به نام خدا

بخش اول :

ابتدا داده های مورد نظر را خوانده و داده های عددی و categorical را جدا کرده و missing value های هر کدام را متناسب با نوع داده هندل می کنیم.برای این کار میتوان از میانگین و میانه و مقدار ثابت یا بیشترین تکرار استفاده کرد.

```
df = pd.read_csv('heart_disease_uci.csv')

#cat col
categorical_cols = df.select_dtypes(include=['object']).columns.tolist()

#missing values for num col
numerical_cols = df.select_dtypes(include=['float64', 'int64']).columns.tolist()
imputer_numeric = SimpleImputer(strategy='mean')
df[numerical_cols] = pd.DataFrame(imputer_numeric.fit_transform(df[numerical_cols]), columns=numerical_cols)


#cat col to num
label_encoder = LabelEncoder()
df[categorical_cols] = df[categorical_cols].apply(label_encoder.fit_transform)

#missing values for cat col
imputer_categorical = ColumnTransformer(
    transformers=[
        ('sex', SimpleImputer(strategy='most_frequent'), ['sex']),
        ('dataset', SimpleImputer(strategy='most_frequent'), ['dataset']),
        ('cp', SimpleImputer(strategy='median'), ['cp']),
        ('fbs', SimpleImputer(strategy='most_frequent'), ['fbs']),
        ('restecg', SimpleImputer(strategy='most_frequent'), ['restecg']),
        ('exang', SimpleImputer(strategy='most_frequent'), ['exang']),
        ('slope', SimpleImputer(strategy='mean'), ['slope']),
        ('thal', SimpleImputer(strategy='most_frequent'), ['thal']),
    ]
)
df[categorical_cols] = pd.DataFrame(imputer_categorical.fit_transform(df[categorical_cols]), columns=categorical_cols)
```

بعد از پردازش, داده ها را نرمال کرده و مجموعه آموزش و تست را جدا می کنیم و اعداد [10, 15, 18, 25, 50, 100, 200] را برای تعداد base classifier ها امتحان می کنیم و با استفاده از GridSearchCV، میبینیم که n_estimators: 15 بهترین انتخاب است.

```
df['num'] = df['num'].apply(lambda x: 0 if x == 0 else 1)

X = df.drop('num', axis=1)
y = df['num']

# Normalize
scaler = StandardScaler()
normalized_data = scaler.fit_transform(X)
X = pd.DataFrame(normalized_data, columns=X.columns)

#split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
adaboost_classifier = AdaBoostClassifier()

param_grid = {
    'n_estimators': [10, 15, 18, 25, 50, 100, 200],
}

#GridSearchCV
grid_search = GridSearchCV(adaboost_classifier, param_grid, cv=5, scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train, y_train)


print("Best Parameters:", grid_search.best_params_)
print("Best Accuracy:", grid_search.best_score_)

test_accuracy = grid_search.best_estimator_.score(X_test, y_test)
print("Test Set Accuracy:", test_accuracy)
```

```
Best Parameters: {'n_estimators': 15}
Best Accuracy: 0.8736440522154808
Test Set Accuracy: 0.842391304347826
```

در ادامه با استفاده از cross-validation folds الگوریتم را با مقادیر مختلف cv اجرا می کنیم.

```
adaboost_classifier = AdaBoostClassifier(n_estimators=15)
#Cross-Validation
k_values = [3, 5, 7]

for k in k_values:
    cross_val_scores = cross_val_score(adaboost_classifier, X_train, y_train, cv=k, scoring='accuracy')
    print(f'Cross-Validation Accuracy (k={k}): {cross_val_scores.mean()}')


adaboost_classifier.fit(X_train, y_train)
test_accuracy = adaboost_classifier.score(X_test, y_test)
print("Test Set Accuracy:", test_accuracy)
```

```
Cross-Validation Accuracy (k=3): 0.86548310381063
Cross-Validation Accuracy (k=5): 0.8736440522154808
Cross-Validation Accuracy (k=7): 0.8696187909125914
Test Set Accuracy: 0.842391304347826
```

