Stacking in Machine Learning

Stacking is an ensemble learning technique that combines predictions from multiple different machine learning models (often of different types) to generate a more accurate and robust final prediction. Unlike **bagging** and **boosting**, which involve combining several instances of the same base learner, stacking typically combines different types of learners. The core idea behind stacking is to leverage the strengths of various models to improve the overall predictive performance.

How Stacking Works

1. **Base Models (Level 0 models)**: Several different machine learning models are trained on the training data. These could be decision trees, logistic regression, random forests, SVMs, etc.

2. **Meta Model (Level 1 model)**: Once the base models are trained, their predictions are used as input features to train a final model, called the meta-model (or meta-learner). The meta-model is responsible for combining the base models' predictions to produce the final output.

3. **Cross-Validation**: To avoid overfitting, stacking typically uses cross-validation when making predictions for the training data of the meta-learner. This ensures that predictions for training instances are made by models that did not see those instances during their training.

Steps of Stacking:

1. **Train the Base Models**: Train different machine learning models on the training data.
2. **Generate Predictions for Meta Model**: For each base model, generate predictions on the training data using cross-validation and use these predictions to create a new dataset.
3. **Train the Meta Model**: Using the predictions from the base models as features, train the meta-model.
4. **Make Final Predictions**: When making predictions on new data, first generate predictions from the base models and then pass those predictions to the meta-model to produce the final prediction.

Stacking Example in Python

Below is an example using Python's Scikit-learn and StackingClassifier.

```
# Import libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import StackingClassifier
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris

# Load the iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Define base models
base_models = [
    ('svc', SVC(probability=True, kernel='linear')),
    ('rf', RandomForestClassifier(n_estimators=10,
random_state=42))
]

# Define meta model
meta_model = LogisticRegression()

# Create a StackingClassifier with base models and meta model
stacking_clf = StackingClassifier(estimators=base_models,
final_estimator=meta_model)

# Train the stacking model
stacking_clf.fit(X_train, y_train)

# Make predictions
y_pred = stacking_clf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Stacking Classifier Accuracy: {accuracy:.2f}")
```

Explanation:

1. **Base Models (Level 0 Models)**:
   - `SVC`: A Support Vector Classifier
   - `RandomForestClassifier`: A Random Forest model
2. **Meta Model (Level 1 Model)**:
   - `LogisticRegression`: Logistic regression is used as the meta-learner, taking the predictions of the base models as input.

How It Works:

- **Training Phase**: The base models (`SVC` and `RandomForestClassifier`) are trained on the training data. The predictions from these models are then used as features to train the meta-model (`LogisticRegression`).

- **Prediction Phase**: When predicting new data, the base models generate their predictions, and those predictions are passed to the meta-model, which outputs the final prediction.

Practical Explanation:

1. **Diversity of Models**: Stacking allows you to combine models that capture different types of relationships in the data. For example, one model might be good at identifying linear relationships, while another model might excel at detecting non-linear patterns. The meta-model learns to combine these diverse outputs into a single, superior prediction.

2. **Cross-Validation in Training**: Cross-validation is typically applied during the training of base models to avoid overfitting. This ensures that each base model's predictions for the training data are "honest" (i.e., not influenced by having already seen that data point).

3. **Meta-Model's Role**: The meta-model learns how to weigh and combine the predictions of the base models to make the final decision. It acts as a judge that decides which base model to trust more for specific instances.

Example for Regression Problem (StackingRegressor)

Stacking can also be applied to regression tasks, using `StackingRegressor`.

```
from sklearn.ensemble import StackingRegressor
from sklearn.linear_model import RidgeCV
from sklearn.svm import SVR
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error

# Load dataset
data = fetch_california_housing()
X, y = data.data, data.target

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

# Base models
estimators = [
    ('svr', SVR(kernel='linear')),
    ('ridge', RidgeCV())
]

# Meta model
stack_regressor = StackingRegressor(
    estimators=estimators,
    final_estimator=RidgeCV()
)

# Train and predict
```

```
stack_regressor.fit(X_train, y_train)
y_pred = stack_regressor.predict(X_test)

# Evaluate
mse = mean_squared_error(y_test, y_pred)
print(f"Stacking Regressor MSE: {mse:.2f}")
```

Pros and Cons of Stacking

Pros

- **Better Performance**: Stacking can significantly improve prediction accuracy by leveraging the strengths of multiple models.
- **Diversity**: Different base models can capture different patterns in the data, improving the overall robustness of the model.
- **Flexibility**: You can use any type of base model, including neural networks, tree-based models, or linear models, making stacking highly flexible.

