Data Mining 2

Topic 05 : Ensemble Learning

Lecture 01 : Introduction to Ensemble Learning

Dr Kieran Murphy

Department of Computing and Mathematics, Waterford Institute of Technology.
(Kieran.Murphy@setu.ie)

Spring Semester, 2025

Outline

- Ensemble learners
- Bootstrapping and Bagging
- Boosting, AdBoost

Outline

Motivation

Condorcet's jury theorem is a political science theorem about the relative probability of a given group of individuals arriving at a correct decision:

Theorem 1 (Condorcet's jury theorem)

Assume a group of n independent voters wishes to reach a decision by majority vote. One of the two outcomes of the vote is correct, and each voter has an independent probability, p, of voting for the correct decision.

- *If $p > 0.5$, then adding more voters increases the probability that the majority decision is correct. In the limit, the probability that the majority votes correctly approaches 1 as the number of voters, n, increases.*
- *If $p < 0.5$ then adding more voters makes things worse: the optimal jury consists of a single voter.*

Take Home Message

- Even weak decision makers have benefit as long as they individualy perform better than chance ($p > 0.5$).
- "Wisdom of Crowds" needs independence!
- What happens if $p < 0.5$?

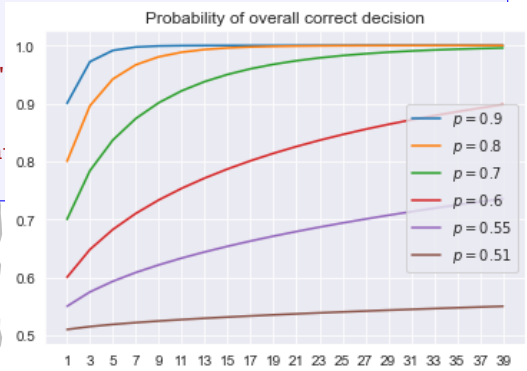
Condorcet's Jury Theorem

```
from scipy.stats import binom
nValues = np.array(range(1,41,2))

prob_correct = lambda n,p: 1-binom.cdf(n//2, n, p)

for p in [0.9, 0.8, 0.7, 0.6, 0.55, 0.51]:
    muValues = prob_correct(nValues,p)
    plt.plot(nValues,muValues,label="$p=%s$" % p)

plt.title("Probability of overall correct decision")
plt.xticks(nValues)
plt.legend(loc="center right")
plt.savefig("jury_decision.pdf",bbox_inches="tight")
plt.show()
```

... just calculating cumulative binomial distribution probabilities.

Condorcet's Jury Theorem

```
from scipy.stats import binom
nValues = np.array(range(1,41,2))

prob_correct = lambda n,p: 1-binom.cdf(n//2, n, p)

for p in [0.9, 0.8, 0.7, 0.6, 0.55, 0.51]:
    muValues = prob_correct(nValues,p)
    plt.plot(nValues,muValues,label="$p=%s$" % p)

plt.title("Probability of overall correct decision"
plt.xticks(nValues)
plt.legend(loc="center right")
plt.savefig("jury_decision.pdf",bbox_inches="tigh
plt.show()
```



... just calculate cumulative binomial distribution probabilities.

With **enough** voters the probability of correct decision approaches one..

For p large, the probability converges quickly to one..

'enough' becomes large for $p \approx 0.5$.

Practical Motivation — Netflix Prize*

- Open competition to predict user ratings for films, based only on previous ratings.
- Prize was awarded on 2009 to BellKor's Pragmatic Chaos team which bested Netflix's own algorithm for predicting ratings by 10.06%. was an ensemble of 107 modules.

Leaderboard

Display top 20 leaders.

Rank	Team Name	Best Score	% Improvement	Last Submit Time
1	The Ensemble	0.8553	10.10	2009-07-26 18:38:22
2	BellKor's Pragmatic Chaos	0.8554	10.09	2009-07-26 18:18:28
Grand Prize - RMSE <= 0.8563				
3	Grand Prize Team	0.8571	9.91	2009-07-24 13:07:49
4	Opera Solutions and Vandelay United	0.8573	9.89	2009-07-25 20:05:52
5	Vandelay Industries !	0.8579	9.83	2009-07-26 02:49:53
6	PragmaticTheory	0.8582	9.80	2009-07-12 15:09:53
7	BellKor in BigChaos	0.8590	9.71	2009-07-26 12:57:25
8	Dace	0.8603	9.58	2009-07-24 17:18:43
9	Opera Solutions	0.8611	9.49	2009-07-26 18:02:08
10	BellKor	0.8612	9.48	2009-07-26 17:19:11
11	BigChaos	0.8613	9.47	2009-06-23 23:06:52
12	Feeds2	0.8613	9.47	2009-07-24 20:06:46
Progress Prize 2008 - RMSE = 0.8616 - Winning Team: BellKor in BigChaos				
13	xiangliang	0.8633	9.26	2009-07-21 02:04:40
14	Gravity	0.8634	9.25	2009-07-26 15:58:34
15	Ces	0.8642	9.17	2009-07-25 17:42:38
16	Invisible Ideas	0.8644	9.14	2009-07-20 03:26:12
17	Just a guy in a garage	0.8650	9.08	2009-07-22 14:10:42
18	Craig Carmichael	0.8656	9.02	2009-07-25 16:00:54
19	J Dennis Su	0.8658	9.00	2009-03-11 09:41:54
20	acmehill	0.8659	8.99	2009-04-16 06:29:35
Progress Prize 2007 - RMSE = 0.8712 - Winning Team: KorBell				
Cinematch score on quiz subset - RMSE = 0.9514				

Ensembe Methods

Definition 2 (Ensemble Learner)

An ensemble learner is a set of models whose individual decisions are combined in some way to classify new examples.

- Simplest approach:
  - Generate/Train multiple classifiers
  - Each votes on test instance
  - Take majority as classification
- Classifiers are different due to different sampling of training data, or randomised parameters within the classification algorithm.
- Aim: take simple mediocre algorithm and transform it into a super classifier without requiring any fancy new algorithm.
- Differences in training strategy, and in combination method:
  - Parallel training with different training sets: Bagging or Cross-validated committees
  - Sequential training, iteratively re-weighting training examples so current classifier focuses on hard examples: boosting
  - Parallel training with objective encouraging division of labor: mixture of experts

Why do Ensemble Methods Work?

Variance reduction

If the training sets are completely independent, it will always help to average an ensemble because this will reduce variance without affecting bias (e.g., bagging)

- Reduce sensitivity to individual data points.

Bias reduction

For simple models, average of models has much greater capacity than single model (e.g., hyperplane classifiers, Gaussian densities).

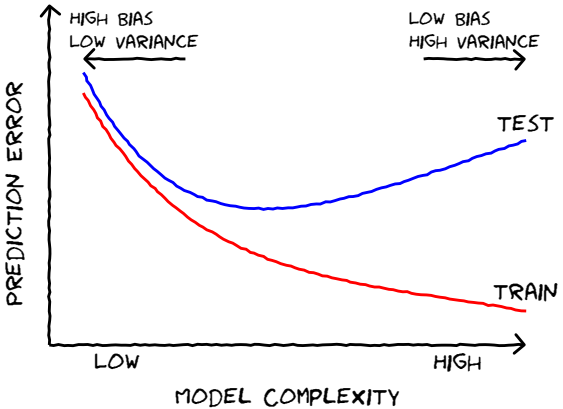
- Averaging models can reduce bias substantially by increasing capacity, and control variance by fitting one component at a time (e.g., boosting)

Classification vs Regression

- Regression models can be averaged.
- Classification models can be averaged if output is probability of being in a class, otherwise can use majority vote if classifier only outputs class.

