МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ЗВІТ

З ЛАБОРАТОРНОЇ РОБОТИ №5

З ДИСЦИПЛІНИ: « ПРОГРАМУВАННЯ ІНТЕЛЕКТУАЛЬНИХ ІНФОРМАЦІЙНИХ СИСТЕМ»

Виконав:
ІП-15 Мєшков А.І.

Перевірив:
Курченко О.А.

Київ 2023

ЗАВДАННЯ

1 Потюнити параметри за цим туторіалом. Порівняти з дефолтними. Зрозуміти роботу алгоритмічного підбору параметрів.
https://www.kaggle.com/code/shreayan98c/hyperparameter-tuning-tutorial

2 Зробити ансамблі і бустинг за цим туторіалом. Пояснити відмінності. Порівняти з просто тюнингом. Пояснити коли і що використовувати.
https://www.kaggle.com/code/pavansanagapati/ensemble-learning-techniques-tutorial

ХІД РОБОТИ

```
import pandas as pd
import numpy as np
from sklearn import metrics, svm
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_predict
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier, AdaBoostClassifier, VotingClassifier
from sklearn.neighbors import KNeighborsClassifier
import warnings

warnings.filterwarnings("ignore")

df = pd.read_csv("data.csv")
df.drop(['Unnamed: 32', 'id'], axis=1, inplace=True)
df.head()

X = df.drop(['diagnosis'], axis = 1)
y = df['diagnosis']
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.3, random_state = 42)
print("Size of training set:", X_train.shape)
print("Size of test set:", X_test.shape)
```

Output:

Size of training set: (398, 30)

Size of test set: (171, 30)

```
Логістична регресія
lr = LogisticRegression()
lr.fit(X_train, y_train)
y_pred = lr.predict(X_test)
acc_lr = metrics.accuracy_score(y_test, y_pred)
print( 'Accuracy of initial Logistic Regression model : ', acc_lr )
```

Output:

Accuracy of initial Logistic Regression model : 0.9707602339181286

```
lr = LogisticRegression()
parameters = {'penalty': ['l1', 'l2','elasticnet', 'none'],
              'C': [0.001, 0.01, 0.1, 1, 10, 100],
```

```
              'solver': ['lbfgs', 'newton-cg','liblinear', 'sag', 'saga'],
              'max_iter': [100, 200, 500, 1000]
             }

grid_obj = GridSearchCV(lr, parameters, cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

lr = grid_obj.best_estimator_
print(lr.fit(X_train, y_train))
y_pred = lr.predict(X_test)
acc_lr_t = metrics.accuracy_score(y_test, y_pred)
print( 'Accuracy of tuned Logistic Regression model : ', acc_lr_t)
```

Output:

```
LogisticRegression(C=10, penalty='l1', solver='liblinear')
Accuracy of tuned Logistic Regression model :  0.9707602339181286
```

```
Дерево рішень
dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)
y_pred = dt.predict(X_test)
acc_dt = metrics.accuracy_score(y_test, y_pred)
print( 'Accuracy of initial Decision Tree model : ', acc_dt)
```

Output:

```
Accuracy of initial Decision Tree model :  0.9298245614035088
```

```
dt = DecisionTreeClassifier()
parameters = {'max_features': ['log2', 'sqrt','auto'],
              'criterion': ['entropy', 'gini'],
              'max_depth': [2, 3, 5, 10, 50],
              'min_samples_split': [2, 3, 50, 100],
              'min_samples_leaf': [1, 5, 8, 10]
             }

grid_obj = GridSearchCV(dt, parameters)
grid_obj = grid_obj.fit(X_train, y_train)

dt = grid_obj.best_estimator_
print(dt.fit(X_train, y_train))
y_pred = dt.predict(X_test)
acc_dt_t = metrics.accuracy_score(y_test, y_pred)
print( 'Accuracy of tuned Decision Tree model : ', acc_dt_t)
```

Output:

DecisionTreeClassifier(criterion='entropy', max_depth=10, max_features='log2')
Accuracy of tuned Decision Tree model : 0.9473684210526315

Випадковий ліс

```
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
acc_rf = metrics.accuracy_score(y_test, y_pred)
print( 'Accuracy of initial Random Forest model : ', acc_rf)
```

Output
Accuracy of initial Random Forest model : 0.9649122807017544

```
rf = RandomForestClassifier()
parameters = {'n_estimators': [4, 6, 9, 10, 15],
              'max_features': ['log2', 'sqrt','auto'],
              'criterion': ['entropy', 'gini'],
              'max_depth': [2, 3, 5, 10],
              'min_samples_split': [2, 3, 5],
              'min_samples_leaf': [1, 5, 8]
             }

grid_obj = GridSearchCV(rf, parameters)
grid_obj = grid_obj.fit(X_train, y_train)

rf = grid_obj.best_estimator_
print(rf.fit(X_train, y_train))
y_pred = rf.predict(X_test)
acc_rf_t = metrics.accuracy_score(y_test, y_pred)
print( 'Accuracy of tuned Random Forest model : ', acc_rf_t)
```