n_estimators یک hyperparameter است که تعداد weak learner ها را مشخص می کند.
افزایش تعداد weak learner ها به طور کلی عملکرد مجموعه AdaBoost را تا یک نقطه خاص بهبود می بخشد. با این حال، اضافه کردن بیش از حد آنها می تواند منجر به overfitting شود، که در آن مدل به جای تعمیم خوب به داده های دیده نشده، شروع به حفظ داده های آموزشی می کند.
Cross-validation برای ارزیابی میزان تعمیم مدل به داده های جدید و دیده نشده مهم است. در این روش مجموعه داده را به چند قسمت تقسیم می‌کنیم، مدل را با برخی قسمت‌ها آموزش می‌دهیم و با برخی دیگر اعتبارسنجی می‌کنیم .انتخاب تعداد بخش ها می تواند بر قابلیت اطمینان ارزیابی مدل تأثیر بگذارد. تعداد بیشتر قطعات می‌تواند تخمین قوی‌تری از عملکرد ارائه دهد، اما می‌تواند از نظر محاسباتی سنگین باشد.

بخش دوم :

1- روش Stacking یکی از روش‌های آ ensamble learning (یا یادگیری تجمعی) در زمینه یادگیری ماشین است. در این روش، مدل‌های مختلف به نام‌های "base learners" یا "base models" انتخاب می‌شوند و یک مدل برخط (meta-model) بر روی خروجی‌های این مدل‌های پایه آموزش داده می‌شود.
Stacking می‌تواند بهبودی در عملکرد مدل نهایی نسبت به مدل‌های پایه داشته باشدو از نقاط قوت مدل‌های مختلف بهره‌مند شود و از نقاط ضعف هر کدام جلوگیری کند.

: Stacking

1.آموزش مدل‌های پایه (Base Models): ابتدا، چندین مدل یادگیری ماشین مستقل (مثلاً درخت تصمیم، ماشین‌های پشتیبان، رگرسیون خطی و غیره) آموزش داده می‌شوند. هر کدام از این مدل‌ها می‌توانند با الگوریتم‌ها و پارامترهای مختلف آموزش داده شوند.
2. تولید پیش‌بینی‌ها: سپس، این مدل‌های پایه بر روی داده‌های آموزش و یا داده‌های تست پیش‌بینی می‌کنند.
3. آموزش مدل پیش‌بینی (Meta-Model): خروجی‌های پیش‌بینی شده توسط مدل‌های پایه به عنوان ویژگی‌های ورودی برای مدل پیش‌بینی (meta-model) وارد می‌شوند.
Meta-model می‌تواند یک مدل ساده باشد (مثل رگرسیون خطی) که بر اساس پیش‌بینی‌های مدل‌های پایه، یک مدل نهایی برای تصمیم‌گیری ایجاد می‌کند.
4. پیش‌بینی با استفاده از Meta-Model: حالا با داشتن meta-model، می‌توان از آن برای پیش‌بینی خروجی نهایی استفاده کرد

```python
#Step 1: Prepare the Data => first cell

#Step 2: Model Selection
base_models = [
    ('svm', SVC(kernel='linear', probability=True)),
    ('decision_tree', DecisionTreeClassifier()),
    ('gradient_boosting', GradientBoostingClassifier())
]

#Step 3: Training the Base Models
for name, model in base_models:
    model.fit(X_train, y_train)

#Step 4: Predictions on the Validation Set
base_model_predictions = {name: model.predict(X_test) for name, model in base_models}

#Step 5: Developing a Meta Model
meta_model = LogisticRegression()

#Step 6: Training the Meta Model
meta_model_input = pd.DataFrame(base_model_predictions)
meta_model.fit(meta_model_input, y_test)

#Step 7: Making Test Set Predictions
stacking_input = pd.DataFrame({name: model.predict(X_test) for name, model in base_models})
stacking_predictions = meta_model.predict(stacking_input)

#Step 8: Model Evaluation
stacking_accuracy = accuracy_score(y_test, stacking_predictions)
stacking_classification_report = classification_report(y_test, stacking_predictions)

print("Accuracy:", stacking_accuracy)
print("Classification:\n", stacking_classification_report)
```

```
Accuracy: 0.8695652173913043
Classification:
              precision    recall  f1-score   support

           0       0.83      0.85      0.84        75
           1       0.90      0.88      0.89       109

    accuracy                           0.87       184
   macro avg       0.86      0.87      0.87       184
weighted avg       0.87      0.87      0.87       184
```

4- بیایید یک مثال ساده از استفاده از Stacking را در نظر بگیریم که نشان دهد چگونه می‌توان اهمیت هر مدل پایه را تشخیص داد.

