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',

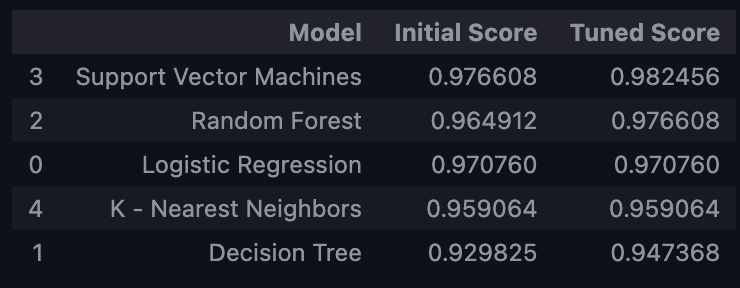
'K - Nearest Neighbors'],

'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:**



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**

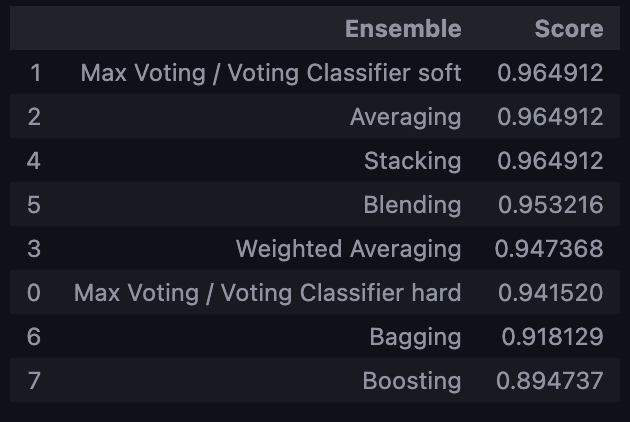
df\_res2 = pd.DataFrame({

'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:**



Висновок:

Part 1:

1. **Метод опорних векторів (SVM):**
   1. Початковий показник: 0.976608
   2. Налаштований показник: 0.982456
   3. Висновок: Налаштування гіперпараметрів призвело до невеликого покращення точності для моделі методу опорних векторів.
2. **Випадковий ліс:**
   1. Початковий показник: 0.964912
   2. Налаштований показник: 0.976608
   3. 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:**
   1. Початковий показник: 0.970760
   2. Налаштований показник: 0.970760
   3. 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):**
   1. Початковий показник: 0.959064
   2. Налаштований показник: 0.959064
   3. Висновок: Налаштування гіперпараметрів не суттєво вплинуло на точність моделі K-найближчих сусідів. Здається, що параметри за замовчуванням працюють добре.
5. **Дерево рішень:**
   1. Початковий показник: 0.929825
   2. Налаштований показник: 0.947368
   3. 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.

1. **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.