Bias/Variance Tradeoff

Bias: error from insufficient complexity in the model



Variance: error from sensitivity to small fluctuations in the training set

- On the left — Model is too simple, has large bias (as model is too simple to learn signal) but small variance (as model is too simple to learn/be affected by noise).
- On the right — Model is too complicated, has small bias (as model can learn signal) but has large variance (as model also can learn noise).

Reduce Variance Without Increasing Bias

It seems that all we can do to is select how complicated our model is to minimise the generalisation error ($\text{bias}^2 + \text{Var} + \text{noise}$). But this is not the case:

- It is possible to reduce variance without affecting bias by averaging.

Averaging reduces variance

- Given N independent estimates for X, each with variance of $\text{Var}(X)$, we have

$$\text{Var}(\bar{X}) = \frac{\text{Var}(X)}{N}$$

But only if independent!

One Problem

We have only one training set, so where do multiple models (independent estimates) come from? — apply Bootstrap sampling

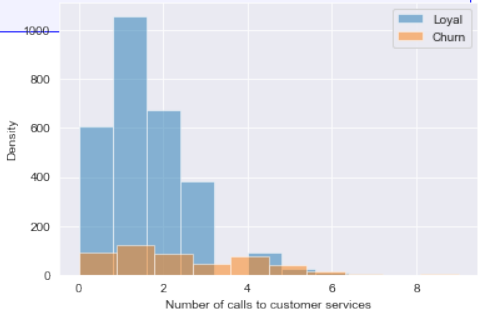
Bootstrap sampling

Given set D containing N training examples, create D' by drawing N examples at random with replacement from D.

Example — Bootstrapping for Small Samples I

Consider our Churn dataset with only 3333 rows. Lets look at the distribution of `Cust_Serv_Calls` for both the loyal customers and the churning customers . . .

```
df.loc[df['Churn']==0,'Cust_Serv_Calls'].hist(label='Loyal',alpha=0.5)
df.loc[df['Churn']==1,'Cust_Serv_Calls'].hist(label='Churn',alpha=0.5)
plt.xlabel('Number of calls to customer services')
plt.ylabel('Density')
plt.legend()
plt.savefig("churn__Cust_Serv_Calls__hist.pdf",bbox="tight")
plt.show()
```

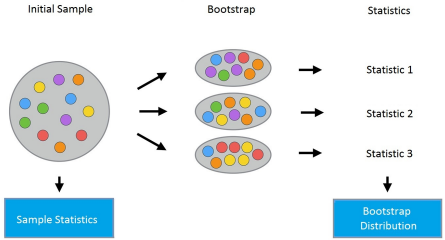


- Looks like loyal customers make fewer calls to customer service than those who eventually leave.
- So we should estimate the average number of customer service calls in each group.
- As dataset is small, we would not get a good estimate by simply calculating the mean of the original samples.
- We would be better off applying a bootstrap method . . .

Example — Bootstrapping for Small Samples II

Create two utility functions to generate the bootstrap samples and to compute statistic estimates:

- We could have used `np.random.choice` with option `replace=True` to get the same effect.
- α%-confidence intervals are computed as done back in semester 4.




```
4
def get_bootstrap_samples(data, n_samples):
    """Generate bootstrap samples using the bootstrap method."""
    indices = np.random.randint(0, len(data), (n_samples, len(data)))
    samples = data[indices]
    return samples

def stat_intervals(stat, alpha):
    """Produce an interval estimate."""
    boundaries = np.percentile(stat, [100*alpha/2, 100*(1-alpha/2)])
    return boundaries
```

Example — Bootstrapping for Small Samples III

Next we split the data set, generate bootstrap samples and resulting confidence intervals ...

```
np.random.seed(42)

loyal_calls = df.loc[df['Churn']==0, 'Cust_Serv_Calls'].values
churn_calls = df.loc[df['Churn']==1, 'Cust_Serv_Calls'].values

# Generate the samples using bootstrapping and calculate the mean
loyal_mean_scores = [np.mean(sample)
    for sample in get_bootstrap_samples(loyal_calls, 1000)]
churn_mean_scores = [np.mean(sample)
    for sample in get_bootstrap_samples(churn_calls, 1000)]

print("Service calls from loyal: mean interval",
    stat_intervals(loyal_mean_scores, 0.05))
print("Service calls from churn: mean interval",
    stat_intervals(churn_mean_scores, 0.05))
```

```
Service calls from loyal: mean interval [1.40700877 1.4922807 ]
Service calls from churn: mean interval [2.06625259 2.38307453]
```

Outline

Outline of Workflow

Aim

Demo the (possible) impact of using ensemble learners — voting classifier, boosting and stacking.

Summary

- Wanted to check impact of different learners across multiple datasets so needed a needed a robust (even if non-optimal) preprocessing pipeline.
  - Uses UCL python module `ucimlrepo`, to select/download dataset.
- Focus on model selection so ...
  - No data cleaning — just dropped rows? with NA.
  - No EDA — what is that?
  - Primitive feature data type specification —- feature is cat $\Leftrightarrow$ `nunique` ≤ 12)
  - Default preprocessing — `StandardScaler` of num, `OneHotEncoder` of cat (condom principle).
- Use loop for model selection — reduce code and more scalable.

Setup

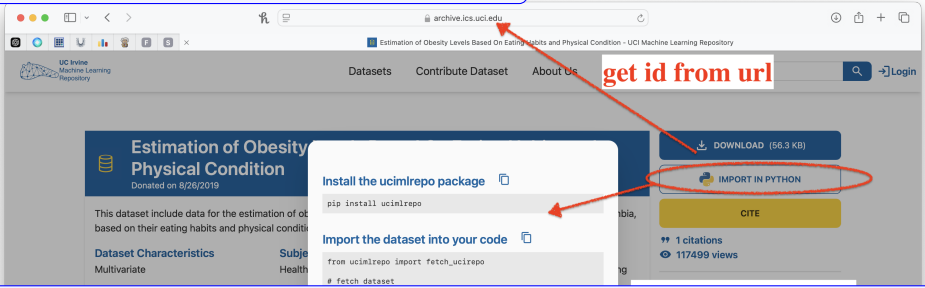
- We start importing the usual modules.
  - `pandas` uses `numpy`, but we often want direct access to `numpy` functions.
  - `seaborn` simplifies styling charts and has a nice selection of statistical charts.
- By default `pandas` limits the number of columns shown. I have never found this to be a god idea.

```
6
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('darkgrid')

import pandas as pd
pd.set_option('display.max_columns', 1000)
pd.set_option('display.width', 1000)

from IPython.display import display, Markdown

SEED = 2025
```

Use `SEED` to reduce variability when generating examples or building models, but always test effect of different seed values.

Dataset

7
```
from ucimlrepo import fetch_ucirepo
dataset = fetch_ucirepo(id=544)
```



8
```
from textwrap import wrap
print("Name:", dataset.metadata.name, sep='\n')
print("Abstract:", "\n".join(wrap(dataset.metadata.abstract,95)), sep='\n')
```

```
Name:
Estimation of Obesity Levels Based On Eating Habits and Physical Condition
Abstract:
This dataset include data for the estimation of obesity levels in individuals from the
countries of Mexico, Peru and Colombia, based on their eating habits and physical condition.
```

UCL DotDict

UCL fetch_ucirepo returns a dotdict (a dictionary structure that supports property type access (dot notation):

9
```
type(dataset)
```
```
ucimlrepo.dotdict.dotdict
```

Top level keys provide access to the data, meta information, and variable properties.