مثال: تشخیص ایمیل اسپم

فرض کنید ما یک مسئله تشخیص اسپم ایمیل داریم. می‌خواهیم از Stacking برای ترکیب نتایج مدل‌های مختلف بهبود دهیم و اهمیت هر مدل پایه را بررسی کنیم.

مدل‌های پایه:(Base Models)

مدل درخت تصمیم :(Decision Tree) ممکن است برخی از ویژگی‌های خاصی در متن ایمیل‌ها را به خوبی تشخیص دهد.

مدل ماشین‌های پشتیبان :(Support Vector Machine) ممکن است بر روی ساختار مخفی در داده تمرکز کند.
مدل نزدیک‌ترین همسایه :(K-Nearest Neighbors) ممکن است بر اساس شباهت به نمونه‌های مشابه تصمیم بگیرد.

آموزش مدل پیش‌بینی:(Meta-Model)
مدل رگرسیون لجستیک :(Logistic Regression) به عنوان مدل پیش‌بینی اصلی (meta-model) انتخاب می‌شود.
ورودی‌های این مدل، خروجی‌های مدل‌های پایه می‌شوند.

ترکیب نتایج:
مدل رگرسیون لجستیک آموزش داده می‌شود تا بر اساس خروجی‌های مدل‌های پایه، تصمیم‌های نهایی را بگیرد.

اهمیت مدل‌های پایه:
با تحلیل وزن‌هایی که مدل رگرسیون لجستیک به ویژگی‌های مختلف نسبت می‌دهد، می‌توانیم اهمیت هر مدل پایه را بسنجیم.
مثلاً، اگر ویژگی‌های مشخصی که توسط مدل درخت تصمیم تشخیص داده شده است، وزن بالایی داشته باشند، این نشان‌دهنده این است که درخت تصمیم در تصمیم‌گیری نهایی مدل بسیار موثر بوده است.
این روش نه تنها به بهبود کارایی نهایی مدل کمک می‌کند بلکه اطلاعاتی ارائه می‌دهد که می‌تواند به ما درک بهتری از نحوه عملکرد هر مدل پایه و ویژگی‌های مهم داده‌ها کمک کند.

-5
استفاده از مدل‌های پایه متنوع: در استفاده ازStacking ، مدل‌های پایه معمولاً به صورت مستقل انتخاب می‌شوند و ممکن است از الگوریتم‌ها و پارامترهای مختلف استفاده کنند. این تنوع در مدل‌های پایه می‌تواند کمک کند که هر مدل به نحوی دیگری اطلاعات مهم را از داده‌ها استخراج کند.

استفاده ازMeta-Model : وجود یک مدل پیش‌بینی (meta-model) که بر روی خروجی‌های مدل‌های پایه آموزش داده می‌شود، می‌تواند از overfittingجلوگیری کند. Meta-model تلاش می‌کند اطلاعات مفید و اصلی را از خروجی‌های مدل‌های پایه استخراج کند و از آنها برای ساختن یک مدل نهایی استفاده کند، اما به دلیل ساختار ساده‌تر، احتمال overfitting در آن کمتر است.

استفاده ازCross-Validation : با استفاده ازcross-validation ، احتمال overfitting کاهش می‌یابد. در هر مرحله ازcross-validation ، مدل پایه روی یک قسمت از داده‌ها آموزش داده می‌شود و بر روی بخش دیگری از داده‌ها ارزیابی می‌شود. این کار به کاهش احتمال memorization داده‌ها و افزایش قابلیت تعمیم مدل کمک می‌کند.

-6

: Bagging (Bootstrap Aggregating)
روش: درBagging ، چندین مدل مشابه (مثل درخت تصمیم یا ماشین پشتیبان) به صورت مستقل از یکدیگر آموزش می‌بینند.
تولید داده آموزشی: برای هر مدل، داده‌ها به صورت تصادفی با جایگزینی انتخاب می‌شوند.(bootstrap sampling)
ترکیب نتایج: خروجی‌های مدل‌ها ترکیب شده و میانگین یا رای‌گیری بر اساس اکثریت به عنوان خروجی نهایی در نظر گرفته می‌شود.
مثال: Random Forest

