

## 6. Predictions and Machine Learning

- We have seen in chapter 2 that regressions are a powerful tool to approximate the Conditional Expectation Function  $E[Y_i|X_i]$
- But: In a regression we impose assumptions on the functional form of the CEF
- For instance: When we estimate a model

$$Y_i = \alpha + \beta_1 \cdot X_{i1} + \beta_2 \cdot X_{i2} + \varepsilon_i$$

we approximate the CEF with a linear function

- We can estimate much more complex functions and can for instance add interaction terms, quadratic terms etc...
  - ... but this will always be assumptions that we impose
  - ... & we cannot be sure about the true functional form of the CEF
- Here: Use more general machine learning techniques to estimate the CEF relaxing assumptions about its functional form

## 6.1. Key Concepts

### Regression and Classification

- In „machine learning speak“ people distinguish between regression and classification
- In this terminology:
  - Regression means solving a prediction problem where the dependent variable is a continuous variable (a quantity like age, income, job satisfaction,...)
  - Classification means solving a prediction problem where the dependent variable is a discrete variable (dummy variable, “label“ , “class“ such as employee turnover (yes/no), ...)
- Here we will only cover regression problems
- Classification methods are often very similar (and sometimes you can also apply a regression method for binary classification problems)

## Recall: Basic Properties of the CEF

### Result: CEF Prediction Property

Let  $m(X_i)$  be any function of  $X_i$ . The CEF solves

$$E[Y_i|X_i] = \arg \min_{m(X_i)} E[(Y_i - m(X_i))^2]$$

so it is the best predictor of  $Y_i$  given  $X_i$  in the sense that it solves the minimum mean square error (MMSE) prediction problem.

### Result: CEF Decomposition Property

We can decompose  $Y_i$  such that  $Y_i = E[Y_i|X_i] + \varepsilon_i$

(i) where  $\varepsilon_i$  is mean independent of  $X_i$  that is  $E[\varepsilon_i|X_i] = 0$

(ii) and therefore  $\varepsilon_i$  is uncorrelated with any function of  $X_i$

# Supervised Learning

- The part of ML we are interested in is **supervised learning** as the task of learning a function  $\hat{f}(X)$  (“training an algorithm”) that maps an input  $X$  to an output  $Y$  based on a sample
- The learning method learns from a training sample consisting of a set of input-output observations
- Hence, we have (as before) a data set  $D$  with  $N$  observations and  $M$  explanatory variables

$$(y_1, x_{11}, x_{12}, x_{13}, \dots, x_{1M}), \\ (y_2, x_{21}, x_{22}, x_{23}, \dots, x_{2M}), \dots$$

- In ML the vector  $x_i$  of explanatory variables is called the **feature vector** and the matrix  $X$  of the feature vectors of all observations the **feature matrix**
- We want to estimate a function  $\hat{f}_D(X)$  that approximates the CEF

## Parametric and Non-Parametric Approaches

- An OLS Regression is a specific **parametric** ML algorithm
  - Easy to fit as only few parameters need to be estimated
  - Is the best predictor when the underlying function is indeed linear
  - Easy to interpret
- But:
  - Makes strong assumptions on functional form
  - May perform poorly in prediction task when underlying CEF is non-linear and when the number of independent variables is large
- Other **non-parametric** Algorithms may then be more flexible:
  - Do not assume a specific functional form
  - Are thus more flexible to adapt to complex forms of the CEF
  - But are harder to interpret and may perform worse on small data sets

# A simple Algorithm: k-Nearest Neighbor Regression

## Key idea:

- For a given value  $x_i$  we could approximate the CEF  $E[Y_i|X_i = x_i]$  by computing the average of  $Y_i$  across observations with  $X_i = x_i$

**Problem:** We might have very few or no other observations with  $X_i = x_i$

- Instead, use the average value of  $Y_i$  of the  $k$  nearest neighbors of  $x_i$ :

$$\hat{f}_D(x_0) = \frac{1}{K} \sum_{j \in N_k(x_0)} y_j$$

where  $N_k(x)$  is a neighborhood containing the indices of the  $K$  closest  $x$  values in the training data

- Closest neighbors for instance those with smallest Euclidean distance

$$\sqrt{\sum_{j=1}^M (x_{0j} - x_{1j})^2}$$

## k-Nearest Neighbor Regression: Standardization

### Note:

- When the  $x_i$  vector consists of multiple variables measured in different units the measured distance will depend on the unit of measurement
  - If there is, for instance, a wage variable the chosen currency unit will affect the assignment of neighbors
  - In other words: variables where there are large nominal distances will have stronger effects on “who is an observation’s neighbor”

### Hence:

- When the  $x_i$  vector consists of multiple variables measured in different units, then it is useful to standardize all independent variables/features

$$x_{ij}^{std} = \frac{x_{ij} - \bar{x}_j}{std(x_j)}$$

where  $\bar{x}_j$  is the mean of the variable and  $std(x_j)$  its standard deviation

# The Loss Function and Mean Squared Error

An important concept in ML is the **loss function**

$$L(Y_i, \hat{f}_D(X_i))$$

- The loss function measures the “loss” of approximating  $Y_i$  by  $\hat{f}(X_i, D)$
- Common choice: Squared error

$$L(Y_i, \hat{f}_D(X_i)) = (Y_i - \hat{f}_D(X_i))^2$$

## Central Goal:

Find an algorithm  $\hat{f}_D(X_i)$  that minimizes the expected loss

## Recall:

- When estimating a linear regression we did exactly that, being restricted to the class of linear functions
- In ML more general and flexible functional forms are often considered (sometimes at the expense of interpretability)