Output:
RandomForestClassifier(max_depth=10, max_features='log2', min_samples_leaf=5, n_estimators=10)
Accuracy of tuned Random Forest model : 0.9766081871345029

Support Vector Machine

```
sc = StandardScaler()
X_train_sc = sc.fit_transform(X_train)
X_test_sc = sc.transform(X_test)

svc = svm.SVC()

svc.fit(X_train_sc, y_train)
y_pred = svc.predict(X_test_sc)
```

```
acc_svm = metrics.accuracy_score(y_test, y_pred)
print( 'Accuracy of initial SVM model : ', acc_svm)
```
Output:
Accuracy of initial SVM model : 0.9766081871345029

```
sc = StandardScaler()
X_train_sc1 = sc.fit_transform(X_train)
X_test_sc1 = sc.transform(X_test)

svc = svm.SVC()

parameters = [
    {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']},
    {'C': [1, 10, 100, 1000], 'kernel': ['linear']}
]

grid_obj = GridSearchCV(svc, parameters)
grid_obj = grid_obj.fit(X_train_sc1, y_train)

svc = grid_obj.best_estimator_
print(svc.fit(X_train_sc1, y_train))
y_pred = svc.predict(X_test_sc1)
acc_svm_t = metrics.accuracy_score(y_test, y_pred)
print( 'Accuracy of tuned SVM model : ', acc_svm_t)
```
Output:
SVC(C=1000, gamma=0.0001)
Accuracy of tuned SVM model : 0.9824561403508771

```
K-Neighbours
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
acc_knn= metrics.accuracy_score(y_test, y_pred)
print( 'Accuracy of initial KNN model : ', acc_knn)
```
Output:
Accuracy of initial KNN model : 0.9590643274853801

```
knn = KNeighborsClassifier()
parameters = {'n_neighbors': [3, 4, 5, 10],
              'weights': ['uniform', 'distance'],
              'algorithm' : ['auto', 'ball_tree', 'kd_tree', 'brute'],
              'leaf_size' : [10, 20, 30, 50]
             }
```

```
grid_obj = GridSearchCV(knn, parameters)
grid_obj = grid_obj.fit(X_train, y_train)

knn = grid_obj.best_estimator_
print(knn.fit(X_train, y_train))
y_pred = knn.predict(X_test)
acc_knn_t = metrics.accuracy_score(y_test, y_pred)
print( 'Accuracy of tuned KNN model : ', acc_knn_t)
```

Output:

KNeighborsClassifier(leaf_size=10)

Accuracy of tuned KNN model : 0.9590643274853801

```
models = pd.DataFrame({
    'Model': ['Logistic Regression', 'Decision Tree', 'Random Forest', 'Support Vector Machines',
              'K - Nearest Neighbors'],
    'Initial Score': [acc_lr, acc_dt, acc_rf, acc_svm, acc_knn],
    'Tuned Score': [acc_lr_t, acc_dt_t, acc_rf_t, acc_svm_t, acc_knn_t]})
models.sort_values(by='Tuned Score', ascending=False)
```

Output:

	Model	Initial Score	Tuned Score
3	Support Vector Machines	0.976608	0.982456
2	Random Forest	0.964912	0.976608
0	Logistic Regression	0.970760	0.970760
4	K - Nearest Neighbors	0.959064	0.959064
1	Decision Tree	0.929825	0.947368

```
Part 2
label_encoder = LabelEncoder()
df['diagnosis'] = label_encoder.fit_transform(df['diagnosis'])
y = df['diagnosis']
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.3, random_state = 333)
X_train.shape, X_test.shape

Max Voting / Voting Classifier
lr_clf = LogisticRegression(solver='lbfgs', multi_class='multinomial', random_state=42)
dt_clf = DecisionTreeClassifier(random_state=42)
svm_clf = svm.SVC(probability=True, gamma='scale', random_state=42)
```

```
hard_voting_clf = VotingClassifier(estimators=[('lr', lr_clf), ('dt', dt_clf),
('svm', svm_clf)], voting='hard')
hard_voting_clf.fit(X_train, y_train)

y_pred_hard = hard_voting_clf.predict(X_test)
acc_hard = metrics.accuracy_score(y_test, y_pred_hard)

soft_voting_clf = VotingClassifier(estimators=[('lr', lr_clf), ('dt', dt_clf),
('svm', svm_clf)], voting='soft')
soft_voting_clf.fit(X_train, y_train)

y_pred_soft = soft_voting_clf.predict(X_test)
acc_soft = metrics.accuracy_score(y_test, y_pred_soft)

acc_hard, acc_soft
```