10
```
dataset.keys()
```
```
dict_keys(['data', 'metadata', 'variables'])
```

A quick scan of next level keys shows structure . . .

11
```
dataset.data.keys()
```
```
dict_keys(['ids', 'features', 'targets', 'original', 'headers'])
```

12
```
dataset.metadata.keys()
```
```
dict_keys(['uci_id', 'name', 'repository_url', 'data_url', 'abstract', 'area', 'tasks', 'characteristics',
```

13
```
dataset.variables.keys()
```
```
Index(['name', 'role', 'type', 'demographic', 'description', 'units', 'missing_values'], dtype='object')
```

Convert to Dataframe

I want the data in a `pandas.DataFrame` with classification target in column `Target`.

- Some UCL datasets have multiple target columns — since they can be a mixture of classification and regression problems — so I wanted a simple test to catch unsuitable datasets.

14

```
df = dataset.data.original

targets = dataset.metadata.target_col
assert len(targets)==1, f"Want a dataset with one target column. Got {targets}"

df.rename(columns={targets[0]:'Target'}, inplace=True)

print(df.shape)
df.head()
```

```
(2111, 17)
```

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SMOKE	CH2O	SCC	FAF	TUE	CALC	MTRAN
0	Female	21.0	1.62	64.0	yes	no	2.0	3.0	Sometimes	no	2.0	no	0.0	1.0	no	Public_Transportatio
1	Female	21.0	1.52	56.0	yes	no	3.0	3.0	Sometimes	yes	3.0	yes	3.0	0.0	Sometimes	Public_Transportatio
2	Male	23.0	1.80	77.0	yes	no	2.0	3.0	Sometimes	no	2.0	no	2.0	1.0	Frequently	Public_Transportatio
3	Male	27.0	1.80	87.0	no	no	3.0	3.0	Sometimes	no	2.0	no	2.0	0.0	Frequently	Walking
4	Male	22.0	1.78	89.8	no	no	2.0	1.0	Sometimes	no	2.0	no	0.0	0.0	Sometimes	Public_Transportatio

EDA in 5 seconds — `info`, `describe`, and `countplot`

15
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2111 entries, 0 to 2110
Data columns (total 17 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Gender                          2111 non-null   object
 1   Age                             2111 non-null   float64
 2   Height                          2111 non-null   float64
 3   Weight                          2111 non-null   float64
 4   family_history_with_overweight  2111 non-null   object
 5   FAVC                            2111 non-null   object
 6   FCVC                            2111 non-null   float64
 7   NCP                             2111 non-null   float64
 8   CAEC                            2111 non-null   object
 9   SMOKE                           2111 non-null   object
 10  CH2O                            2111 non-null   float64
 11  SCC                             2111 non-null   object
 12  FAF                             2111 non-null   float64
 13  TUE                             2111 non-null   float64
 14  CALC                            2111 non-null   object
 15  MTRANS                          2111 non-null   object
 16  Target                          2111 non-null   object
dtypes: float64(8), object(9)
memory usage: 280.5+ KB
```

- No issue with NA. Great, so no rows to drop and no issues with misleading **`float`** type due to NA.
- Mixture of categorical and numerical features.

EDA in 5 seconds — `info`, `describe`, and `countplot` II

16
```
df.describe()
```

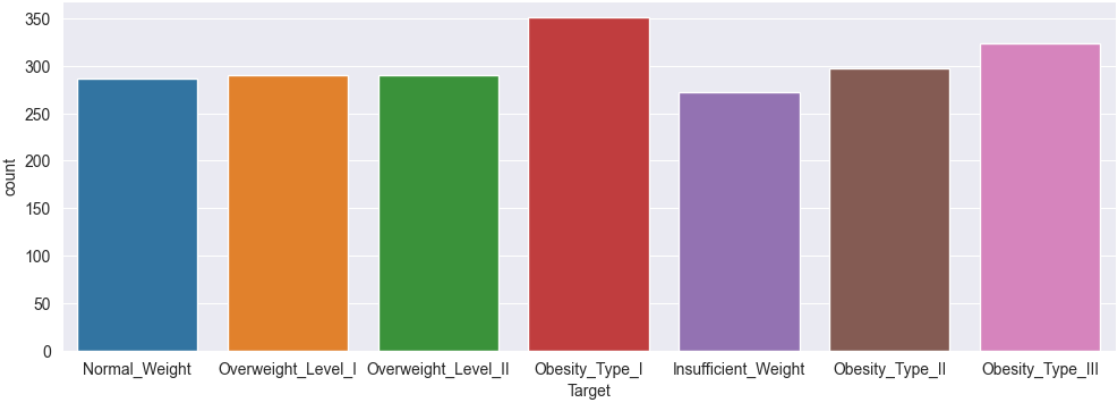
	Age	Height	Weight	FCVC	NCP	CH2O	FAF	TUE
count	2111.000000	2111.000000	2111.000000	2111.000000	2111.000000	2111.000000	2111.000000	2111.000000
mean	24.312600	1.701677	86.586058	2.419043	2.685628	2.008011	1.010298	0.657866
std	6.345968	0.093305	26.191172	0.533927	0.778039	0.612953	0.850592	0.608927
min	14.000000	1.450000	39.000000	1.000000	1.000000	1.000000	0.000000	0.000000
25%	19.947192	1.630000	65.473343	2.000000	2.658738	1.584812	0.124505	0.000000
50%	22.777890	1.700499	83.000000	2.385502	3.000000	2.000000	1.000000	0.625350
75%	26.000000	1.768464	107.430682	3.000000	3.000000	2.477420	1.666678	1.000000
max	61.000000	1.980000	173.000000	3.000000	4.000000	3.000000	3.000000	2.000000

- Different ranges among features $\Longrightarrow$ need scaling.
- No missing values — but still have `dropna` in workflow (for other datasets).

17
```
df.dropna(inplace=True)
```

EDA in 5 seconds — `info`, `describe`, and `countplot` III

```
18
plt.figure(figsize=(12,4))
sns.countplot(data=df, x='Target', hue='Target')
plt.show()
```



Target has 7 levels, but is evenly balanced $\Longrightarrow$ what is the expected accuracy of a random classifier?

Features

Separate the variables/features and the target ...

19
```
X = df.drop('Target', axis=1)
y = df['Target']
```

Need a robust (if not optimal) separation of categorical vs numerical features.

- Could use `dtype` but I was not confident that all dataset stored data in correct format.
- And remember what happens to **`int`** columns that contain missing values.
- Instead I used heuristic based on the number of distinct values to classify features.

20
```
cat_features = [c for c in X.columns if X[c].nunique()<=12]
num_features = [c for c in X.columns if c not in cat_features]
features = cat_features + num_features

assert len(set(features)) == len(features), f"Duplicated features. {features}."
assert len(set(features)) == X.shape[1], f"Missing some features. {len(set(features))} != {X.sha

print(f"{cat_features=}")
print(f"{num_features=}")
```

```
cat_features=['Gender', 'family_history_with_overweight', 'FAVC', 'CAEC', 'SMOKE', 'SCC', 'CALC', 'MTRANS']
num_features=['Age', 'Height', 'Weight', 'FCVC', 'NCP', 'CH2O', 'FAF', 'TUE']
```

Train-Test Split

Split the data into train (and validation) and test:

- Not doing any hyper-parameter tuning so don't need validation set — but `cross_validation` would be better anyway since dataset is small.
- Always use `stratify` in classification tasks.
- Always verify size of returned objects.

21
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.25, stratify=y, random_state=SEED)

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((1583, 16), (528, 16), (1583,), (528,))
```

Label Encoding of Target

Best practice is to train encoder on `train` dataset only and then, apply encoder to both `train` and `test` datasets. (Usually use `fit_transform` on `train`)

22
```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
le.fit(y_train)
y_train = le.transform(y_train)
y_test = le.transform(y_test)
```

Alternative approach

You could have applied label encoder before train-test split, as in

23
```
from sklearn.preprocessing import LabelEncoder

label_enc = LabelEncoder()
y = label_enc.fit_transform(y)
```

but 1st approach is better practise.

