0.943212 (0.034076) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
10, 'subsample': 1.0}
0.934419 (0.038401) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
50, 'subsample': 0.5}
0.950230 (0.023112) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
50, 'subsample': 0.7}
0.959597 (0.026454) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
50, 'subsample': 1.0}
0.890508 (0.163476) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
100, 'subsample': 0.5}
0.958396 (0.024261) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
100, 'subsample': 0.7}
0.966040 (0.023495) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
100, 'subsample': 1.0}
0.862406 (0.202078) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
500, 'subsample': 0.5}
0.955524 (0.027809) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
500, 'subsample': 0.7}
0.962521 (0.025521) with: {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators':
500, 'subsample': 1.0}
0.925637 (0.031299) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
10, 'subsample': 0.5}
0.939703 (0.027419) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
10, 'subsample': 0.7}
0.940309 (0.028457) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
10, 'subsample': 1.0}
0.936163 (0.028838) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
50, 'subsample': 0.5}
0.954929 (0.028159) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
50, 'subsample': 0.7}
0.943233 (0.030921) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
50, 'subsample': 1.0}
0.942533 (0.033282) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
100, 'subsample': 0.5}
```

```
0.962531 (0.028189) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
100, 'subsample': 0.7}
0.936800 (0.031530) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
100, 'subsample': 1.0}
0.923266 (0.032197) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
500, 'subsample': 0.5}
0.914515 (0.034064) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
500, 'subsample': 0.7}
0.913962 (0.030749) with: {'learning_rate': 1.0, 'max_depth': 7, 'n_estimators':
500, 'subsample': 1.0}
0.934398 (0.038233) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
10, 'subsample': 0.5}
0.940320 (0.031549) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
10, 'subsample': 0.7}
0.932696 (0.035351) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
10, 'subsample': 1.0}
0.942627 (0.031318) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
50, 'subsample': 0.5}
0.958991 (0.024148) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
50, 'subsample': 0.7}
0.932686 (0.032314) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
50, 'subsample': 1.0}
0.943233 (0.027788) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
100, 'subsample': 0.5}
0.956694 (0.023403) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
100, 'subsample': 0.7}
0.929198 (0.035176) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
100, 'subsample': 1.0}
0.909284 (0.042929) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
500, 'subsample': 0.5}
0.904031 (0.046308) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
500, 'subsample': 0.7}
0.912176 (0.028548) with: {'learning_rate': 1.0, 'max_depth': 9, 'n_estimators':
500, 'subsample': 1.0}
```