Output:

(0.9415204678362573, 0.9649122807017544)

```
Averaging
lr_clf.fit(X_train, y_train)
dt_clf.fit(X_train, y_train)
svm_clf.fit(X_train, y_train)

lr_probs = lr_clf.predict_proba(X_test)
dt_probs = dt_clf.predict_proba(X_test)
svm_probs = svm_clf.predict_proba(X_test)

avg_probs = (lr_probs + dt_probs + svm_probs)/3
avg_prediction = (avg_probs[:, 1] >= 0.5).astype(int)

acc_avg = metrics.accuracy_score(y_test, avg_prediction)
acc_avg
```

Output:

0.9649122807017544

```
Weighted Averaging
weight_lr = 0.4
weight_dt = 0.15
weight_svm = 0.45

weighted_avg_probs = (lr_probs * weight_lr + dt_probs * weight_dt + svm_probs *
weight_svm) / (weight_lr + weight_dt + weight_svm)
```

```
weighted_avg_prediction = (weighted_avg_probs[:, 1] >= 0.5).astype(int)

acc_weighted_avg = metrics.accuracy_score(y_test, weighted_avg_prediction)
acc_weighted_avg
```

Output:

0.9473684210526315

```
Stacking
lr_clf.fit(X_train, y_train)
dt_clf.fit(X_train, y_train)
svm_clf.fit(X_train, y_train)

lr_oof_pred = cross_val_predict(lr_clf, X_train, y_train, cv=5,
method="predict_proba")
dt_oof_pred = cross_val_predict(dt_clf, X_train, y_train, cv=5,
method="predict_proba")
svm_oof_pred = cross_val_predict(svm_clf, X_train, y_train, cv=5,
method="predict_proba")

stacked_features_train = np.column_stack((lr_oof_pred[:, 1], dt_oof_pred[:, 1],
svm_oof_pred[:, 1]))

meta_model = LogisticRegression()
meta_model.fit(stacked_features_train, y_train)

lr_pred_test = lr_clf.predict_proba (X_test)[:, 1]
dt_pred_test = dt_clf.predict_proba (X_test)[:, 1]
svm_pred_test = svm_clf.predict_proba (X_test)[:, 1]

stacked_features_test = np.column_stack ((lr_pred_test, dt_pred_test,
svm_pred_test))
final_pred = meta_model.predict(stacked_features_test)
acc_stacking = metrics.accuracy_score(y_test, final_pred)
acc_stacking
```

Output:

0.9649122807017544

```
Blending
X_train_blending, X_val, y_train_blending, y_val = train_test_split(X_train,
y_train, test_size=0.3, random_state=0)

lr_clf.fit(X_train_blending, y_train_blending)
dt_clf.fit(X_train_blending, y_train_blending)
```

```
svm_clf.fit(X_train_blending, y_train_blending)

lr_pred_val = lr_clf.predict_proba(X_val)[:, 1]
dt_pred_val = dt_clf.predict_proba(X_val)[:, 1]
svm_pred_val = svm_clf.predict_proba(X_val)[:, 1]

blender_features = np.column_stack((lr_pred_val, dt_pred_val, svm_pred_val))

blender = LogisticRegression()
blender.fit(blender_features, y_val)

lr_pred_test = lr_clf.predict_proba(X_test)[:, 1]
dt_pred_test = dt_clf.predict_proba (X_test)[:, 1]
svm_pred_test = svm_clf.predict_proba (X_test)[:, 1]

blender_test_features = np.column_stack((lr_pred_test, dt_pred_test,
svm_pred_test))

final_blend_pred = blender.predict(blender_test_features)
acc_blending = metrics.accuracy_score(y_test, final_blend_pred)
acc_blending
```

Output:

0.9532163742690059

```
Bagging
base_model = DecisionTreeClassifier(random_state=42)

bagging_clf = BaggingClassifier(base_estimator=base_model, n_estimators=100,
random_state=42)

bagging_clf.fit(X_train, y_train)

bagging_pred = bagging_clf.predict(X_test)
acc_bagging = metrics.accuracy_score(y_test, bagging_pred)
acc_bagging
```

Output:

0.9181286549707602

```
Boosting
ada_boost_clf = AdaBoostClassifier(base_estimator=base_model, n_estimators=100,
random_state=42)

ada_boost_clf.fit(X_train, y_train)
```

```
ada_boost_pred = ada_boost_clf.predict(X_test)
acc_ada_boost = metrics.accuracy_score(y_test, ada_boost_pred)
acc_ada_boost
```