## The Loss Function and Mean Squared Error

- To assess the quality of the prediction in a regression problem (i.e. with a continuous outcome), we thus often use the **mean squared error (MSE)**
- We use the data set (or part of it) to compute the MSE

$$MSE = \frac{1}{N} \sum_{i=1}^N \left( y_i - \hat{f}_D(x_i) \right)^2$$

**Recall from Chapter 2: The Mean Squared Error is directly connected to  $R^2$**

$$R^2 = 1 - \frac{\frac{1}{N} \sum_{i=1}^N \left( y_i - \hat{f}_D(x_i) \right)^2}{\frac{1}{N} \sum_{i=1}^N (y_i - \bar{y})^2} = 1 - \frac{MSE}{V[y_i]}$$

- Minimizing the MSE corresponds to maximizing the  $R^2$  (as  $V[y_i]$  is given)
- The  $R^2$  has a more accessible interpretation than the MSE: *The share of the variance in the outcome that is predictable based on the features*
- A prediction yielding  $R^2 = 1$  would be perfect, one with  $R^2 = 0$  useless

- Typically, you start a project by inspecting the data, cleaning it, and preparing it for the analysis
- To do so you first pick two subsets of the variables in your data frame:
  - A (one dimensional) array containing the dependent variable to be predicted, which you typically name  $y$
  - A two dimensional array or data frame containing the explanatory variables (features) of each observation, which you typically name  $X$
- For instance, you can define `y=df['JobSatis']`
- To define the feature matrix  $X$ , for instance
  - include all other variables: `X=df.drop(columns='JobSatis')`
  - or include a subset of the variables: `X=df[['age','wage']]`
- Sometimes there are missing values shown in a data frame as NaN
  - rows with NaN can be dropped using `df = df.dropna()`
  - and columns with `df = dfp.dropna(axis=1)`

- Key ML methods are implemented in package scikit-learn
- Starting point: we can also run a linear regression
- Import:

```
from sklearn.linear_model import LinearRegression
```

- Then we can perform a regression with the following code:

```
lreg = LinearRegression().fit(X, y)
```

- $X$  is the (two dimensional) feature matrix
  - $y$  is the (one dimensional) dependent variable to be predicted
- We can then use the results to make predictions  

```
lreg.predict([[50]])
```

 predicts the  $y$  for  $X = 50$
- Note: As Sci-kit Learn rather aims at making predictions, we do not easily get nice regression tables. To obtain these rather use `statsmodels`

- To fit a k-Nearest Neighbor Regression import:  
`from sklearn.neighbors import KNeighborsRegressor`
- Then we can perform a knn regression with the following code:  
`knn=KNeighborsRegressor(n_neighbors=8).fit(X, y)`
  - where `X` is again the feature matrix and `y` the variable to be predicted
  - `n_neighbors` specifies the number of neighbors considered
- We obtain predicted values for each row by `y_pred = knn.predict(X)`
- The method `mean_squared_error` computes the MSE:  
`from sklearn.metrics import mean_squared_error`  
`print(mean_squared_error(y, y_pred))`
- The method `r2_score` computes the  $R^2$ :  
`from sklearn.metrics import r2_score`  
`print(r2_score(y, y_pred))`

- We have seen in section 1 of the course how to “manually” standardize variables (subtracting the mean and dividing by the standard deviation)
- Scikit-Learn provides a convenient way to standardize all variables in  $X$  at the same time:

```
from sklearn.preprocessing import StandardScaler  
XStd = StandardScaler().fit_transform(X)
```

- Then `XStd` is a standardized version of  $X$

- We again work with the Linked Personnel Panel (LPP) data
- Your task now is to train an algorithm that predicts job engagement
- To save some time download a notebook `LPPEngaDataCleaning`
  - This notebook already performs data cleaning (dropping NaN values, generating scales for personality traits etc.)
  - Note: in general be careful in the data cleaning process and check your steps (when you for instance drop all rows where there is missing data for at least some variables you may loose large parts of the data quickly)
- Save a copy as `EngagementPred` & work with this copy
- Recall: you can find a variable description here:  
<https://github.com/dsliwka/EEMP2023/blob/main/Data/VariablesLabelsLPP.pdf>
- And a documentation (unfortunately only in German) here:  
[https://github.com/dsliwka/EEMP2023/blob/main/Data/Lpp\\_CampusFileDocumentation.pdf](https://github.com/dsliwka/EEMP2023/blob/main/Data/Lpp_CampusFileDocumentation.pdf)

- First prepare the data for the prediction task:
  - Define your  $y$  vector  $y = . .$
  - Define your feature matrix  $X = . .$   
Note: Here keep all variables except the outcome variable (engagement)
  - Generate a standardized version  $X_{Std}$  of the feature matrix  $X$  with the `StandardScaler`
- Then perform a k-Nearest Neighbor Regression to predict  $enga$  with  $k = 5$  neighbors using outcome vector  $y$  and feature matrix  $X_{Std}$
- Compute the mean squared error and the  $R^2$  of this prediction
- Save the notebook