Feature Encoding

- Default (`StandardScaler`) scaling on numerical features, and one-hot encoding on categorical.
- I prefer to construct a dataframe with encoded features — aids model interpretation.

```
24 from sklearn.preprocessing import StandardScaler, OneHotEncoder

ss = StandardScaler()
ss.fit(X_train[num_features])

ohe = OneHotEncoder(handle_unknown='ignore', sparse_output=False)
ohe.fit(X_train[cat_features])

def encode(data):
    return pd.concat([
        pd.DataFrame(ss.transform(data[num_features]), columns=ss.get_feature_names_out()),
        pd.DataFrame(ohe.transform(data[cat_features]), columns=ohe.get_feature_names_out())
    ], axis=1)

X_train = encode(X_train)
X_test = encode(X_test)

X_train.shape, X_test.shape
```

Logistic Regression

We will use accuracy as metric to evaluate classifiers — no obvious reason not to.

25
```
from sklearn.metrics import accuracy_score, classification_report
```

Logistic regression is a decent (linear) classifier, especially after one-hot encoding.

26
```
from sklearn.linear_model import LogisticRegression

lr_model = LogisticRegression(max_iter=1000)
lr_model.fit(X_train, y_train)

y_pred = lr_model.predict(X_test)
lr_acc = accuracy_score(y_test, y_pred)
print(f'LR accuracy is {lr_acc:.4%}')
print(classification_report(y_test, y_pred))
```

```
LR accuracy is 85.2273%
              precision    recall  f1-score   support

           0       0.87      0.96      0.91        68
           1       0.79      0.68      0.73        72
           2       0.93      0.84      0.88        88
           3       0.96      0.96      0.96        74
           4       0.95      1.00      0.98        81
           5       0.71      0.71      0.71        73
           6       0.73      0.81      0.77        72

    accuracy                           0.85       528
   macro avg       0.85      0.85      0.85       528
weighted avg       0.85      0.85      0.85       528
```

85% is much better that random, and model has consistent performance within each class.

But is this a good model? Need context

K-Nearest Neighbours

27

```
from sklearn.neighbors import KNeighborsClassifier

knn_model = KNeighborsClassifier()
knn_model.fit(X_train, y_train)

y_pred = knn_model.predict(X_test)
knn_acc = accuracy_score(y_test, y_pred)
print(f'KNN accuracy is {knn_acc:.4%}')
print(classification_report(y_test, y_pred))
```

```
KNN accuracy is 81.6288%
              precision    recall  f1-score   support

           0       0.72      0.93      0.81        68
           1       0.72      0.29      0.42        72
           2       0.85      0.92      0.89        88
           3       0.94      0.97      0.95        74
           4       0.98      1.00      0.99        81
           5       0.67      0.77      0.72        73
           6       0.78      0.79      0.79        72

    accuracy                           0.82       528
   macro avg       0.81      0.81      0.79       528
weighted avg       0.81      0.82      0.80       528
```

Model is comparable but little worse than Logistic Regression.

Note model has more difficulty in identifying class 1? Why?

Decision Trees

28

```python
from sklearn.tree import DecisionTreeClassifier

dt_model = DecisionTreeClassifier(random_state=SEED)
dt_model.fit(X_train, y_train)

y_pred = dt_model.predict(X_test)
dt_acc = accuracy_score(y_test, y_pred)
print(f'DT accuracy is {dt_acc:.4%}')
print(classification_report(y_test, y_pred))
```

```
DT accuracy is 92.2348%
              precision    recall  f1-score   support

           0       0.89      0.99      0.94        68
           1       0.88      0.79      0.83        72
           2       0.94      0.91      0.92        88
           3       0.99      0.95      0.97        74
           4       0.99      1.00      0.99        81
           5       0.87      0.89      0.88        73
           6       0.89      0.93      0.91        72

    accuracy                           0.92       528
   macro avg       0.92      0.92      0.92       528
weighted avg       0.92      0.92      0.92       528
```

Decision tree model is best so far, what about niäve Bayes, or … or …?

Approach here "works" but is a lot of repeated code.

Loop over Estimates – A List of Tuples

- Create list of classifiers with (name, instance) pairs
- Simplest approach, and accepted as input in some `sklearn` ensemble models.

```
29 from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

estimators = [
    ('DT', DecisionTreeClassifier(random_state=SEED)),
    ('KNN', KNeighborsClassifier()),
    ('LR', LogisticRegression(max_iter=1000)),
    ('RF', RandomForestClassifier(random_state=SEED)),
]

for name, model in estimators:
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    score = accuracy_score(y_test, y_pred)
    print(f'{name:10} accuracy is {score:.4%}')
```

```
DT         accuracy is 92.2348%
KNN        accuracy is 81.6288%
LR         accuracy is 85.2273%
RF         accuracy is 91.6667%
```

Loop over Estimates – A List of Dictionaries

- Create a list of dictionaries — more code, but more useful (can store predictions/resuilts).
- Can convert to dataframe for display/saving.

```
30 estimators = [
       {'name':'DT', 'model':DecisionTreeClassifier(random_state=SEED)},
       {'name':'KNN', 'model':KNeighborsClassifier()},
       {'name':'LR', 'model':LogisticRegression(max_iter=1000)},
       {'name':'RF', 'model':RandomForestClassifier(random_state=SEED)},
   ]

   for estimator in estimators:
       name, model = estimator['name'], estimator['model']
       model.fit(X_train, y_train)
       y_pred = model.predict(X_test)
       score = accuracy_score(y_test, y_pred)
       estimator['y_pred'] = y_pred
       estimator['accuracy'] = score

   pd.DataFrame.from_dict(estimators)
```

	name	model	y_pred	accuracy
0	DT	DecisionTreeClassifier(random_state=2025)	[4, 3, 6, 4, 2, 2, 5, 6, 3, 0, 4, 0, 4, 2, 1, ...	0.922348
1	KNN	KNeighborsClassifier()	[4, 3, 6, 4, 2, 2, 5, 6, 2, 0, 4, 0, 4, 2, 5, ...	0.816288
2	LR	LogisticRegression(max_iter=1000)	[4, 3, 6, 4, 2, 6, 5, 6, 2, 0, 4, 0, 4, 2, 5, ...	0.852273
3	RF	(DecisionTreeClassifier(max_features='sqrt', r...	[4, 3, 6, 4, 2, 2, 5, 6, 2, 1, 4, 0, 4, 2, 1, ...	0.916667

OK, enough base estimators,
now let's see about some ensembles

Outline

Voting classifier

Voting is an ensemble technique that combines multiple models' predictions.

- Two types: Hard Voting and Soft Voting.