Cons

- **Complexity**: Stacking increases the complexity of the model both in terms of training time and interpretability. Debugging issues and tuning the model can become more challenging.
- **Overfitting Risk**: Although stacking tries to reduce overfitting by using cross-validation, if not handled properly, the meta-model can overfit to the predictions of the base models.
- **Training Time**: Training multiple models and then a meta-model adds to the computational expense.

Conclusion

Stacking is a powerful ensemble learning technique that combines different types of models to improve performance. It's a versatile method that can be applied to both classification and regression problems. Its ability to blend multiple diverse models makes it particularly effective when the data has complex patterns that no single model can capture adequately on its own. However, its increased complexity should be handled carefully to avoid overfitting and excessive computational costs.

Import all the required framework

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
```

Load the entire dataset

In [2]:
```
data = pd.read_csv('./diabetes.csv')
```

```
data.head()
```

Out[2]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPed
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	

Split the data into the input and target features

In [3]:
```
## Split the data into input features and the target outcome
X = data.drop(columns=['Outcome'], axis = 1)
y = data['Outcome']
```

Split the entire data into two subsets(50:50):

- Subset 1 (Train Set)
- Subset 2 (Validation Set/Hold Out Set)

In [4]:
```
from sklearn.model_selection import train_test_split
train, val_train, test, val_test = train_test_split(X, y, test_size = 0.5
```

Split the train set further into train and test set (80:20)

In [5]:
```
X_train,X_test,y_train,y_test =  train_test_split(train,test,test_size=0.
```

Model 1: Implementation of KNeighboursClassifier

In [6]:
```
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
knn.score(X_test, y_test)
```

Out[6]:
```
0.7402597402597403
```

Model 2: Implementation of SVC Classifier

In [7]:
```
svc = SVC()
svc.fit(X_train, y_train)
svc.score(X_test, y_test)
```

Out[7]:
```
0.7402597402597403
```

Predictions of the model in the validation set

In [8]:
```
predict_val1 = knn.predict(val_train)
predict_val2 = svc.predict(val_train)
```

In [9]:
```
predict_val1
```

Out[9]:
```
array([0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1,
       1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
       0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1,
       1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1,
       0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
       1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1,
       0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0,
       0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1,
       0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0,
       0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1,
       0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1,
       0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
       0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1,
       0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1,
       0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 0, 1])
```

In [10]:
```
predict_val2
```

Out[10]:
```
array([0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1,
       0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
       0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1,
       0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
       0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0,
       1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1,
       0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0,
       0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
       0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1,
       0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0,
       0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0,
       1, 0, 0, 0, 1, 1, 1, 0, 0, 0])
```

In [ ]:
```
predict_val = np.column_stack((predict_val1, predict_val2))
predict_val
```

In [12]:
```
predict_test1 = knn.predict(X_test)
predict_test2 = svc.predict(X_test)
predict_test = np.column_stack((predict_test1, predict_test2))
```

Stacking: Input features are the predictions of the model M1(KNN) and model M2(SVC) stored inside the predict_val(input features) for the RandomForestClassifier

In [13]:
```
from sklearn.ensemble import RandomForestClassifier
rand_clf = RandomForestClassifier()
rand_clf.fit(predict_val, val_test)
```

Out[13]:
```
RandomForestClassifier
RandomForestClassifier()
```

In [14]:
```
rand_clf.score(predict_test, y_test)
```

Out[14]:
```
0.7402597402597403
```

Hyperparameter tuning: GridSearchCV

In [15]:
```
grid_param = {
    "n_estimators" : [90,100,115],
    'criterion': ['gini', 'entropy'],
    'min_samples_leaf' : [1,2,3,4,5],
    'min_samples_split': [4,5,6,7,8],
    'max_features' : ['auto','log2']
}
```

In [16]:
```
from sklearn.model_selection import GridSearchCV

grid_search = GridSearchCV(estimator=rand_clf,param_grid=grid_param,cv=5,
```

In [ ]:
```
grid_search.fit(predict_val, val_test)
```

In [18]:
```
grid_search.best_params_
```

Out[18]:
```
{'criterion': 'gini',
 'max_features': 'log2',
 'min_samples_leaf': 1,
 'min_samples_split': 4,
 'n_estimators': 90}
```

In [19]:
```
rand_clf_hyperparameter_tuning = RandomForestClassifier(n_estimators=90,
rand_clf_hyperparameter_tuning.fit(predict_val, val_test)
```

Out[19]:
```
RandomForestClassifier
RandomForestClassifier(min_samples_split=4, n_estimators=90)
```

In [20]:
```
rand_clf_hyperparameter_tuning.score(predict_test, y_test)
```

Out[20]:
```
0.7402597402597403
```