# Hyperparameters

- **Parameters** are learned by the algorithm during training (like regression coefficients)
- **Hyperparameters** refers to something that is passed to the algorithm, i.e. is set by the user and determines how the algorithm works
  - The number of neighbors to inspect in a KNN model is a hyperparameter that we have to specify when we create the model
- We can compare the prediction accuracy of a model to determine the best values for the hyperparameters (→ [tuning of hyperparameters](#))

**In order to understand this further we will consider:**

- The problem of **overfitting**
- The importance of **separating training and test data**
- The **bias-variance trade-off**



## 6.2 Overfitting, Training Error, and Test Error

**Recall:** Quality of the prediction often assessed by *mean squared error* (MSE):

$$MSE = \frac{1}{N} \sum_{i=1}^N \left( y_i - \hat{f}_D(x_i) \right)^2$$

**A key problem:**

- When the estimated functional form is very flexible, we will overestimate the predictive power of our algorithm when computing the MSE on the **same data** which we used to train the algorithm

**This is due to *Overfitting*:**

- The function  $\hat{f}_D(X)$  estimated on a sample  $D$  will tend to follow patterns too closely that by chance occur in  $D$  rather than the whole population!
- When overfitting plays a big role, then  $\hat{f}_D(X)$  will be a bad predictor for observations that are not part of the training sample  $D$

## Key element of ML:

- Assess the prediction quality **out-of-sample!**
- To do so: Randomly split the sample  $D$  in two parts  $D_{train}$  and  $D_{test}$ 
  - Use only a subset  $D_{train}$  of the data to train the algorithm
  - Use the remainder  $D_{test}$  to assess the quality of the prediction

## Important distinction: Training error and test error

- *Training error*: Average of loss function over the *training data*  $D_{train}$

$$\overline{err}_{train} = \frac{1}{n_{train}} \sum_{j \in D_{train}} L(y_j, \hat{f}_{D_{train}}(x_j))$$

- *Test error/generalization error*: Average loss when applying  $\hat{f}_{D_{train}}(X)$  for observations that are *not part of the training data*  $D_{test}$

$$\overline{err}_{test} = \frac{1}{n_{test}} \sum_{j \in D_{test}} L(y_j, \hat{f}_{D_{train}}(x_j))$$

- First import:

```
from sklearn.model_selection import train_test_split
```

- The method `train_test_split` conveniently splits the data set into training and test data:

```
X_train, X_test, y_train, y_test  
= train_test_split(X, y, train_size=0.7, random_state=181)
```

### Note:

- The `train_size` parameter determines the share of observations used for the training data set, the remaining observations are the test set
- The method returns four arrays: `X_train` and `X_test` are the feature matrices for the observations in the train and test sets
- `y_train` and `y_test` are the outcome variables in the two data sets
- As the data sets are randomly sampled you will get a different sample each time → fix the sampling with the parameter `random_state=181`

- We can then train the algorithm on our test data:

```
knn = KNeighborsRegressor(n_neighbors=10).fit(X_train,  
                                             y_train)
```

- Print the training  $R^2$ :

```
print(r2_score(y_train, knn.predict(X_train)))
```

- Print the test  $R^2$ :

```
print(r2_score(y_test, knn.predict(X_test)))
```

- It can be convenient to print rounded numbers (here with two digits):

```
print(round(r2_score(y_test, knn.predict(X_test)), 2))
```

## Your Task

## Predicting Engagement

- Open again your notebook `EngagementPred`
- Now split the data in a training and a test set where the test set should comprise 70% of the observations and set the `random_state=181`
- Create a notebook cell in which you
  - Train the knn Algorithm on the training set (again with `n_neighbors=5`)
  - Print the  $R^2$ 
    - on the training set and
    - on the test set
- Interpret your findings
- Save the notebook

## Your Task

## Predicting Engagement

- Copy the cell in which you trained the knn Algorithm and computed the  $R^2$ s
  - In the new cell increase the number of neighbors to 100
  - Run the cell
- Copy the cell once more
  - In the new cell increase the number of neighbors to 300
  - Run the cell
- Compare the training and test  $R^2$  between the three cells
  - How does the training  $R^2$  change?
  - How does the test  $R^2$  change?

## 6.3 Optimal Complexity: Bias and Variance

A model/algorithm is said to be more **complex** or **flexible** when it has more parameters and thus can more easily adapt to patterns in the data

- A multiple linear regression is, for instance, more complex
  - when it has more explanatory variables and more interaction terms
  - or when it includes polynomial terms
- A knn-regression is more flexible when the number of neighbors is smaller
  - it then can adapt more flexibly to patterns in the near neighborhood
- ...

### Note:

- A more flexible model is more likely to be able to come closer to the true CEF
- But: A more flexible model is also more likely to pick up patterns in the data that are simply due to the specific sampling of the training data
- This consideration leads to an important trade-off between **bias** and **variance**

## Bias

- Recall: When  $f(X_0)$  is the true CEF then  $Y_0 = f(X_0) + \epsilon$
- Define: An algorithm  $\hat{f}$ 's **bias** at a specific vector of feature values  $X_0$

$$\text{bias}(\hat{f}(X_0)) = E_D[\hat{f}_D(X_0)] - f(X_0)$$

Key idea:

- Suppose we have different training data sets  $D$
- For each training data set we train a specific algorithm to obtain  $\hat{f}_D(X_0)$
- The bias is the difference between the average prediction made by the algorithm across different samples and the true CEF  $f(X_0)$
- It is preferable to have a low bias!
  - Then there is no systematic difference between our predictions based on certain training data sets and the true CEF
  - An algorithm that exhibits little bias can estimate the CEF precisely when we have an extremely large sample