	Hard Voting	**Soft Voting**
Decision Rule	Majority class vote	Weighted probability average
Base Model Predictions	Discrete (class labels)	Continuous (probabilities)
Best for	High-confidence classifiers	Well-calibrated probability models
Risk	Can ignore useful probability info	Sensitive to poorly calibrated models
Example Calculation	$\arg\max \sum I(y_i = c)$	$\arg\max \sum w_i \Pr(y_i = c)$

- Use hard voting when base models are diverse and high-confidence (i.e., probability estimates are reliable).
- Use soft voting when models output well-calibrated probabilities of when models have different reliability levels (since soft voting allows weighting).

Voting Classifier (Hard)

```
estimators = [
    ('DT', DecisionTreeClassifier(random_state=SEED)),
    ('KNN', KNeighborsClassifier()),
    ('LR', LogisticRegression(max_iter=1000)),
    ('RF', RandomForestClassifier(random_state=SEED)),
]

from sklearn.ensemble import VotingClassifier

voting_classifier = VotingClassifier(estimators=estimators, voting='hard')
voting_classifier.fit(X_train, y_train)

y_pred = voting_classifier.predict(X_test)
voting_hard_acc = accuracy_score(y_test, y_pr
print(f'Voting Ensemble (hard) accuracy is {vot
print(classification_report(y_test, y_pred))
```

```
Voting Ensemble (hard) accuracy is 92.2348%
              precision    recall  f1-score   support

           0       0.88      1.00      0.94        68
           1       0.82      0.81      0.81        72
           2       0.97      0.95      0.96        88
           3       1.00      0.96      0.98        74
           4       0.99      1.00      0.99        81
           5       0.83      0.81      0.82        73
           6       0.96      0.92      0.94        72

    accuracy                           0.92       528
   macro avg       0.92      0.92      0.92       528
weighted avg       0.92      0.92      0.92       528
```

OK, it only matched the best of the base estimators (DT), but notice range of the individual class f1-scores has narrowed.

Voting Classifier (Soft)

```
32 voting_classifier = VotingClassifier(estimators=estimators, voting='soft')
   voting_classifier.fit(X_train, y_train)

   y_pred = voting_classifier.predict(X_test)
   voting_soft_acc = accuracy_score(y_test, y_pred)
   print(f'Voting Ensemble (soft) accuracy is {vot
   print(classification_report(y_test, y_pred))
```

```
Voting Ensemble (soft) accuracy is 94.6970%
              precision    recall  f1-score   support

           0       0.91      0.99      0.94        68
           1       0.91      0.85      0.88        72
           2       0.97      0.94      0.95        88
           3       1.00      0.96      0.98        74
           4       0.98      1.00      0.99        81
           5       0.92      0.92      0.92        73
           6       0.95      0.97      0.96        72

    accuracy                           0.95       528
   macro avg       0.95      0.95      0.95       528
weighted avg       0.95      0.95      0.95       528
```

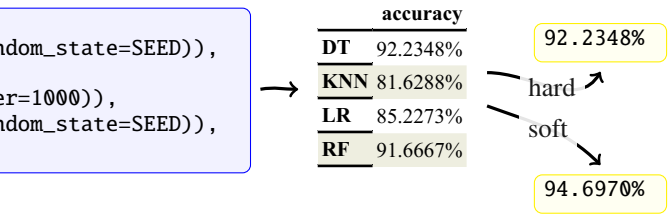
Best so far: 94.697%

Soft voting generally achieves better performance than hard voting, because it assigns higher weight to the predictions with high confidence by the individual models.

Summary to Date

The current voting classifier and performance ...

```
33 estimators = [
       ('DT', DecisionTreeClassifier(random_state=SEED)),
       ('KNN', KNeighborsClassifier()),
       ('LR', LogisticRegression(max_iter=1000)),
       ('RF', RandomForestClassifier(random_state=SEED)),
   ]
```



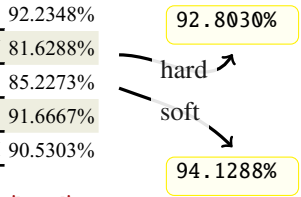
- `VotingClassifier` performs only as well as best base estimator (DT) when using `voting='hard'`.
- Best outcome is with `VotingClassifier` and `voting='hard'`.
- But can we improve on this?
  - What about adding more base estimators that score well?
  - What about dropping some of the poorer performing classifiers?

Will Adding More (Good) Estimators Improve the Ensemble?

Lets add a support vector classifier. The default scores† over 91% so better than half of the existing estimators . . .

```
34 estimators = [
       ('DT', DecisionTreeClassifier(random_state=SEED)),
       ('KNN', KNeighborsClassifier()),
       ('LR', LogisticRegression(max_iter=1000)),
       ('RF', RandomForestClassifier(random_state=SEED)),
       ('SVM', SVC(probability=True)),
   ]
```

	accuracy
DT	92.2348%
KNN	81.6288%
LR	85.2273%
RF	91.6667%
SVM	90.5303%



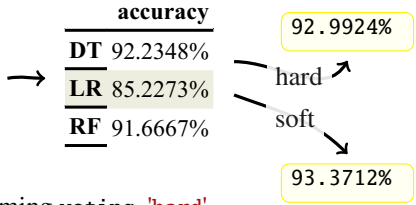
- Adding a good performing classifier resulted in better performing `voting='hard'` classifier but a worse `voting='soft'` classifier. Why?
  - *Correlation among Classifiers* — if new classifier's predictions are not independent or diverse enough, it may not contribute new information, reducing the overall effectiveness of the ensemble.
  - *Overfitting* — by adding another classifier, the ensemble might overfit to the training data, especially if the new classifier itself overfits.

†The parameter `probability=True` causes the classifier to generate probability predictions, needed for soft voting classifier.

What About Dropping Some of the Poorer Performing Classifiers?

- Lets drop k-nearest neighbour classifier, since it is the worst performing classifier.

```
35 estimators = [
       ('DT', DecisionTreeClassifier(random_state=SEED)),
       ('LR', LogisticRegression(max_iter=1000)),
       ('RF', RandomForestClassifier(random_state=SEED)),
   ]
```



- Dropping the worst performing classifier resulted better performing `voting='hard'` classifier but a worse `voting='soft'` classifier. Why?
  - *Correlation among Classifiers* — if the dropped classifier's predictions were more independent, its removal could reduce the overall effectiveness of the ensemble.
  - *Performance on Specific Cases* — it is possible that the dropped classifier was better on specific cases then even though its overall score was lower. So its removal negatively affected the ensemble.
  - *Ensemble Robustness* — Voting classifiers rely on the collective decision-making of their base estimators. Removing one estimator may disrupt a balance in decision aggregation, leading to poorer performance on some samples.

Expected the unexpected — evaluate decisions

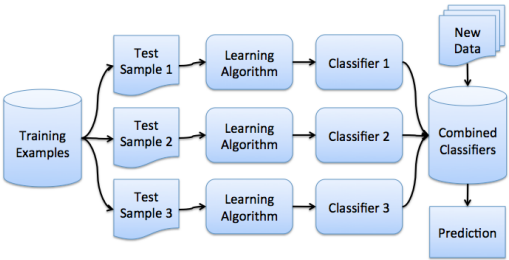
Outline

Bagging aka Bootstrap AGgregation

Bagging is the application of the Bootstrap procedure to high-variance machine learning algorithm, typically decision trees.

- A sample of the observations (row) is selected randomly with replacement(Bootstrapping).
- A sample of features (columns) is selected to create a model trained in the observations.

