Natural Language and Information Retrieval
Degree in Data Science
ETSINF. Academic year 2022-2023

Practical Work 3

Session 3: Ensemble Methods

Reynier Ortega Bueno
rortega@prhlt.upv.es

Goal

The goal of this lab session on ensemble methods is to help students understand the benefits and drawbacks of combining multiple machine learning models. Students should be able to understand the different types of ensemble methods, including voting, stacking, bagging, boosting, and random forests. Implement ensemble methods using popular machine learning libraries like scikit-learn in Python. Evaluate the performance of ensemble methods using standard machine learning metrics like accuracy, precision, and recall. Compare the performance of different ensemble methods and identify the strengths and weaknesses of each of them. Through hands-on exercises, students will gain practical experience with implementing and evaluating ensemble methods, and will develop a deeper understanding of how they can be used to improve the accuracy and reliability of machine learning models.

Bibliography

- Rokach, L. (2019). *Ensemble learning: pattern classification using ensemble methods*. World Scientific Publishing Company Incorporated.
- Zhang, C., & Ma, Y. (Eds.). (2012). *Ensemble machine learning: methods and applications*. Springer Science & Business Media.
- Zhou, Z. H. (2012). *Ensemble methods: foundations and algorithms*. CRC press.
- *Seni, G., & Elder, J. F. (2010). Ensemble methods in data mining: improving accuracy through combining predictions. Synthesis lectures on data mining and knowledge discovery, 2(1), 1-126.*

ENSEMBLE OF MODELS

Ensemble methods can often outperform simple classifiers due to the concept of the *"wisdom of the crowd"* or *"collective intelligence".* By combining the predictions of multiple models ensemble methods can mitigate the individual weaknesses of each model and produce more accurate and robust predictions. One theoretical foundation for why ensemble methods work well is the *"bias-variance tradeoff"*. In machine learning, the goal is to build a model that can accurately generalize to new data points that have not been seen before. The *“bias-variance tradeoff”* refers to the tradeoff between a model's ability to fit the training data (low bias) and generalize on new data (low variance).

Simple models such as decision trees tend to have high bias and low variance, meaning they are likely to underfit the training data but have low variance in their predictions. On the other hand, complex models such as neural networks tend to have low bias but high variance, meaning that they are likely to overfit the training data and have high variance in their predictions. Ensemble methods can help to balance the “bias-variance tradeoff” by combining multiple models with different biases and variances. For example, bagging and random forest methods use multiple models to create an ensemble with lower variance and better generalization performance. Boosting techniques such as AdaBoost and Gradient Boosting can improve bias and variance by iteratively adding new models that focus on correcting the errors of previous models.

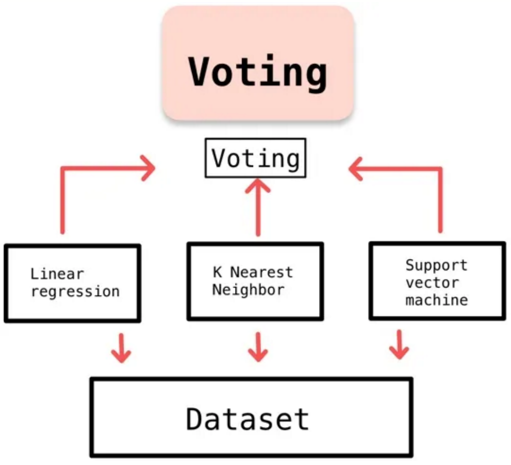
Approaches to combine models

The most common approaches for supervised ensemble models in machine learning are:

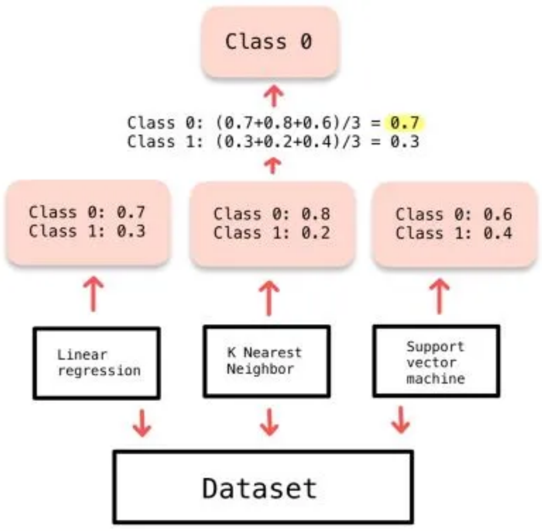
1. **Voting**: In this approach, multiple base models are trained on the same dataset, and the final prediction is made by taking a majority vote of the predictions of the single models. This approach is typically used when the base models are diverse and complementary.
2. **Bagging**: In this approach, multiple instances of the same base model are trained on different subsets of the training data. The final prediction is made by averaging the predictions of the individual models. Bagging is typically used when the base model has high variance or is unstable.
3. **Boosting**: In this approach, multiple base models are trained sequentially, with each subsequent model learning from the errors of the previous model. The final prediction is made by taking a weighted average of the predictions of the single models. Boosting is typically used when the base model is weak, aiming to improve accuracy and reduce bias.
4. **Stacking**: In this approach, multiple base models are trained on the same dataset, and the predictions of the individual models are used as input features for a final model. The final model learns to combine the predictions of the single models to improve accuracy. Stacking is typically used when the base models are diverse and complementary.

VOTING
The idea is that by combining the predictions of multiple models that are trained on the same data but using different algorithms or parameter settings, the resulting prediction will be more accurate and robust than any single model's prediction.

There are two main types of voting ensembles: *"hard voting"* and *"soft voting"*. In hard voting, the prediction of the ensemble is determined by a simple majority vote of the predictions of the base models. For example, if three base models predict that an input belongs to the class *humor*, and two predict that it belongs to the class *prejudiced humor*, then the ensemble would predict the class *humor*. In soft voting, the ensemble prediction is based on the average probabilities of the base models' predictions for each class. The class with the highest average probability is then selected as the final prediction.



Voting ensemble: hard voting



Voting ensemble: soft voting

Voting ensembles can be applied to a wide range of machine learning tasks, including classification, regression, etc. They can be particularly useful in situations where the performance of a single model is limited, or where different models have different strengths and weaknesses. By combining the predictions of multiple models, voting ensembles can provide a more accurate and robust prediction that is less likely to be affected by individual model biases or errors.

EXAMPLE 1

```
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
#Here you can define the training and test sets for the HUHU
subtask (1, 2a)
X_train, X_test, y_train, y_test= None, None, None, None
# create the base models, Logistic Regression, Decision Tree,
and Support Vector Machine
model1 = LogisticRegression()
model2 = DecisionTreeClassifier()
model3 = SVC()
# create the voting ensemble
ensemble = VotingClassifier(estimators=[('lr', model1), ('dt',
model2), ('svm', model3)], voting='hard')
# train the ensemble
ensemble.fit(X_train, y_train)
# Evaluate the ensemble
ensemble.score(X_test, y_test)
```

The voting parameter is set to *'hard'* to indicate that we want to use hard voting.