## Variance

- Consider again the use of different training data sets  $D$
- Again each training data set yields (in general) a different prediction  $\hat{f}_D(X_0)$
- Key question: How much do these estimates vary?
- This estimation **variance** at  $X_0$  is

$$variance\left(\hat{f}(X_0)\right) = E\left[\left(\hat{f}_D(X_0) - E[\hat{f}_D(X_0)]\right)^2\right]$$

- It is the mean squared deviation between predictions made by the same algorithm when trained with different training data sets  $D$   
(Note the relationship to the standard error in regressions!)
- It is preferable to have a low variance!
  - Then the predictions are rather stable. That is, if the algorithm is trained on different training data sets, it makes similar predictions for the same feature vector  $X_0$

## The Bias-Variance Decomposition

- Suppose now that we draw an observation  $(Y_0, X_0)$  from the *test data set*
- Expected mean squared error

$$E \left[ \left( Y_0 - \hat{f}_D(X_0) \right)^2 \right] = E \left[ \left( f(X_0) + \epsilon - \hat{f}_D(X_0) \right)^2 \right]$$

- One can show that this is equal to

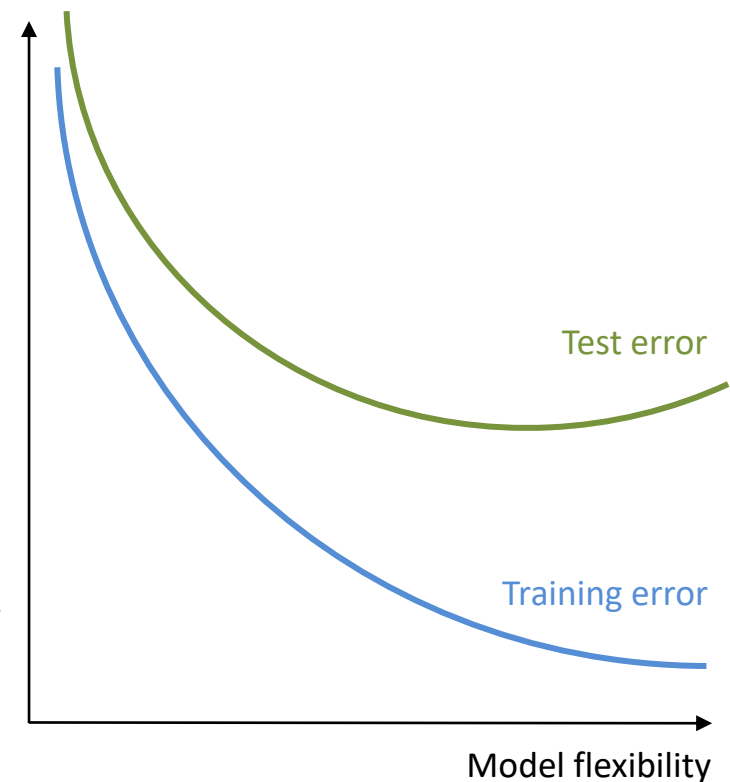
$$V[\epsilon] + \left( E[\hat{f}_D(X_0)] - f(X_0) \right)^2 + E \left[ \left( \hat{f}_D(X_0) - E[\hat{f}_D(X_0)] \right)^2 \right]$$

where

- $V[\epsilon]$  is the *irreducible error*
- $E[\hat{f}_D(X_0)] - f(X_0)$  is the *bias* of the algorithm
- $E \left[ \left( \hat{f}_D(X_0) - E[\hat{f}_D(X_0)] \right)^2 \right]$  is the *variance* of the algorithm