This is a big deal - as a by product of sampling the features, the classifier can also report on feature importance. (See RandomForest Classifier)



- Repeat to create many models and models can be trained in parallel.
  - When bagging with decision trees, we are less concerned about individual trees overfitting the training data. For this reason and for efficiency, the individual decision trees are grown deep (e.g. few training samples at each leaf-node of the tree) and the trees are not pruned.
  - These trees will have both high variance and low bias. These are important characteristics of sub-models when combining predictions using bagging.
- Prediction is given based on the aggregation of predictions from all the models.

Bagging aka Bootstrap AGgregation

Method

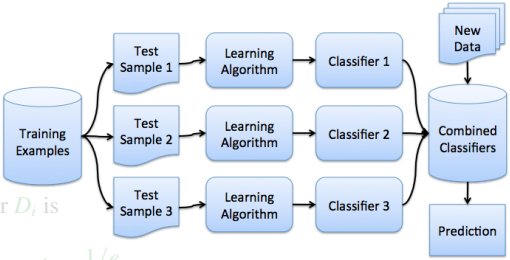
- Create M bootstrap samples, $D_1, D_2, \ldots, D_M$ of size N.
  - When picking each element of sample D_i, each element in D has probability of $1/N$ of being selected.
  - The probability of an element of D not being selected for D_i is

$$(1 - 1/N)^N \xrightarrow[N \to \infty]{} 1/e$$

  - Hence the probability of an element of D being selected for D_i is $1 - 1/e = 0.632$.

A bootstrap sample contains 63% of the original data.

- Separately train classifier on each D_i.
- Classify new instance by majority vote / average.



[†]Leo Breiman (1994)

Bagging aka Bootstrap AGgregation

Method

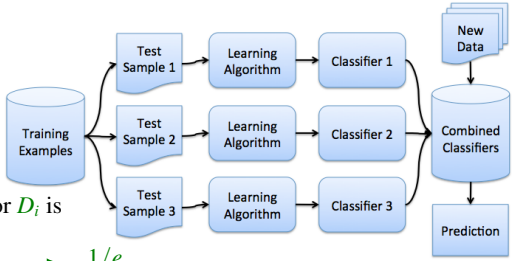
- Create M bootstrap samples, $D_1, D_2, \ldots, D_M$ of size N.
  - When picking each element of sample D_i, each element in D has probability of $1/N$ of being selected.
  - The probability of an element of D not being selected for D_i is

$$(1 - 1/N)^N \xrightarrow[N \to \infty]{} 1/e$$

  - Hence the probability of an element of D being selected for D_i is $1 - 1/e = 0.632$.

A bootstrap sample contains 63% of the original data.

- Separately train classifier on each D_i.
- Classify new instance by majority vote / average.



[†]Leo Breiman (1994)

Bagging — Performance I

Definition 3 (Unstable learner)

A learner is unstable if its output classifier undergoes major changes in response to small changes in training data

- Unstable: decision-tree, neural network, rule learning algorithms, ...
- Stable: linear regression, nearest neighbour, linear threshold algorithms, ...

Bagging tends to

- works well for unstable learners
- can have a mild negative effect on the performance of stable methods

Best Case

$$\mathrm{Var}\Big(\mathrm{Bagging}\big(L(x,D)\big)\Big) = \frac{\mathrm{Var}\big(L(x,D)\big)}{M}$$

However, in practice the models are correlated, so reduction is smaller than $1/M$. Also variance of models trained on fewer training cases can be somewhat larger.

Bagging — Performance II

- Bagging reduces the variance of a classifier by decreasing the difference in error when we train the model on different datasets.
- In other words, bagging prevents overfitting.
- The efficiency of bagging comes from the fact that the individual models are quite different due to the different training data and their errors cancel each other out during voting.
- Additionally, outliers are likely omitted in some of the training bootstrap samples.
- Bagging is effective on small datasets.
  - Dropping even a small part of training data leads to constructing substantially different base classifiers.
  - If you have a large dataset, you would generate bootstrap samples of a much smaller size.

Example

The skikit-learn documentation has a simulation showing the effect of bagging.

Bagging with sklearn's `BaggingClassifier`

```
36 from sklearn.ensemble import BaggingClassifier
   from sklearn.tree import DecisionTreeClassifier

   bagging_classifier = BaggingClassifier(
       DecisionTreeClassifier(),
       random_state=SEED
   )
   bagging_classifier.fit(X_train, y_train)
   bagging_preds = bagging_classifier.predict(X_test)
   bagging_acc = accuracy_score(y_test, bagging_preds)
   print(f'Bagging Ensemble accuracy is {bagging_acc:.4%}')
```

```
Bagging Ensemble accuracy is 93.5606%
```

- The 93.5% is not as good as the best voting classifier result. However, bagging ensembles often outperform voting ensembles, and they can decrease the overfitting of decision trees (which tend to overfit easily).
- Random forest can be considered a bagging ensemble with the `max_samples=1`, that is, it uses the entire training set to train the based learners.

Bagging with sklearn's `BaggingClassifier` + Hyper-parameter Tuning

```
37 from sklearn.ensemble import BaggingClassifier
   from sklearn.tree import DecisionTreeClassifier

   bagging_classifier = BaggingClassifier(
       DecisionTreeClassifier(max_depth=10, class_weight='balanced'),
       n_estimators=30, max_samples=0.5, max_features=0.5, bootstrap=False,
       random_state=SEED, n_jobs=-1
   )
   bagging_classifier.fit(X_train, y_train)
   bagging_preds = bagging_classifier.predict(X_test)
   bagging_acc = accuracy_score(y_test, bagging_preds)
   print(f'Bagging Ensemble accuracy is {bagging_acc:.4%}')
```

```
Bagging Ensemble accuracy is 95.4545%
```

- OK, in the above I semi-randomly picked values for the hyper-parameter tuning, so don't know if this result can be improved on. But it is the best so far.
- Bagging ensembles can be trained in parallel using different CPU cores, using `n_jobs=-1`, which can reduce the processing time.

Outline

Motivation — ‘How May I Help You?’

Boosting refers to a group of algorithms that utilise weighted averages to make weak learners into stronger learners.

Problem

Automatically categorise type of call requested by phone customer (`Collect`, `CallingCard`, `PersonToPerson`, etc.)

- ‘Yes I’d like to place a collect call long distance please’ (`Collect`)
- ‘Operator I need to make a call but I need to bill it to my office’ (`ThirdNumber`)
- ‘Yes I’d like to place a call on my master card please’ (`CallingCard`)
- ‘I just called a number in sioux city and I musta rang the wrong number because I got the wrong party and I would like to have that taken off of my bill’ (`BillingCredit`)

Observations

- Easy to find ‘rules of thumb’ that are ‘often’ correct
  e.g. ‘IF "card" occurs in utterance THEN predict ‘`CallingCard`’
- Hard to find single highly accurate prediction rule.

[†] Gorin et al

Boosting Approach

Outline

- Devise procedure for deriving rough rules of thumb
- Apply procedure to subset of examples
- Obtain rule of thumb
- Apply to 2nd subset of examples
- Obtain 2nd rule of thumb
- Repeat T times

Key Steps

- How do we choose examples on each round?
  - Concentrate on hardest examples (those most often miss-classified by previous rules of thumb) — focus by sampling or weighing the whole data set.
- How do we combine rules of thumb into a single prediction rule?
  - Take (weighted) majority vote of rules of thumb.