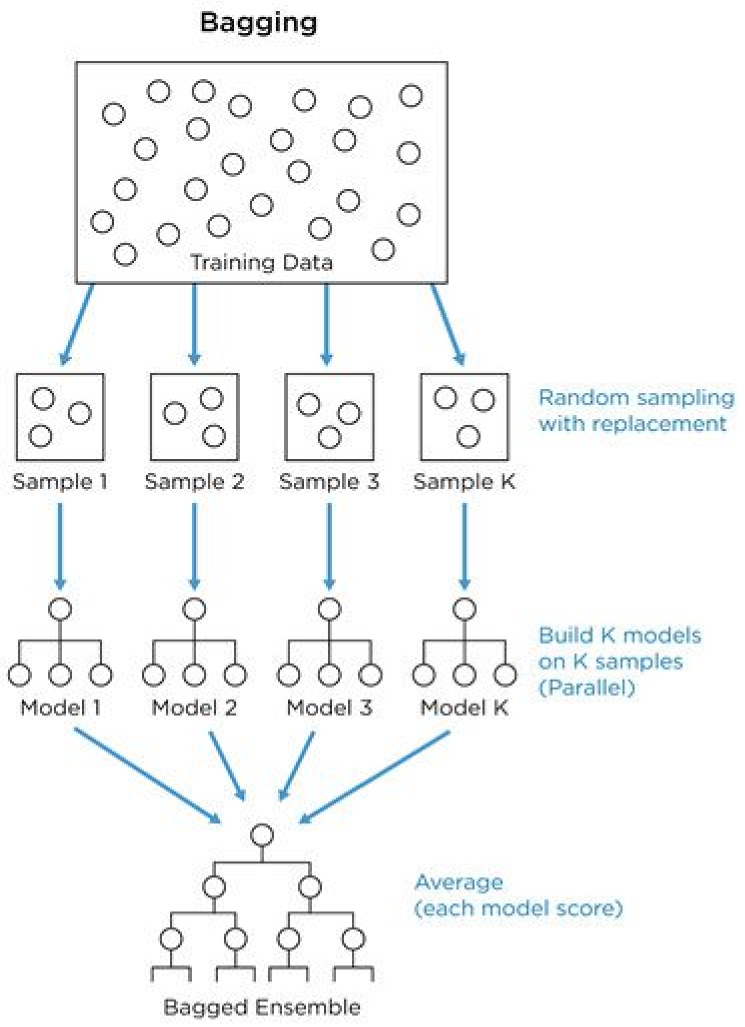
EXAMPLE 2

```
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
#Here you can define the training and test sets for the HUHU
subtask (1, 2a)
X_train, X_test, y_train, y_test= None, None, None, None
# create the base models, Logistic Regression, Decision Tree,
and Support Vector Machine
# create the base models
model1 = LogisticRegression()
model2 = DecisionTreeClassifier()
model3 = SVC(probability=True)
# create the voting ensemble
ensemble = VotingClassifier(estimators=[('lr', model1), ('dt',
model2), ('svm', model3)], voting='soft')
# train the ensemble
ensemble.fit(X_train, y_train)
# evaluate the ensemble
ensemble.score(X_test, y_test)
```

This example created the same three base models but set the *probability* parameter of the Support Vector Machine model to True. This enables the model to **output class probabilities required for soft voting**. We then create a soft voting ensemble by setting the voting parameter to *'soft'*.

BAGGING

Bagging is based on bootstrap resampling, which involves randomly sampling the training data with replacement to create multiple subsets of the same size as the original data set. The basic idea behind bagging is to train a set of base models on different bootstrap samples of the training data and then aggregate their predictions using a simple averaging or voting scheme to create the final prediction. This can help reduce the model's variance and improve its generalization performance.


Bagging
Training Data
Sample 1
Sample 2
Sample 3
Sample K
Random sampling with replacement
Model 1
Model 2
Model 3
Model K
Build K models on K samples (Parallel)
Average (each model score)
Bagged Ensemble

General schema of the bagging ensemble

The bagging algorithm can be summarized as follows:

1. Create multiple bootstrap samples of the training data, each of the same size as the original data set.
2. Train a base model on each bootstrap sample.

3. Aggregate the predictions of the base models using a simple averaging or voting scheme to create the final prediction.

Random Forest is an extension over bagging. In this approach, multiple decision trees are trained on different subsets of the training data. It also takes the random selection of features rather than using all features to grow trees. The final prediction is made by averaging the predictions of the individual trees. Random forests are typically used when the dataset has high dimensionality and complex relations among the features.

EXAMPLE 3

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
#Here you can define the training and test sets for the HUHU
subtask (1, 2a)
X_train, X_test, y_train, y_test= None, None, None, None
ensemble
=BaggingClassifier(base_estimator=DecisionTreeClassifier(),
n_estimators=10)
# train the ensemble
ensemble.fit(X_train, y_train)
# evaluate the ensemble
ensemble.score(X_test, y_test)
```

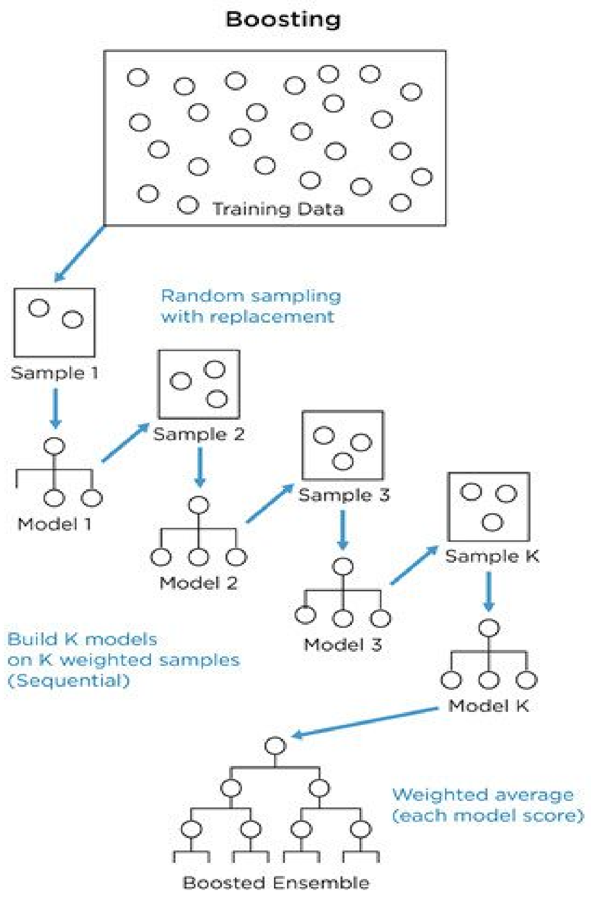
EXAMPLE 4