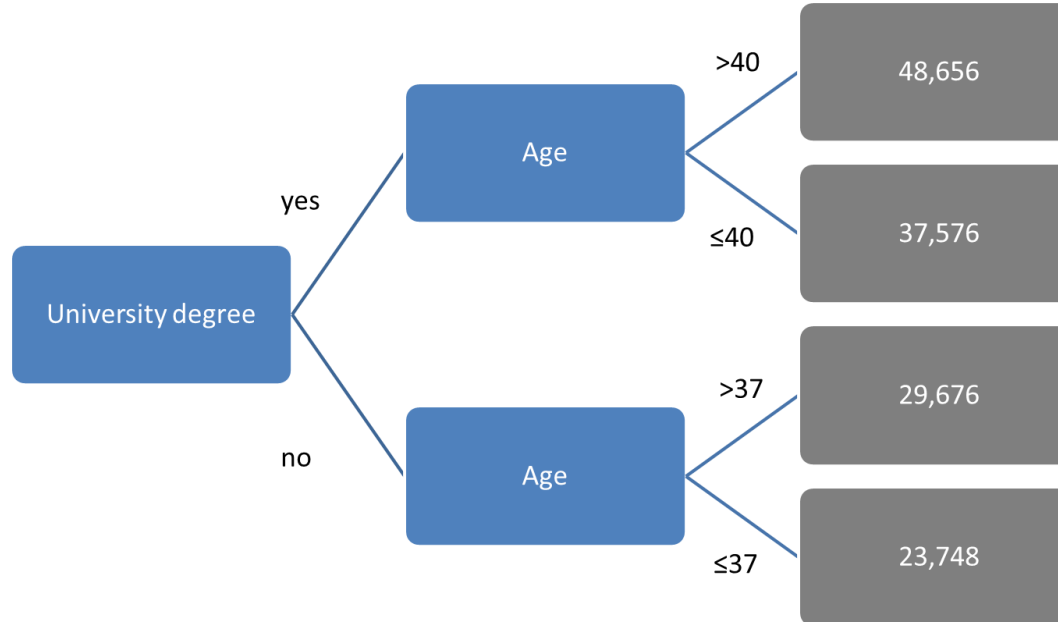
# The Bias-Variance Trade-off

- When increasing the flexibility of a model the training error is reduced  
→ a more flexible model can pick up more detailed patterns
- Initially the test/validation error tends to fall as well
  - as otherwise the model is too simple/inflexible
  - here the **bias** is reduced when adding complexity
- But when the complexity increases further, the test error will tend to increase again
  - Model becomes so flexible that it will map random patterns in training data
  - Hence, the **variance** increases
  - A very complex model will “memorize” the training data & thus **overfit**



## 6.4 Decision Trees and Random Forests

- Decision trees are the building block of powerful ML algorithms
- The key idea of a decision tree is simple:
  - Sequentially partition the data
  - At each step generate two subsets
  - Base the split in each step on a specific condition on one variable
  - Choose the splits by minimizing the expected loss (MSE)



# Decision Trees: How it works

## Trees are constructed as follows: (“Recursive Binary Splitting”)

(Compare James/Witten/Hastie/Tibshirani (2023, pp. 327)

- Begin at the top of the tree & successively split the predictor space
- Each split generates two new branches
- To perform the split
  - select feature  $j$  and cutpoint  $s$  **such that the split into two “regions”** (for instance,  $\text{age} < 45$  and  $\text{age} \geq 45$ ) **leads to the smallest loss (MSE)**
  - repeat the process, looking within each region generated in the previous step for the best predictor and cutpoint minimizing the loss within the respective region
  - ...
- Repeat up to a specific maximal depth of the tree (which you have to set as a hyperparameter)

**Decision Trees: Recursive Binary Splitting**

At each step:

- For each branch look at the set of observations in this branch
- Within this set, define two subsets for feature  $j$  and cutpoint  $s$   
 $R_1(j, s) = \{X_j | X_j < s\}$  and  $R_2(j, s) = \{X_j | X_j \geq s\}$
- Let  $\bar{y}_{j,s}$  be the average value of  $y_i$  across all observations in  $R_1(j, s)$   
Note: This  $\bar{y}_{j,s}$  is the predicted value for this region
- Seek the values of  $j$  and  $s$  that minimize  
$$\sum_{i \in R_1(j, s)} (y_i - \bar{y}_{j,s})^2 + \sum_{i \in R_2(j, s)} (y_i - \bar{y}_{j,s})^2$$
- Repeat the steps until a certain (predetermined) depth of the tree is attained
- The depth of the tree is a hyperparameter to be tuned

Evaluation of Management Practices - Dirk Sliwka - University of Cologne 2023

- To fit a decision tree import:

```
from sklearn.tree import DecisionTreeRegressor
```

- Then we can fit the tree with:

```
dtree = DecisionTreeRegressor(max_depth=3) .  
      fit(X_train, y_train)
```

- where `max_depth` is a hyperparameter that gives the maximal depth of the tree (the number of layers)
  - Note: Restricting `max_depth` prevents overfitting as a tree without a maximum depth will simply map out the whole data set
- Another hyperparameter is `min_samples_leaf` specifying a minimal number of observation that have to be in each leaf

- For a (not too large) tree it is convenient to plot the tree

```
from sklearn.tree import plot_tree
```

- To set the plot size

```
plt.figure(figsize=(20, 10))
```

- Plot the tree and show it

```
plot_tree(dtree, feature_names=X.columns,  
          fontsize =10)  
plt.show()
```

- Note:

- `feature_names` is a parameter with which you specify the names of the features to be displayed
- `X.columns` gives back a list of the variable names in the dataframe containing the features

## Your Task

## Predicting Engagement

- Open again your notebook `EngagementPred`
- Now instead of the Knn regression train a decision tree with a maximum depth of 3 (note: here you do not need to standardize the X matrix)
- Plot the tree and interpret your findings
- Compute the  $R^2$ 
  - on the training set and
  - on the test set
- Save the notebook



# Ensemble Learning and Random Forests

- Decision trees are relatively easy to understand and interpret
- But they tend to suffer from high variance (in particular with high depth):
  - When splitting the data into two parts & fitting a tree to both halves, the results could be quite different (if the sample size is not very large)
  - In other words, deep trees tend to overfit the data
- But it has turned out that training different trees on the same data set and averaging their predictions improves prediction accuracy
- This is the broader idea of **ensemble** methods in ML: Use multiple learning algorithms and average their predictions to obtain a higher accuracy
- An important and powerful such ensemble method is a **Random Forest**:
  - Train many decision trees & use the average prediction
  - Each single tree will have a high variance, but allows for low bias
  - Averaging across trees will reduce variance

# Ensemble Learning and Random Forests

## Important:

- If the different trees are very similar, the improvement will be small as they will make similar predictions
- Hence a *random* forest introduces intentional *randomness* in the construction of each tree (therefore the name!)