If Boosting is possible, then

- can use (fairly) wild guesses to produce highly accurate predictions.
- for any learning problem:
  - either can always learn with nearly perfect accuracy
  - or there exist cases where cannot learn even slightly better than random guessing

Boosting

Boosting
General method of converting rough rules of thumb into highly accurate prediction rule.

- Assume given 'weak' learning algorithm that can consistently find classifiers ('rules of thumb') at least slightly better than random, say, accuracy $\geq 55\%$ (in two-class setting).

'weak learning assumption'

- Then given sufficient data, a boosting algorithm can provably construct single classifier with very high accuracy, say, 99%.
- In contrast to bagging which has little effect on Bias, boosting reduces both bias and variance.

History

Schapire '89 — first provable boosting algorithm

Freund '90 — 'optimal' algorithm that 'boosts by majority'

nd & Schapire '95 — introduced 'AdaBoost' algorithm, strong practical advantages over previous boosting algorithms

AdaBoost (Adaptive Boosting) Algorithm I

Outline

- First train the base classifier on all the training data with equal importance weights on each case.

  Given weights and data how do we train a classifier?

- Then re-weight the training data to emphasise the hard cases and train a second model.

  How do we re-weight the data?

- Repeat above steps.
- Finally, use a weighted committee of all the models for the test data.

  How do we weight the models in the committee?

Notation

- Input: feature matrix, X; target vector, $\boldsymbol{y} \in \{-1, 1\}$
  - Recall: X_m is m(th) feature/column and X^n is n(th) row/example — superscript represent rows/examples/cases.
- Output: m(th) classifier predicted output $\boldsymbol{f}_m(X) = \{-1, 1\}$

AdaBoost Algorithm — How do we train a classifier?

Weights

Let $\boldsymbol{w}_m^n$ represent the weights of example n for classifier m. With

$$\boldsymbol{w}_1^n = 1/N$$

Whenever, we change these weights, we will rescale so that they sum to one.

Training

Given feature matrix, X, target vector, $\boldsymbol{y}$ and weights w_m^n, we train a (weak) classifier using cost function for classifier m:

$$J_m = \sum_{n=1}^{N} w_m^n \underbrace{\left[f_m(X^n) \neq y^n\right]}_{\text{1 if error, else 0}} = \sum \text{weighted errors}$$

AdaBoost Algorithm — How do we update weights?

- Define the unnormalized error rate of a classifier as

$$\epsilon_m = J_m$$

and the quality of the classifier as

$$\alpha_m = \frac{1}{2} \ln \left(\frac{1 - \epsilon_m}{\epsilon_m} \right)$$

This is zero if the classifier has weighted error rate of 0.5 and infinity if the classifier is perfect.

- The weights for the next round are then

$$w_{m+1}^n = w_m^n \cdot \frac{\exp\{-\alpha_m y^n f_m(X^n)\}}{\sum_{n=1}^N w_m^n \exp\{-\alpha_m y^n f_m(X^n)\}}$$

Notice the product $y^n f_m(X^n)$.

- This is +1 when prediction matches actual and -1 otherwise.
- So $\exp\{-\alpha_m y^n f_m(X^n)\}$ will be small when prediction matches actual and big otherwise $\implies$ increases weight for harder cases to focus on them.

AdaBoost Algorithm — How do we make predictions?

After M boosting iterations, we have m classifiers. To use in prediction of new cases we weight the binary prediction of each classifier by the quality of that classifier:

$$f(X_{\text{test}}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m f_m(X_{\text{test}})\right)$$

Bagging vs Boosting

Unlike bagging that had each model run independently and then aggregate the outputs at the end without preference to any model. Boosting is all about "teamwork". Each model that runs, dictates what features the next model will focus on.

- Bagging and Boosting decrease the variance of your single estimate as they combine several estimates from different models. So both may result in a model with higher stability.
- If the single model has low bias (under-fitting) then bagging is unlikely to improve this. However, Boosting could generate a combined model with lower errors as the sequence of model should have increased bias (so better fit the signal).
- If the single model has high variance (over-fitting) then bagging is the better option. Boosting does not address over-fitting (in fact, can increase it). Bagging, due to its averaging should reduce variance.

[†] See Bagging and Random Forest Ensemble Algorithms for Machine Learning https://machinelearningmastery.com/bagging-and-random-forest-ensemble-algorithms-for-machine-learning/

AdaBoost using `sklrean.AdaBoostClassifier`


```
38 from sklearn.ensemble import AdaBoostClassifier
   from sklearn.tree import DecisionTreeClassifier

   adaboost_classifier = AdaBoostClassifier(
       estimator=DecisionTreeClassifier(),
       algorithm='SAMME',
       random_state=SEED)

   adaboost_classifier.fit(X_train, y_train)

   y_pred = adaboost_classifier.predict(X_test)
   adaboost_acc = accuracy_score(y_test, y_pred)
   print('adaboost accuracy is {0:7.4f} %'.format(adaboost_acc*100))
```

```
adaboost accuracy is 93.1818 %
```

after minimal hyper-parameter tuning
(was 90.3% with no tuning)
CAN get over 97% with more tuning (not shown here)

- Uses many weak learners, usually shallow decision stumps (trees of depth=1).
- Models are trained sequentially, adjusting sample weights:
- Misclassified points get higher weights in the next iteration.
- Final prediction is a weighted vote of all weak learners.
- Works well with simple models, robust to overfitting, but can struggle with noisy data and outliers.

Gradient Boosting using `sklearn.GradientBoostingClassifier` I

Algorithm

- The individual (or base) models in gradient boosting are decision rrees.
- The base models are fitted sequentially, each new model is fit to minimise a loss function based on the error (gradient) of a loss function made by the previous models:
  - the initial model is trained on the full dataset
  - and the errors (gradients) of the loss function are obtained based on the difference between the predicted class of the model and the target label.
  - The next model is trained by using the negative gradient of the loss function in an attempt to correct the mistakes made by the previous model.
  - Repeat for each base model.

  Ideally, as more models are trained the errors should become smaller, that is, the predictions by the models should better match the target labels.
- The residual errors represent the gradient of the loss function with respect to the predicted values
  $\Longrightarrow$ called Gradient Boosting.

Gradient Boosting using `sklearn.GradientBoostingClassifier` II

```
39 from sklearn.ensemble import GradientBoostingClassifier

grad_boost_classifier = GradientBoostingClassifier(
    n_estimators=500,
    learning_rate=0.1,
    max_depth=2,
    random_state=SEED)

grad_boost_classifier.fit(X_train, y_train)

gboost_preds = grad_boost_classifier.predict(X_test)
gboost_acc = accuracy_score(y_test, gboost_preds)
print(f'GradientBoostingClassifier accuracy is {gboost_acc:.4%}')
```

```
GradientBoostingClassifier accuracy is 94.5076%
```

no hyper-parameter tuning

- In `sklearn`, the Gradient Boosting Classifier has hyper-parameters, most important are:
  - `n_estimators` — the number of estimators i.e. decision trees.
  - `max_depth` — maximum depth of the trees.
  - `learning_rate` — how quickly the models are updated. If the learning rate is too high, the models will be updated more quickly, which can cause overfitting.

eXtreme Gradient Boosting using `xgboost.XGBClassifier`

```
40 import xgboost as xgb

   xgb_classifier = xgb.XGBClassifier(randc
   xgb_classifier.fit(X_train, y_train)

   xgboost_preds = xgb_classifier.predict(X_test)
   xgboost_acc = accuracy_score(y_test, xgboost_preds)
   print('XGBoost accuracy is {0:7.4f} %'.format(xgboost_acc*100))
```

```
XGBoost accuracy is 96.4015 %
```

no hyper-parameter tuning

- Not part of `sklearn`, so install package `xgboost` first.
- XGBoost or eXtreme Gradient Boosting ensemble has several modifications to the original Gradient Boosting algorithm, which speed up the training process and improve the predictive performance:
  - Includes the L_1 and L_2 regularisation terms into the objective function to reduce overfitting.
  - uses a more efficient tree-building algorithm that relies on histogram-based methods and pruning techniques to optimise the tree splitting.
  - Can also handle missing data, either by treating the data samples with missing values as a separate category, or it can apply techniques for filling in the missing values. Therefore, it can work without significant exploratory data analysis and preprocessing.