```
from sklearn.ensemble import RandomForestClassifier
#Here you can define the training and test sets for the HUHU
subtask (1, 2a)
X_train, X_test, y_train, y_test= None, None, None, None
ensemble           =RandomForestClassifier(n_estimatorsint=10,
max_depth=2, random_state=0)
# train the ensemble
ensemble.fit(X_train, y_train)
```

```
# evaluate the ensemble
ensemble.score(X_test, y_test)
```

BOOSTING

Boosting is an ensemble learning method combining multiple weak learners to create a strong learner. Conversely to bagging, which trains each weak learner independently, boosting trains weak learners sequentially, with each subsequent learner concentrating on the data points the previous learners misclassified. For that, the boosting method reweights the training data iteratively so that the next model focuses more on the data points that the previous model misclassified. AdaBoost and XGBoost are two boosting approaches.


Boosting
Training Data
Random sampling with replacement
Sample 1
Model 1
Sample 2
Model 2
Sample 3
Model 3
Sample K
Model K
Build K models on K weighted samples (Sequential)
Weighted average (each model score)
Boosted Ensemble

General schema of the boosting ensemble

AdaBoost (Adaptive Boosting) is a popular boosting algorithm[1] introduced by Freund and Schapire in 1997. The AdaBoost algorithm works by iteratively adding weak learners to strong learners. At each iteration, the algorithm selects a weak learner that can best classify the examples that the previous weak learners misclassified. The final model is a weighted combination of the weak learners, where the weights are determined by the accuracy of each weak learner on the training data.

The mathematical foundation of AdaBoost can be described as follows. Suppose we have a dataset with input features *X* and corresponding target values *y*, where y is a binary classification label (+1 or -1). We also have a set of base models, denoted by *M={Model1, Model2, ..., Modelk}*, which can be used to make binary predictions on X. AdaBoost aims to train a weighted combination of the base models that can make accurate binary predictions on *X*.

EXAMPLE 5

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
#Here you can define the training and test sets for the HUHU
subtask (1, 2a)
X_train, X_test, y_train, y_test= None, None, None, None
# Create the boosting classifier
ensemble=AdaBoostClassifier(base_estimator=LogisticRegression(
),n_estimators=10)
# Train the boosting classifier on the training data
ensemble.fit(X_train, y_train)
# Evaluate the boosting classifier on the testing data
score = stacking.score(X_test, y_test)
print("Accuracy:", score)
```

[1] Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, *55*(1), 119-139.

XGBoost (eXtreme Gradient Boosting)[2] is a robust ensemble learning algorithm introduced by Chen and Guestrin in 2016. It extends the gradient boosting method, focusing on scalability and speed. XGBoost is widely used in machine learning competitions and is known for its high accuracy and performance. The XGBoost method minimizes a differentiable loss function, such as mean squared error or log loss, using gradient descent. XGBoost iteratively adds decision trees to a weighted combination of previous trees, where the weights are determined by the negative gradient of the loss function.

