6. Predictions and Machine Learning

- We have seen in chapter 2 that regressions are a powerful tool to approximate the Conditional Expectation Function $E\left[Y_i \middle| X_i\right]$
- 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

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 (a “label“ or a “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

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

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 $Y_i = E\left[Y_i \middle| X_i\right] + \varepsilon_i$

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

(ii) and therefore ε_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

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

- 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
  - 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

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:

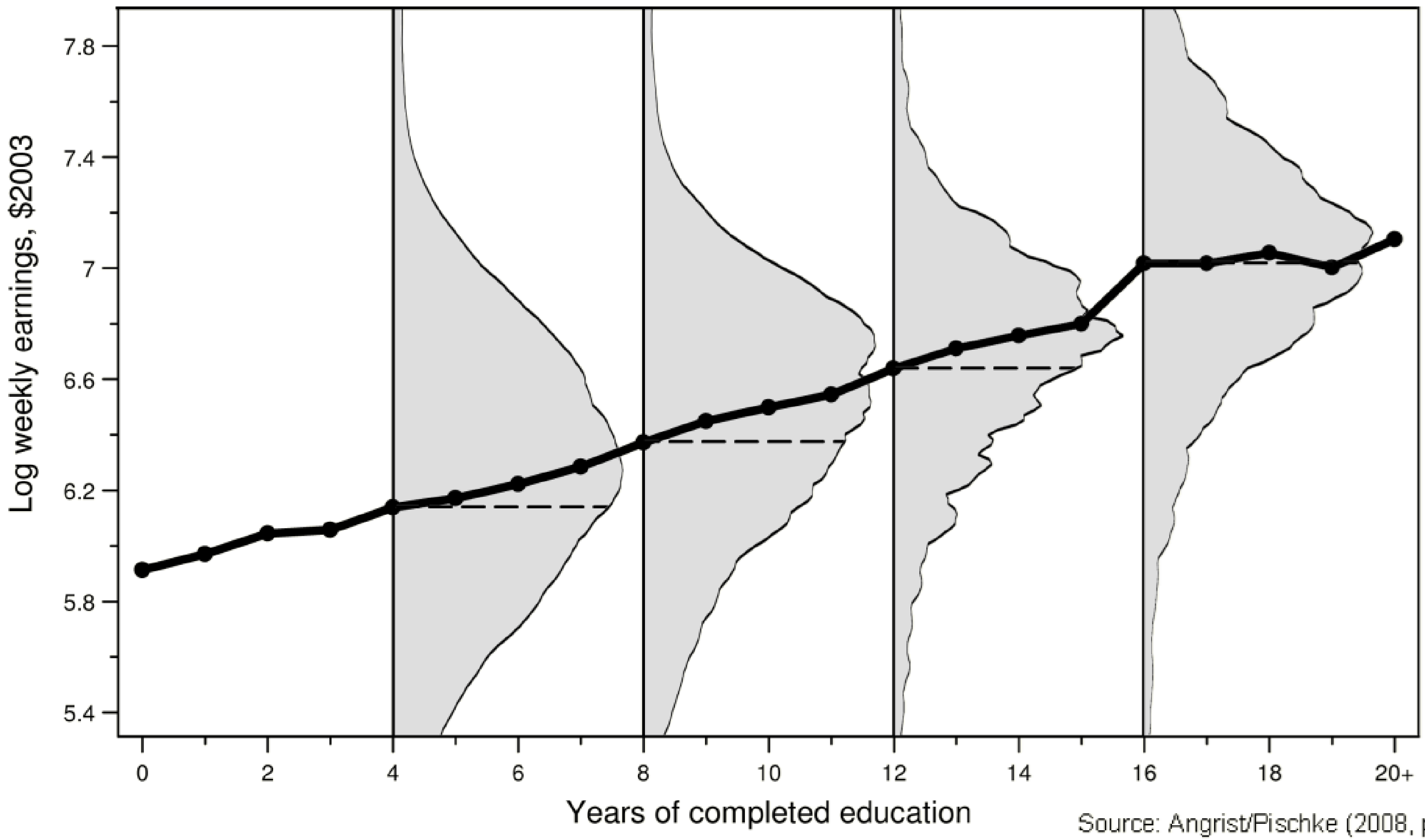
$$\hat{f}_D(x_0) = \frac{1}{K} \sum_{j \in N_k(x_0)} y_j$$

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

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

Recall: The CEF of earnings as a function of years of education



Source: Angrist/Pischke (2008, p. 31)

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
  - Or, if there is a tenure variable measuring the time an employee has been with a firm it will matter whether it is measured in months or years
  - 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

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

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 atre often considered (sometimes at the expense of interpretability)

The Loss Function and Mean Squared Error

Hence:

- When assessing the predictive performance of an algorithm for a regression problem (i.e. one with a continuous outcome variable), we often use the **mean squared error (MSE)**

- That is, we use the data set (or part of it) to compute the squared deviation between actual values of y and predicted values $\hat{y}$

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

- This can, for instance, we used to compare the performance of different algorithms

Mean Squared Error and R^2

Note: The Mean Squared Error is directly connected to R^2

- The coefficient of determination R^2 is the proportion of the variance in the dependent variable that is predictable from the independent variables
- It is given by

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

where $\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$ is the mean of the y_i

- Hence

$$R^2 = 1 - \frac{MSE}{V[y_i]}$$

- As the variance of the dependent variable $V[y_i]$ is given, minimizing the MSE corresponds to maximizing the R^2

Python

Data Cleaning and Preparation

- 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()`

Python

Using scikit-learn (sklearn)

- Key ML methods are implemente 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 a (two dimensional) array containing the explanatory variables (features) of each observation
  - `y` is a (one dimensional) array containing the 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`