: Boosting
روش: درBoosting ، مدل‌های ضعیف به صورت متوالی آموزش داده می‌شوند. هر مدل به خطای مدل قبلی توجه دارد و سعی در تصحیح آن خطا دارد.
تولید داده آموزشی: تاکید بر داده‌هایی است که تاکنون به درستی تشخیص داده نشده‌اند یا خطای زیادی دارند.
ترکیب نتایج: خروجی‌های مدل‌ها با وزن‌دهی ترکیب شده و خطاها متناسب با اهمیتشان در آموزش مدل‌های بعدی مدل‌های ضعیف کاهش

می‌یابد.
مثال: AdaBoost، Gradient Boosting

Stacking :
روش: درStacking ، چندین مدل به صورت متوالی آموزش داده می‌شوند و یک مدل پیش‌بینی (meta-model) بر اساس خروجی‌های این مدل‌ها آموزش می‌بیند.
تولید داده آموزشی: خروجی‌های مدل‌های پایه به عنوان ویژگی‌ها به مدل پیش‌بینی وارد می‌شوند
ترکیب نتایج: مدل پیش‌بینی به عنوان مدل نهایی استفاده می‌شود. مدل‌های پایه به صورت مستقل آموزش می‌بینند و خروجی‌های آنها به عنوان ویژگی‌ها برای مدل پیش‌بینی وارد می‌شوند.
مثال: مدل‌های متنوع مانند درخت تصمیم، ماشین پشتیبان، رگرسیون خطی.

بخش سوم :

الف. Ensemble Learning و نقش XGBoost:

Ensemble Learning:

Ensemble Learning یک روش در یادگیری ماشین است که از ترکیب چندین مدل (یا یادگیری از چندین مدل) به منظور بهبود عملکرد و افزایش دقت در پیش‌بینی استفاده می‌کند. Ensemble Learning می‌تواند از دو نوع مختلف باشد:

1. Bagging (Bootstrap Aggregating): مانند Random Forest که از ترکیب چندین درخت تصمیم با bootstrap sampling استفاده می‌کند.

2. Boosting: مانند AdaBoost یا XGBoost که مدل‌ها به صورت متوالی آموزش می‌بینند و تمرکز بر خطای مدل‌های قبلی دارند.

XGBoost:

XGBoost یک الگوریتم Boosting است که از چارچوب گسترده‌ای برای اجتماع درخت‌های تصمیم بهره می‌برد. این الگوریتم به صورت معماری بهینه ساخته شده و ویژگی‌هایی مانند استفاده از توابع هدف (objective functions) خاص و اجتماع تکنیک‌های pruning برای جلوگیری از overfitting دارد. XGBoost با اجتماع مدل‌های ضعیف به مدل قوی‌تری می‌رسد و از مزایایی مانند سرعت بالا و دقت خوب برخوردار است.

ب. Sparsity-aware در XGBoost:

XGBoost به عنوان یک "sparsity-aware" الگوریتم شناخته می‌شود، که به این معناست که برای مدیریت داده‌های پراکنده (sparse data) بهینه شده است. این الگوریتم با ارایه راهکارهایی برای مدیریت داده‌هایی که بسیاری از ویژگی‌ها با مقادیر صفر (sparse) هستند، به بهبود عملکرد مدل کمک می‌کند. از چندین روش برای این منظور استفاده می‌شود:

1. Column Block: از ساختار داده‌ها به صورت ستونی (column-wise) استفاده می‌شود تا مدیریت داده‌های پراکنده بهبود یابد.

2. Sparsity Aware Split Finding: الگوریتم جستجوی بهینه برای تقسیم‌بندی (split) درخت با در نظر گرفتن ساختار داده‌های پراکنده به کار می‌رود.

ج. Missing Values در XGBoost:

XGBoost می‌تواند با مدیریت missing values در داده‌های ورودی به صورت موثری ساخته شده باشد. الگوریتم به صورت خودکار با استفاده از یک مقدار جایگزین برای مقادیر گم‌شده (missing values) کار می‌کند. این موضوع به معنایی است که نیاز به پر کردن missing values به

صورت جداگانه قبل از آموزش مدل وجود ندارد.
Missing value در XGBoost به عنوان یک ویژگی مجزا شناخته نمی‌شود و در فرآیند بهینه‌سازی برخورد به آن به شکلی خاص نمی‌شود. این الگوریتم خود به خوبی با مقادیر گم‌شده کنار می‌آید و قابلیت تعمیم مدل را افزایش می‌دهد.