## Random forests introduce two forms of randomness:

1. For each tree use a different random sample of the data
  - Apply so-called **bootstrap aggregation (bagging)**
  - That is, for each tree draw a sample of the same size as the original data set (with replacement such that an observation can be drawn twice)
  - Train the tree on this sample
2. For each node in a tree consider only a **randomly chosen subset of the features** as candidates to use for the next split

# Random Forests: Bagging

## Bagging:

(James/Witten/Hastie/Tibshirani (2023, pp. 340))

- Draw  $b = 1, \dots, B$  different “bootstrap samples” of the same size  $N$  as the original sample
  - That is, for each sample draw  $N$  times an observation from the original sample (with replacement – an observation in the original sample can show up several times in the bootstrap sample!)
  - For each sample  $b$  train your algorithm (here: the decision tree) to obtain a predictor  $\hat{f}_b(x)$
- Key idea: Data sets are different, but they are drawn from the same distribution as the original data so share the underlying characteristics
- The bagging prediction is then simply

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^B \hat{f}_b(x)$$

## Random Forest: Using Subsets of Features at each Split

Underlying reasoning for using only a **subset of the features**:

- Improve the prediction by avoiding that the different trees are too similar
- For instance, when some predictors are very important, the splits at the top of each tree will be very similar (even with bagging)
- But when the trees are similar, their predictions will be highly correlated and then less can be learned from aggregating their results
- Note: splits lower down in the tree have less data and thus are more prone to be affected by noise

### The approach:

- At each node of the tree allow only a randomly drawn subset of  $P$  of the  $M$  features to be used for the next split ( $P$  is set as a hyperparameter)
- Idea: “decorrelate” the trees to force them to learn where it is more difficult
- Recommendation: When there are  $M$  features, allow  $P = \sqrt{M}$  features (randomly picked at each node) to be used for the next split

- To fit a decision tree import:

```
from sklearn.ensemble import RandomForestRegressor
```

- Then we can fit the random forest with:

```
forest = RandomForestRegressor(n_estimators=10).  
      fit(X_train, y_train)
```

where `n_estimators` sets the number of trees to be estimated

- Another hyperparameter is `max_features` specifying the number of features randomly drawn at each split that can be used for this split, where
  - you can either set a specific number such as `max_features=4`, or
  - follow the convention to use the square root of  $M$  by specifying `max_features='sqrt'`
- You may also set (& tune by gridsearch) further tree hyperparameters such as `max_depth` or `min_samples_leaf`
- Note that random forests tend to work quite well without much hyperparameter tuning



## Your Task

## Predicting Engagement

- Open again your notebook `EngagementPred`
- Now train a random forest
  - Set `max_features="sqrt"`
  - Set the number of estimators to 500 and the random state to 181
- Compute the test  $R^2$
- Save the notebook

## 6.5 Interpretation: Feature Importance & Partial Dependence Plots

- We use ML methods to make predictions but often we also want to understand & interpret how these predictions work
- For some methods like linear regressions or (shallow) decision trees that is quite easy
- For others (in particular when there are many features) it is more difficult, and the algorithm remains a “black box”
- Important tools to open the black box to some extent:
  - Estimate the importance of specific features for the prediction  
→ feature importance
  - Graphs showing associations  
→ partial dependence plots

## Permutation Feature Importance

- Estimate the mean squared error of your prediction  $MSE(\hat{f}_D, X_{test})$
- For each feature  $j$ : Repeat  $K$  times
  1. Generate feature matrix  $X_{testPerm,jk}$  by randomly permuting feature  $j$ 
    - That is: just randomly shuffle the values, for instance by splitting the data set in pairs and exchanging the values of the pairs
    - Then this feature is completely uninformative
  2. Estimate the error based on the predictions of the permuted data, i.e.  $MSE(\hat{f}_D, X_{testPerm,jk})$
- Calculate permutation feature importance of feature  $j$  as

$$i_j = \frac{1}{K} \sum_{k=1}^K MSE(\hat{f}_D, X_{testPerm,jk}) - MSE(\hat{f}_D, X_{test})$$

- That is: How much does the error increase when I make the feature completely uninformative



- Import feature importance class:

```
from sklearn.inspection import permutation_importance
```

- Then perform estimation of feature importance

```
perm_importance = permutation_importance(model,  
                                         X_test, y_test, n_repeats=30, random_state=0)
```

- For `model` insert the name of the fitted estimator
- `n_repeats` specifies the number of permutations performed

- We obtain the feature importances from

```
perm_importance.importances_mean
```

- We can plot the feature importances in a bar chart:

```
pd.Series(perm_importance.importances_mean,  
          index=X_train.columns).sort_values().plot(kind='barh')
```

- `index=X_train.columns` determines that the bars get the names of the columns in the DataFrame `X_train`

## Your Task

## Predicting Engagement: Feature Importance

- Open again your notebook `EngagementPred`
- Plot the permutation feature importance
- Which features are most important for your prediction

## Partial Dependence Plots

- From the feature importance we learn about the importance for the feature for the prediction
- But we do not learn **how** a feature matters, for instance
  - is there a positive, negative, or non-monotonic association?
  - what is the magnitude of the respective slope of the CEF?
- To inspect this, we can use **partial dependence plots**
  - Pick a feature to be inspected
  - For different potential values of the feature make a prediction for  $y$  holding all other features constant
  - Plot these predictions
- Note: The plots only has a clean ceteris paribus interpretation if the feature in question is uncorrelated with all other features