LightGBM using `lightgbm.sklearn.LGBMClassifier`

```
41 # import lightgbm
from lightgbm.sklearn import LGBMClassifier

lightgbm_classifier = LGBMClassifier(verbose=-1,
    force_row_wise=True,
    random_state=SEED)
lightgbm_classifier.fit(X_train, y_train)

y_pred = lightgbm_classifier.predict(X_test)
lightgbm_acc = accuracy_score(y_test, y_pred)
print('lightgbm accuracy is {0:7.4f} %'.format(lightgbm_acc*100))
```

```
lightgbm accuracy is 96.0227 %
```

no hyper-parameter tuning

- LightGBM or Light Gradient Boosting Machine is another optimised version of gradient boosting ensemble, developed by Microsoft.
- It uses a histogram-based approach for finding optimal splitting of the decision trees.
- And a leaf-wise tree growth strategy that can result in deeper trees compared to other ensemble algorithms.

Categorical Boosting using `catboost.CatBoostClassifier`

```
42 # import catboost
   from catboost import CatBoostClassifier

   catboost_classifier = CatBoostClassifier(verbose=False,
       random_state=SEED)
   catboost_classifier.fit(X_train, y_train)

   y_pred = catboost_classifier.predict(X_test)
   catboost_acc = accuracy_score(y_test, y_pred)
   print('catboost accuracy is {0:7.4f} %'.format(catboost_acc*100))
```

```
catboost accuracy is 94.6970 %
```

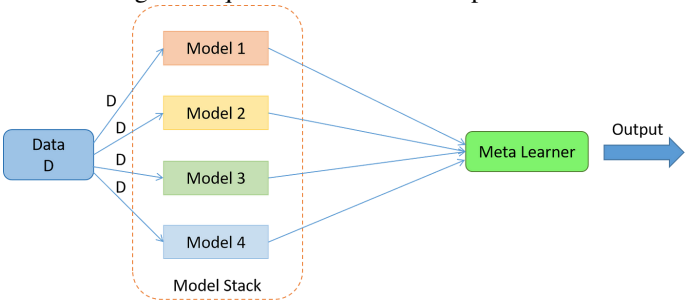
no hyper-parameter tuning

- CatBoost or Categorical Boosting is specifically designed for handling categorical features.
- Can operate directly on data with categorical features, without the requirement for applying ordinal, label encoding or other preprocessing techniques.
- Uses an ordered gradient boosting technique for direct processing of categorical features.
- Uses specialised regularisation techniques (such as depth regularisation and feature permutation-based regularisation) and implementation of specialised symmetric trees structure to prevent overfitting.
- Similar to XGBoost, CatBoost has also support for dealing with missing data values, which simplifies the data preprocessing.

Outline

Stacking Ensemble Models

- Stacking (Stacked Generalisation) is an ensemble learning technique that combines multiple base models using a meta-model.

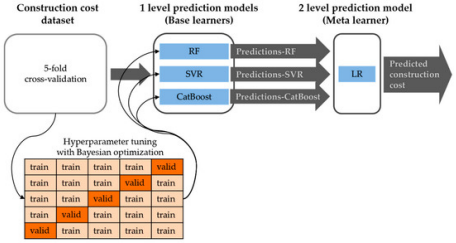


- Unlike bagging and boosting, stacking focuses on leveraging different model types rather than modifying training data.
- The base models (level-0 learners) make predictions, and their outputs are used as inputs for the meta-model (level-1 learner).
- The meta-model learns how to best combine the base model predictions to improve overall accuracy.

Stacking vs. Bagging vs. Boosting

	Bagging	Boosting	Stacking
Goal	Reduce variance	Reduce bias	Improve generalization
Base Model Independence	Identical models	Sequential dependence	Diverse models
Model Training	Parallel	Sequential (adaptive)	Parallel + meta-model
Risk of Overfitting	Low	Higher	Moderate (depends on meta-model)
Computational Cost	Moderate	High	Very High

- Can combine techniques to get arbitrary complicated models.
- In particular, interleaving hyper-parameter tuning can be effective.
- For example, see https://doi.org/10.3390/app12199729



Stacking (Imports and setup)

Could just use option cv in StackingClassifier since dataset is balanced, but want pipeline that can also handle unbalanced datasets.

```
43 from sklearn.model_selection import StratifiedKFold
   from sklearn.ensemble import StackingClassifier
```

Create a list of base models, want them to be as different as possible . . .

```
44 estimators = [
       ('DT', DecisionTreeClassifier(random_state=SEED)),
       ('KNN', KNeighborsClassifier()),
       ('LR', LogisticRegression(max_iter=1000)),
       ('RF', RandomForestClassifier(random_state=SEED)),
       ('XGB', xgb.XGBClassifier(random_state=SEED)),
   ]

   final_estimator = xgb.XGBClassifier(random_state=SEED)
```

Stacking

```
cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=SEED)

stacking_model = StackingClassifier(
    estimators=estimators,
    final_estimator=final_estimator,
    passthrough=True, cv=cv,
    stack_method="predict_proba")

stacking_model.fit(X_train, y_train)

y_pred = stacking_model.predict(X_test)
stacking_acc = accuracy_score(y_test, y_pred)
print(f'stacking accuracy is {stacking_acc:.4%}')
print(classification_report(y_test, y_pred))
```

```
stacking accuracy is 95.6439%
              precision    recall  f1-score   support

           0       0.97      0.94      0.96        68
           1       0.85      0.96      0.90        72
           2       0.98      0.95      0.97        88
           3       0.99      0.96      0.97        74
           4       0.99      1.00      0.99        81
           5       0.98      0.88      0.93        73
           6       0.95      1.00      0.97        72

    accuracy                           0.96       528
   macro avg       0.96      0.96      0.96       528
weighted avg       0.96      0.96      0.96       528
```

Summary

	Model	Accuracy	Rank
0	LR	85.23%	13
1	KNN	81.63%	14
2	DT	92.23%	10
3	SVC	90.53%	12
4	RF	91.67%	11
5	Voting hard(DT+KNN+LR+RF)	92.23%	10
6	Voting soft(DT+KNN+LR+RF)	94.70%	6
7	AdaBoost	93.18%	8
8	AdaBoost + Tuning	96.21%	2
9	GBoost	94.51%	7
10	XGBoost	96.40%	1
11	LightBoost	96.02%	3
12	CatBoost	94.70%	6
13	Stacking	95.83%	4

Take home messages

- It is difficult to predict which type of ensemble model will perform best — model selection is a big deal.
- Impact of hyper-parameter tuning is not covered here but, based on AdaBoost, it can have a significant impact.
  - Possible to get over 97% using ensemble model with a little hyper-parameter tuning.
- Keep a log of your models and their performance.