د. اهداف اصلی XGBoost و AdaBoost:

1. XGBoost:
هدف اصلی: بهبود دقت و کاهش خطا در پیش‌بینی.
مناسب برای: مسائل پیچیده با داده‌های بزرگ و با ابعاد بالا.
رویکرد: از توابع هدف خاص و چارچوب بهینه سازی برای ایجاد یک مدل قوی و مقاوم در برابر overfitting استفاده می‌کند.

2. AdaBoost:
هدف اصلی: افزایش دقت در پیش‌بینی.
مناسب برای: مسائل ساده تا متوسط و با داده‌های معمولی.
رویکرد: با اجتماع مدل‌های ضعیف به مدل قوی می‌رسد و به تمرکز بر داده‌های دشوارتر تمایل دارد.

شباهت‌ها:
هر دو الگوریتم از روش‌های Ensemble Learning استفاده می‌کنند و با ترکیب مدل‌های ضعیف به مدل قوی‌تری می‌رسند.
هدف اصلی هر دو الگوریتم افزایش دقت در پیش‌بینی است.

تفاوت‌ها:

پیچیدگی مسائل:
XGBoost برای مسائل پیچیده با داده‌های بزرگ و با ابعاد بالا مناسب است.
AdaBoost بیشتر برای مسائل ساده تا متوسط با داده‌های معمولی مناسب است.

تکنیک‌های بهینه‌سازی:
XGBoost از چارچوب بهینه سازی و توابع هدف خاص استفاده می‌کند تا یک مدل مقاوم در برابر overfitting ایجاد کند.
AdaBoost از تمرکز بر داده‌های دشوارتر با استفاده از وزن‌دهی به آنها برای افزایش دقت در مدل‌های ضعیف استفاده می‌کند.

ه. مزایای اصلی XGBoost نسبت به AdaBoost:

1. سرعت بالا:
- XGBoost به عنوان یک الگوریتم بهینه شده، عملکرد سریعتری نسبت به AdaBoost دارد.
- استفاده از چارچوب بهینه سازی و تکنیک‌های بهینه‌سازی سبب شده است که XGBoost بر روی مجموعه داده‌های بزرگ با سرعت بالا آموزش ببیند.

2. مقاومیت در برابر overfitting:
- XGBoost با استفاده از pruning و توابع هدف خاص، مدلی مقاوم در برابر overfitting ایجاد می‌کند.
- این ویژگی به XGBoost این امکان را می‌دهد که با داده‌های نویزی و پراکنده مواجه شود.

3. کنترل بر missing values:
- XGBoost به خوبی با missing values در داده‌های ورودی سازگار است و نیاز به پرکردن missing values قبل از آموزش مدل را ندارد.

```
from xgboost import XGBClassifier
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

df = pd.read_csv('heart_disease_uci.csv')
#cat col
categorical_cols = df.select_dtypes(include=['object']).columns.tolist()
#cat col to num
label_encoder = LabelEncoder()
df[categorical_cols] = df[categorical_cols].apply(label_encoder.fit_transform)
df['num'] = df['num'].apply(lambda x: 0 if x == 0 else 1)
X = df.drop('num', axis=1)
y = df['num']
# Normalize
scaler = StandardScaler()
normalized_data = scaler.fit_transform(X)
X = pd.DataFrame(normalized_data, columns=X.columns)
#split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
xgb_model = XGBClassifier()
grid_search = GridSearchCV(xgb_model, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

best_n_estimators = grid_search.best_params_['n_estimators']

k_values = [3, 5, 7]
```

```
for k in k_values:
    xgb_model = XGBClassifier(n_estimators=best_n_estimators)
    xgb_model.fit(X_train, y_train)

    y_pred = xgb_model.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    print(f'\nEvaluation Metrics with k={k}:')
    print(f'Accuracy: {accuracy:.4f}')
    print(f'Precision: {precision:.4f}')
    print(f'Recall: {recall:.4f}')
    print(f'F1-Score: {f1:.4f}')
```

```
Evaluation Metrics with k=3:
Accuracy: 0.8587
Precision: 0.9109
Recall: 0.8440
F1-Score: 0.8762

Evaluation Metrics with k=5:
Accuracy: 0.8587
Precision: 0.9109
Recall: 0.8440
F1-Score: 0.8762

Evaluation Metrics with k=7:
Accuracy: 0.8587
Precision: 0.9109
Recall: 0.8440
F1-Score: 0.8762
```