- Import feature importance class:

```
from sklearn.inspection import PartialDependenceDisplay
```

- Then plot the partial dependence

```
PartialDependenceDisplay.from_estimator(model, X, ['x1', 'x2'])
```

where

- `model` is the fitted estimator
  - `X` is the feature matrix
  - `['x1', 'x2']` is a list of the features for which you want to plot the partial dependence
- You can also plot two-dimensional feature importance  

```
PartialDependenceDisplay.from_estimator(model, X, (('x1', 'x2')))
```

(Note the round brackets () around the tuple)

## Your Task

## Predicting Engagement: Partial Dependence Plots

- Open again your notebook `EngagementPred`
- Plot partial dependence plots for
  - age (Variable `alter`) and
  - health status (Variable `mgesund_allg`)
- Plot also a two-dimensional plot with them both  
`[ ('alter', 'mgesund_allg') ]`

## 6.6 Lasso Regression

- We have seen in the beginning of the lecture that linear regressions can be powerful also for prediction tasks: when the CEF is linear, OLS regressions provide the best estimate
- But a problem we here often face is the choice of explanatory variables
  - When we have data with many predictors (like the LPP): how do we choose among the variables to put on the right hand side?
  - When we think about estimating polynomial functions (as we did when we added quadratic terms), what is the right order of the polynomial
  - When we think about interaction terms: which should we put in?
- We face again a bias-variance trade-off
  - When leaving out variables we may not capture the true CEF well enough
  - When including more variables we have a more complex model are more likely to overfit the training data

## Lasso: How it works

- Regression “punishing” coefficients for deviations from zero (shrinkage)
- In Chapter 2 we have seen that an OLS estimator minimizes

$$E \left[ \left( Y_i - b_0 - \sum_{k=1}^M b_k X_{ik} \right)^2 \right]$$

- A Lasso regression instead minimizes

$$E \left[ \left( Y_i - b_0 - \sum_{k=1}^M b_k X_{ik} \right)^2 \right] + \alpha \cdot \sum_{k=1}^M |b_k|$$

### The key idea is simple:

- $|b_k|$  is the absolute value of the coefficient, i.e. its distance to zero
- Coefficients are thus penalized for deviating from zero and high values of  $\alpha$  **reduce overfitting**
- The hyperparameter  $\alpha$  determines again the trade-off between low bias ( $\alpha = 0$ ) and low variance (large  $\alpha$ )

- To fit a Lasso regression first import:

```
from sklearn.linear_model import Lasso
```

- And then fit:

```
lasso = Lasso(alpha=0.1).fit(X_train, y_train)
```

- You can print the coefficient names and values for instance with

```
print(pd.Series(lasso.coef_, index=X_train.columns))
```



## 6.7 Tuning of Hyperparameters

- As we have seen in the above, we have to find the right balance when determining the complexity of an algorithm
- The more complex/flexible an algorithm is,
  - the lower the bias (we get closer to the true CEF)
  - but the higher the variance, we are more likely to have overfitting
- We should aim at finding values of the hyperparameters that yield the best out-of-sample prediction (lowest MSE) when applied to new data
- Here it is very useful to make **different splits of the data** into training and test data
  - an algorithm performs well when it consistently produces good out-of-sample predictions across different splits
  - This can be done using so-called *cross validation*

## Cross-Validation

**K-fold Cross Validation:** Split multiple times and compute the average test error

- First split data into K roughly equal-sized folds (subsets)  $D_1, D_2, \dots, D_K$
- Repeat for  $k = 1, 2, \dots, K$ :
  - For k-th fold, train the model on data from all other folds, i.e.  $D \setminus D_k$
  - Calculate prediction error on  $D_k$  to obtain  $MSE_k$

$$MSE_k = \frac{1}{|D_k|} \sum_{j \in D_k} L(y_j, \hat{f}_{D \setminus D_k}(x_j))$$

- Compute the average prediction error

$$MSE^{K-fold} = \frac{1}{K} \sum_{k=1}^K MSE_k$$

- Then again pick the specification of an algorithm that minimizes  $MSE^{K-fold}$
- It is common to use  $k=5$  or  $k=10$

## Tuning Hyperparameters with Cross-Validation

- Often we consider models with hyperparameters such as the number of neighbors in KNN, the depth of a tree, or the shrinkage parameter in lasso
- We train an algorithm for a specific hyperparameter  $\alpha$  (or vector of hyperparameters if there are more than one)
- Key question: But what are the “best” values for the hyperparameters?

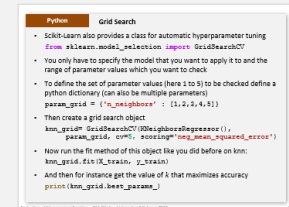
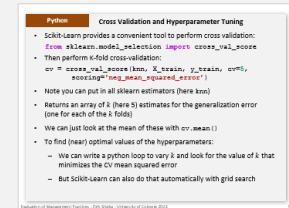
Again: Find the hyperparameters that minimize expected loss **out-of-sample**

- Let  $\hat{f}_D(X, \alpha)$  be the algorithm trained with hyperparameters  $\alpha$
- We can now estimate the K-Fold cross validation (CV) mean squared error

$$MSE^{K-fold}(\hat{f}, \alpha)$$

- And then choose the value of  $\alpha$  that minimizes this
- Note: Some also advocate the random standard error rule: Pick the least complex model that is in a range of one standard error from the best model

