# Ensemble Models

# Ensemble Models

**We finished the earlier notebook with the following considerations**

- DTs are non-linear models and quite prone to overfitting
- This can be mitigated (e.g. by reducing depth), but often with a loss of accuracy
- There is no easy fix by using a single DT

**What about using multiple DTs?**

For example, we could think of training several, different, DTs

- Each disting DT would risk overfitting, but in a different way
- Intuitively, overfitting would then appear as random noise
- ...Which could be canceled out via aggregation (averaging, voting)

**This is the key idea behind ensemble DT models**

We will (briefly) discuss a few of the main approach in this class

# Random Forests

**Random Forests work exactly as we have described**

We assume we built $m$ be different trees (estimators):

- Let $f_i(x)$ be the output of the $i$-th tree
- Let $\mathbb{I}(f_i(x) = k)$ is the indicator function for $f_i(x) = k$
- ...which is equal to 1 if $f_i(x) = k$ and 0 otherwise

**Then the output of a RF for classification is given by:**

$$f(x) = \text{argmax}_{k \in K} \sum_{i=1}^{m} \mathbb{I}(f_i(x) = k)$$

- Intuitively: every tree "votes" for a class
- ...And we pick the class with the largest number of votes

# Random Forests

**We can use Random Forests for regression, too**

In this case the model output is just an average:

$$f(x) = \frac{1}{m} \sum_{i=1}^{m} f_i(x)$$

**We obtain different trees**

...By using slighly different daset to train them. In particular we use:

- Bootstrapping: we select random subsets of the examples
- Bagging: we select random subsets of the attributes

**They are called Random Forests due to:**

- The use of multiple trees
- The introduction of random elements

# Random Forests

## We can learn a Random Forest model as usual

First we build a model object

```
In [36]: from sklearn.ensemble import RandomForestRegressor
         rfm = RandomForestRegressor(n_estimators=150, max_depth=20)
```

- We can specify the number of estimators (along the usual parameters)

Then we train the model:

```
In [37]: rfm.fit(X_tr, y_tr);
```

...And finally we obtain the predictions:

```
In [38]: rfm_pred_tr, rfm_pred_ts = rfm.predict(X_tr), rfm.predict(X_ts)
         print(f'R2: {r2_score(y_tr, rfm_pred_tr):.3} (training), {r2_score(y_ts, rfm_pred_ts):.3} (test)

R2: 0.965 (training), 0.746 (test)
```

# Random Forests

## We can optimize the parameters using grid search and cross validation

This is interesting to check how RF differ from simple DTs

```
In [39]: param_grid = {'n_estimators': np.arange(50, 200, 50), 'max_depth': np.arange(2, 10, 2)}
         rfm_cv = GridSearchCV(rfm, param_grid=param_grid)
         rfm_cv.fit(X_tr, y_tr);
```

The results are not necessarily better

```
In [40]: rfm_cv_pred_tr, rfm_cv_pred_ts = rfm_cv.predict(X_tr), rfm_cv.predict(X_ts)
         print(f'R2: {r2_score(y_tr, rfm_cv_pred_tr):.3} (training), {r2_score(y_ts, rfm_cv_pred_ts):.3}
```

```
R2: 0.925 (training), 0.737 (test)
```

...But the optimal parameters are!

```
In [41]: print(f'Best results with: {rfm_cv.best_params_}')
```

```
Best results with: {'max_depth': 6, 'n_estimators': 50}
```

# Random Forests

## RF and DTs operate in a very different way

The basic ingredient is still a DT

- DTs need to be deep in order to be expressive
- ...But deeper DTs are also very prone to overfitting

RFs have an innate mechanism to counter overfitting

- ...And therefore they can afford being much deeper
- This makes them much more reliable at learning complex relations

They are a bit slower to train

- ...But still fast compared to other complex ML models

# Gradient Boosted Trees

**Gradient Boosted Trees also employ multiple DTs, but in a different fashion**

Technically, the output of a GBT model is just the sum of the individual ouputs

$$f(x) = \sum_{i=1}^{m} f_i(x)$$

- Where the notation is the same we used for random forests

**However, the trees are trained sequentially:**

- The first tree (i.e. $f_1(x)$) is trained as usual
- The second tree is trained to apply a correction over $f_1(x)$
- The third to apply a correction over $f_1(x) + f_2(x)$, etc.

# Gradient Boosted Trees

**The corrections are incremental:**

- Each new tree does not try to completely fix the previous predictions
- ...But rather to make a step in the right direction
- ...As given by an exact or approximate gradient

**There is no need for randomization, in principle:**

- The incremental training process already yields different trees
- However, the scikit-learn implementation has some random elements

**The idea of using a "chain" of corrective models is general**

- It is called gradient boosting and can be applied with other model types
- ...But GBTs are the probably the best known example

# Gradient Boosted Trees

## We can learn Gradient Boosted Trees model as usual

First we build a model object

```
In [42]: from sklearn.ensemble import GradientBoostingRegressor
         gbm = GradientBoostingRegressor(n_estimators=100, max_depth=3)
```

- We can specify the number of estimators (along the usual parameters)

Then we train the model:

```
In [43]: gbm.fit(X_tr, y_tr);
```

...And finally we obtain the predictions:

```
In [44]: gbm_pred_tr, gbm_pred_ts = gbm.predict(X_tr), gbm.predict(X_ts)
         print(f'R2: {r2_score(y_tr, gbm_pred_tr):.3} (training), {r2_score(y_ts, gbm_pred_ts):.3} (test)

         R2: 0.944 (training), 0.723 (test)
```

# Gradient Boosted Trees

## We can optimize the parameters using grid search and cross validation

This is interesting to check how GBTs differ from RFs

In [45]:
```
param_grid = {'n_estimators': np.arange(50, 200, 50), 'max_depth': np.arange(2, 10, 2)}
gbm_cv = GridSearchCV(gbm, param_grid=param_grid)
gbm_cv.fit(X_tr, y_tr);
```

The results should be more or less comparable with RFs

In [46]:
```
gbm_cv_pred_tr, gbm_cv_pred_ts = gbm_cv.predict(X_tr), gbm_cv.predict(X_ts)
print(f'R2: {r2_score(y_tr, gbm_cv_pred_tr):.3} (training), {r2_score(y_ts, gbm_cv_pred_ts):.3}
```

```
R2: 0.888 (training), 0.72 (test)
```

...But the optimal are once again very different!

In [47]:
```
print(f'Best results with: {gbm_cv.best_params_}')
```

```
Best results with: {'max_depth': 2, 'n_estimators': 100}
```

# Gradient Boosted Trees

**GBTs use multiple trees, like RFs, but operate very differently**

Random Forests need to be deep in order to be expressive

- Since all trees trying to learn the same input-output relation
- Depth is needed to capture complexity

Gradient Boosted Trees have an innate mechanism (summation) to:

- Counter overfitting
- and improve expressivity

As a result, individual trees in GBT can be much shallower

- Typically shallow trees tend to work better

**What works best between RFs and GBTs is application-dependent**

...But both are among the best ML models for tabular datasets!