Output:

0.8947368421052632

```
df_res2 = pd.DataFrame({
'Ensemble': ['Max Voting / Voting Classifier hard', 'Max Voting / Voting Classifier soft', 'Averaging', 'Weighted Averaging', 'Stacking', 'Blending', 'Bagging', 'Boosting'],
'Score': [acc_hard, acc_soft, acc_avg, acc_weighted_avg, acc_stacking, acc_blending, acc_bagging, acc_ada_boost]})
df_res2.sort_values (by='Score', ascending=False)
```

Output:

	Ensemble	Score
1	Max Voting / Voting Classifier soft	0.964912
2	Averaging	0.964912
4	Stacking	0.964912
5	Blending	0.953216
3	Weighted Averaging	0.947368
0	Max Voting / Voting Classifier hard	0.941520
6	Bagging	0.918129
7	Boosting	0.894737

Висновок:

Part 1:

1. **Метод опорних векторів (SVM):**
    a. Початковий показник: 0.976608
    b. Налаштований показник: 0.982456
    c. Висновок: Налаштування гіперпараметрів призвело до невеликого покращення точності для моделі методу опорних векторів.
2. **Випадковий ліс:**
    a. Початковий показник: 0.964912
    b. Налаштований показник: 0.976608

c. Висновок: Налаштування гіперпараметрів призвело до поліпшення точності для моделі випадкового лісу, що свідчить про те, що обрані параметри краще підходять для даних.

3. **Логістична регресія:**
    a. Початковий показник: 0.970760
    b. Налаштований показник: 0.970760
    c. Висновок: Точність залишилася незмінною після налаштування гіперпараметрів для логістичної регресії. Це свідчить про те, що параметри за замовчуванням вже були ефективними для цієї моделі.
4. **K - Найближчі сусіди (KNN):**
    a. Початковий показник: 0.959064
    b. Налаштований показник: 0.959064
    c. Висновок: Налаштування гіперпараметрів не суттєво вплинуло на точність моделі K-найближчих сусідів. Здається, що параметри за замовчуванням працюють добре.
5. **Дерево рішень:**
    a. Початковий показник: 0.929825
    b. Налаштований показник: 0.947368
    c. Висновок: Налаштування гіперпараметрів призвело до помітного поліпшення точності для моделі дерева рішень. Налаштовані параметри краще враховують основні закономірності в даних.

Загалом налаштування гіперпараметрів мало різний вплив на різні моделі. Важливо розуміти характеристики кожного алгоритму та набору даних, щоб визначити, чи необхідне налаштування. Результати демонструють важливість точного налаштування параметрів для покращення ефективності моделей машинного навчання.

Part 2:

Максимальне голосування (Voting Classifier - м'яке та жорстке):

- М'яке голосування: Модель отримує ваговану суму ймовірностей класів від усіх базових моделей і вибирає клас з найвищою ймовірністю.
- Жорстке голосування: Клас, який отримав більшість голосів серед базових моделей, обирається.

Усереднення (Averaging):

- Результат усереднюється або вагується (залежно від методу) для передбачень з різних базових моделей.

Стекінг (Stacking):

- Використовується додатковий модель верхнього рівня для об'єднання передбачень вихідних моделей.

Змішування (Blending):

- Також використовує ваговане усереднення передбачень вихідних моделей, але з використанням окремого набору даних для ваг.

Взважене усереднення (Weighted Averaging):

- Тип усереднення, де ваги надаються кожній моделі в залежності від їхньої важливості.

Беггінг (Bagging):

- Використовується для стабілізації та зменшення варіації моделі шляхом об'єднання декількох екземплярів одного алгоритму, навчених на різних підмножинах даних.

Бустинг (Boosting):

- Моделі навчаються послідовно, кожна нова модель намагається виправити помилки попередньої, підвищуючи точність.

1. **Порівняння з простим тюнінгом:**
   - Простий тюнінг параметрів спрямований на оптимізацію параметрів одного алгоритму.
   - Ансамблі та бустинг створюються на основі декількох базових моделей, що може призвести до поліпшення результатів завдяки комбінації різноманітності та сильній узгодженості моделей.

2. **Коли використовувати:**
    - Ансамбль:
        - Коли є декілька сильних моделей і хочете використовувати їх комбінацію для отримання кращих результатів.
        - Якщо моделі різні та непохідні, щоб забезпечити різноманітність.
    - Бустинг:
        - Коли є багато слабких моделей і хочете послідовно вдосконалювати їхню точність.
        - Коли важлива точність за рахунок часу навчання.
    - Простий тюнінг:
        - Коли бажаєте оптимізувати конкретний алгоритм для ваших потреб і вам не потрібна комбінація моделей.