- Scikit-Learn provides convenient tools to perform hyperparameter tuning
- You can estimate the cross validation error with `cross_val_score`
  - Performs k-fold cross validation and gives back estimated generalization errors
- Or you directly tune hyperparameters automatically with `GridSearchCV`
  - You can specify the parameters (such as `n_neighbors` in `knn` or `alpha` in `lasso`) and ranges for these parameters to be checked
  - Then `GridSearchCV` finds the combination of parameter values that minimizes the cross validation error
- For some estimators scikit learn has built-in cross-validation capabilities to automatically select the best hyperparameters
  - Then the above steps are performed directly when fitting the estimator



- First import the cross-validation Lasso estimator:

```
from sklearn.linear_model import LassoCV
```

- And then fit the estimator (here using 5-fold cross validation)

```
lasso = LassoCV(cv=5).fit(X_train, y_train)
```

- You can then again access the estimated coefficients and  $R^2$  as in the above

- The optimally tuned value of alpha can be displayed by

```
print("Alpha=", lassoCV.alpha_)
```

## Your Task

## Predicting Engagement

- Open again your notebook `EngagementPred`
- Now try to find the optimal value of alpha for a Lasso regression
- It is convenient to directly use `LassoCV`
- Compute the test  $R^2$  and optimal value of `alpha`
- Print the coefficient names & values
- Save the notebook

# Conclusion

## Aim of the course:

- Familiarize you with different empirical methods
- Let you apply these methods to gain practical skills in using Python to work with data

## Key messages:

- Linear regressions are a powerful tool to approximate conditional expectations with easily interpretable linear functions
- Other machine learning methods may provide more flexible functional forms but sometimes at the expense of interpretability
- Important to distinguish between the different key objectives (i) to make predictions and (ii) to estimate causal effects
  - For the former: Importance to test prediction quality *out-of-sample*
  - For the latter: Importance to understand the role of *identifying assumptions* and their plausibility in different contexts

# Outlook

## Key Developments:

- In recent years extremely useful new tools have been developed to combine machine learning and causal inference (“causal machine learning”)
- Then data science projects can become very powerful to provide actionable insights

## Two very useful areas:

- Machine learning methods to estimate *Conditional Average Treatment Effects*
  - Estimate treatment effects for people/firms with specific characteristics
  - Example: Which incentives work best for people with specific traits?
- *Double Machine Learning*
  - Machine learning methods to systematically tackle omitted variable bias
  - How to find the optimal set of control variables to reduce or eliminate omitted variable/selection bias



# Appendix

# Decision Trees: Recursive Binary Splitting

## At each step:

- For each branch look at the set of observations in this branch
- Within this set, define two subsets for feature  $j$  and cutpoint  $s$

$$R_1(j, s) = \{X | X_j < s\} \text{ and } R_2(j, s) = \{X | X_j \geq s\}$$

- Let  $\hat{y}_{R_k}$  be the average value of  $y_i$  across all observations in  $R_k(j, s)$

Note: This  $\hat{y}_{R_k}$  is the *predicted value* for this region

- Seek the values of  $j$  and  $s$  that minimize

$$\sum_{i:i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

- Repeat the steps until a certain (predetermined) depth of the tree is attained
- The depth of the tree is a hyperparameter to be tuned

- Scikit-Learn provides a convenient tool to perform cross validation:

```
from sklearn.model_selection import cross_val_score
```

- Then perform K-fold cross-validation:

```
cv = cross_val_score(knn, X_train, y_train, cv=5,  
                     scoring='neg_mean_squared_error')
```

- Note you can put in all sklearn estimators (here `knn`)
- Returns an array of  $k$  (here 5) estimates for the generalization error (one for each of the  $k$  folds)
- We can just look at the mean of these with `cv.mean()`
- To find (near) optimal values of the hyperparameters:
  - We can write a python loop to vary  $k$  and look for the value of  $k$  that minimizes the CV mean squared error
  - But Scikit-Learn can also do that automatically with grid search

- Scikit-Learn also provides a class for automatic hyperparameter tuning
- You only have to specify the model that you want to apply it to and the range of parameter values which you want to check
- To define the set of parameter values (here 1 to 5) to be checked define a python dictionary (can also be multiple parameters)

```
param_grid = {'n_neighbors' : [1,2,3,4,5]}
```

- Then create a grid search object

```
knn_grid= GridSearchCV(KNeighborsRegressor(),  
                        param_grid, cv=5, scoring='neg_mean_squared_error')
```

- Now run the fit method of this object like you did before on knn:

```
knn_grid.fit(X_train, y_train)
```

- And then for instance get the value of  $k$  that maximizes accuracy

```
print(knn_grid.best_params_)
```

- The grid search object also has a method to make predictions based on the optimized set of parameters:

```
knn_grid.predict(X_test)
```

- You can use this to compute the test error and the test  $R^2$

```
mean_squared_error(y_test, knn_grid.predict(X_test))
```

```
r2_score(y_test, knn_grid.predict(X_test))
```

Note furthermore:

- The `GridSearchCV` object has an attribute `cv_results_` which allows to access the evaluation metrics in more detail
- For instance, `knn_grid.cv_results_['mean_test_score']` is an array containing the mean squared error (when this is the specified scoring method) for all parameter combinations that were checked