EXAMPLE 6

```
!pip install xgboost
import xgboost as xgb
from sklearn.metrics import  accuracy_score
#Here you can define the training and test sets for the HUHU
subtask (1, 2a)
X_train, X_test, y_train, y_test= None, None, None, None
# Convert data to DMatrix format
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Set XGBoost parameters
params = {
   'max_depth': 3,
   'eta': 0.1,
   'objective': 'binary:logistic',
   'eval_metric': 'auc'
}
```

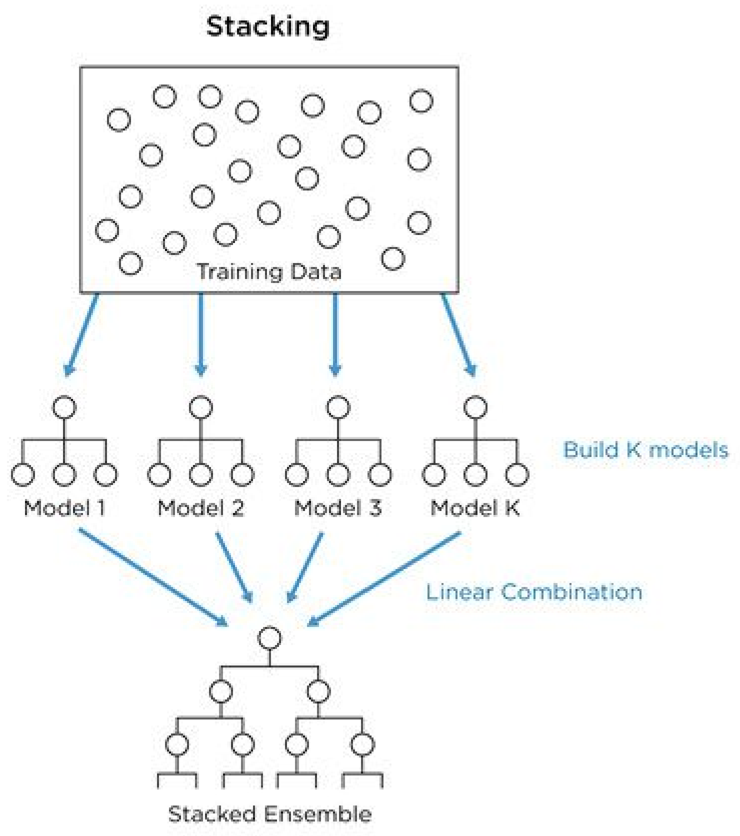
[2] Chen, T., & Guestrin, C. (2016, August). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (pp. 785-794).

```
# Train the model
num_rounds = 100
bst = xgb.train(params, dtrain, num_rounds)
# Make predictions on the test set
y_pred = bst.predict(dtest)
# Evaluate the model
acc = accuracy_score(y_test, y_pred)
print(f"Acc: {acc}")
```

STACKING

Stacking, or stacked generalization, is an ensemble learning technique combining multiple base models to improve prediction performance. The basic idea behind stacking is to train a meta-model on the predictions of several base models, hoping that the meta-model can learn to combine the strengths of the base models and mitigate their weaknesses.

The mathematical foundations of stacking can be described as follows. Suppose we have a dataset with input features *X* and corresponding labels values *y*. We have a set of base models, denoted by *M={Model1, Model2, ..., Modelk}*, which can be used to make predictions on *X*. Stacking aims to train a **meta-model**, denoted by *M'*, that can make more accurate predictions on *X* than each individual base model in *M*.



General schema of the stacking ensemble

Firstly, we first split the data into training and testing sets to do this. We then use the training set to train the base models in *M* and use them to make predictions on the testing set. We stack these predictions together to form a new feature matrix, denoted by *Z*, where each row corresponds to a sample in the testing set, and each column corresponds to the predicted value of the test set by one of the base models in *M*. Then, we use *Z* as the new input features to train *M'*. The target values for *M'* are the true label values of the testing set. *M'* can be any model type, such as linear regression, logistic regression, or neural network. Once *M'* has been trained, we can predict new data points.

Stacking is a robust technique for improving prediction performance in machine learning, but it should be used wisely and carefully considering the computational and interpretational costs.

EXAMPLE 7

```
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split


#Here you can define the training and test sets for the HUHU
subtask (1, 2a)
X_train, X_test, y_train, y_test= None, None, None, None
#  create  the  base  models,K-NNeighbors,  Logistic  Regression,
Decision Tree, and Support Vector Machine
base_models  =  [('dt',  DecisionTreeClassifier()),  ('svm',
SVC()), ('knn', KNeighborsClassifier())]
# Create the meta-model
meta_model = LogisticRegression()
# Create the stacking classifier
ensemble      =      StackingClassifier(estimators=base_models,
final_estimator=meta_model)
# Train the stacking classifier on the training data
ensemble.fit(X_train, y_train)
# Evaluate the stacking classifier on the testing data
score = ensemble.score(X_test, y_test)
print("Accuracy:", score)
```

COMBINING REGRESSION MODELS

The ensemble methods can be applied to the regression problems. It refers to a combination of several regression models to improve the accuracy and robustness of predictions. The idea behind ensemble methods is that the collective model can perform better by combining multiple regression models than any single model. The previous schemas, Bagging, Boosting and Stacking, can be used to build an ensemble. It is essential to **highlight that the base models must be regression models instead of classification models.** Several regression methods are implemented in the scikit-learn python library[3], e.g.,: LinearRegression, Ridge, Lasso, SVR, KNeighborsRegressor DecisionTreeRegressor and MLPRegressor. Some examples of how to build the ensemble of regression models are shown below.

Ensemble using the bagging schema

EXAMPLE 8

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from     sklearn.ensemble     import     RandomForestRegressor,
VotingRegressor
from sklearn.metrics import mean_squared_error
from math import sqrt

#Here you can define the training and test sets for the HUHU
subtask (1, 2a)
X_train, X_test, y_train, y_test= None, None, None, None
# Define the base regression models.
#You can used different regression models
lr = LinearRegression()
dt = DecisionTreeRegressor(max_depth=3)
rf = RandomForestRegressor(n_estimators=100)
```

[3] https://scikit-learn.org/stable/supervised_learning.html#supervised-learning

```
# Define the voting model
ensemble = VotingRegressor(estimators=[('lr', lr), ('dt', dt),
('rf', rf)])
# Fit the voting model to the training data
ensemble.fit(X_train, y_train)
predict = ensemble.predict(X_test)
rmse= mean_squared_error(y_test, predict)
print(f"Root mean squared error RMSE {sqrt(rmse):.2f}")
# Evaluate the model on the test data
score = ensemble.score(X_test, y_test)
print(f"R2 score: {score:.2f}")
```

Ensemble using the AdaBoost schema

EXAMPLE 9

```
from sklearn.ensemble import AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor
# Create a decision tree regressor
dt = DecisionTreeRegressor(max_depth=3)
# Create an AdaBoost regressor with 100 trees
ensemble          =         AdaBoostRegressor(base_estimator=dt,
n_estimators=100)

# Fit the model to the training data
ensemble.fit(X_train, y_train)
# Make predictions on the test data
predict = ensemble.predict(X_test)
rmse= mean_squared_error(y_test, predict)
print(f"Root mean squared error RMSE {sqrt(rmse):.2f}")
```

Ensemble using the stacking schema

EXAMPLE 10

```
from     sklearn.model_selection     import     cross_val_score,
train_test_split
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.svm import SVR
from     sklearn.ensemble     import     RandomForestRegressor,
GradientBoostingRegressor
from mlxtend.regressor import StackingCVRegressor

# Define the base models
lr = LinearRegression()
ridge = Ridge()
rf = RandomForestRegressor(n_estimators=100)
gb = GradientBoostingRegressor(n_estimators=100)
svr= SVR()
# Define the stacking model
ensemble = StackingCVRegressor(regressors=(lr, svr, rf, gb),
                           meta_regressor=ridge,
                           use_features_in_secondary=True)

# Fit the stacking model to the training data
ensemble.fit(X_train, y_train)
predict = ensemble.predict(X_test)
rmse= mean_squared_error(y_test, predict)
print(f"Root mean squared error RMSE {sqrt(rmse):.2f}")
```

ACTIVITY

Implement and evaluate the combination of classification models in the HUHU tasks (at least two different techniques) for each subtask (1 and 2a). Also implement a combination of regression models to address the subtask 2b. Deliver code and report with explanation of the strategy used before the next session.

The Google collaborative environment (Google Colab) can be used to implement the machine learning models:
https://colab.research.google.com/notebooks/welcome.ipynb#scrollTo=5fCEDCU_qrC0