Python

k-Nearest Neighbor Regression

- 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

- The method `mean_squared_error` computes the MSE:

```
from sklearn.metrics import mean_squared_error
y_pred = knn.predict(X)
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))
```

Your Task — Predicting performance

- The dataset `data_performance.csv` contain data on employees in a call center of a Chinese travel agency (taken from Bloom et al. (2015))
- Your task is to train an algorithm to predict the performance of these call center workers
- The variable `z_performance` is a standardized performance measure, which is based on employee performance on their main task (e.g. phone calls answered, calls answered per minute, weekly minutes on the phone)
- Import the data from `https://raw.githubusercontent.com/dsliwka/EEMP2022/main/datasets/data_performance.csv'`
- Please first inspect the data and look at the variables (`df.columns`)
- Plot a histogram of the performance variable with `sns.histplot`

Your Task — Predicting performance

- Then prepare the data:
  - Drop all rows with missing values
  - Define your y vector `y=..`
  - Define your feature matrix `X=..` omitting the dependent variable as well as the variable `personid`
- Then perform a k-Nearest Neighbor Regression to predict `z_performance` with $k = 5$ neighbors
- Compute the mean squared error and the R^2 of this prediction
- Note: You can also compute the R^2 "manually" by computing the variance of the dependent variable y with `y.var()`
- Save the notebook as `PerformancePrediction`

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
- 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

Overfitting, Training Error, and Test Error

Recall: Quality of the prediction often assesed 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

Overfitting, Training Error, and Test Error

Key element of ML:

- Assess the prediction quality **out-of-sample**!
- In order to do so: Use only a subset of the data to train the algorithm
- Use the remainder 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}

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

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

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

Python

Splitting Training and Test Data

- 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`

Python

Training and Test Error

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

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

- Recall: The method `mean_squared_error` computes the MSE:

```
 from sklearn.metrics import mean_squared_error
```

- Print the training mean squared error:

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

- Print the test mean squared error:

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

Your Task   Predicting performance

- Open again your notebook `PerformancePrediction`
- 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`
- Train the knn Algorithm on the training set
- Compute the mean squared error and R^2
  - on the training set and
  - on the test set
- Interpret your findings. Did we obtain a good prediction?
- Save the notebook

Python

Standardizing Variables

- We have seen in section 1 of the course how to “manually“ standardize variables (substracting the mean and dividing by the standard deviation)
- Scikit-Lean provides a convenient way to standardize all variables in X

```
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler()
scaler.fit(X_train)
X_trainS=scaler.transform(X_train)
X_testS=scaler.transform(X_test)
```

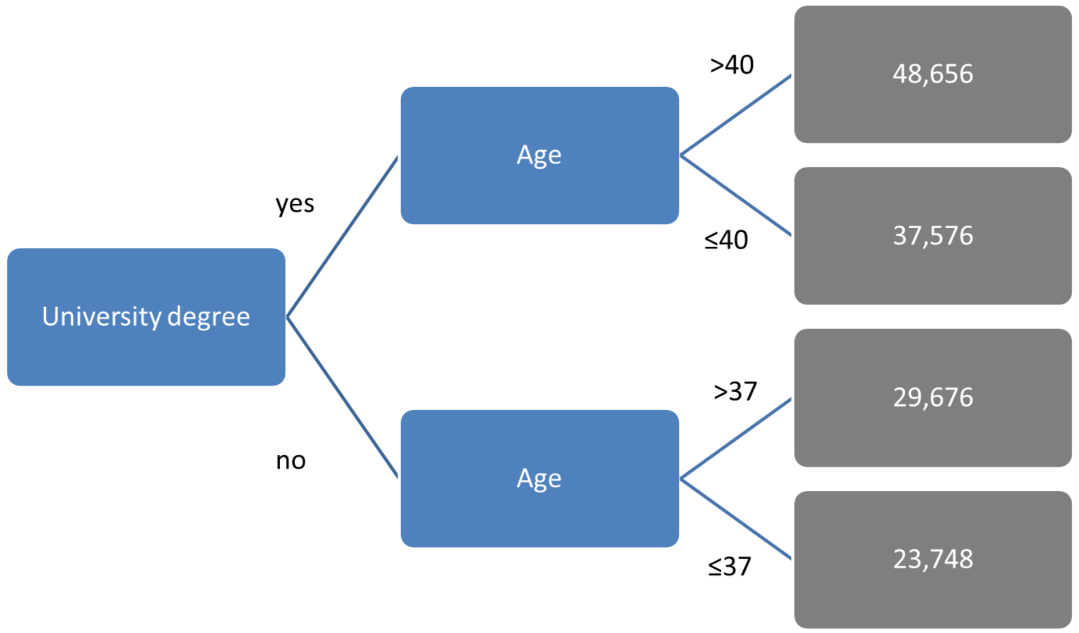
- Then `X_trainS` and `X_testS` are standardized versions of `X_train` and `X_test`
- The standardization is done with the mean and standard deviation of the respective variables in the training set

Your Task Predicting performance

- Open the `PerformancePrediction` notebook
- Now standardize the variables in the feature matrix X with the `StandardScaler`
- Then perform a k-Nearest Neighbor Regression to predict `z_performance` with $k = 5$ neighbors training the algorithm on the standardized data
- Again, compute the test mean squared error and the R^2 of this prediction
- Save the notebook

Decision Trees

- 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 mimizing the expected loss (MSE)



Decision Trees: How it works

Top-down, greedy approach known as *recursive binary splitting*

(Compare James/Witten/Hastie/Tibshirani (2022, pp. 327)

- Begin at the top of the tree & successively split the predictor space
- Each split generates two new branches
- Called "greedy" because at each step, the best split is made at that particular step (rather than looking ahead)
- To perform recursive binary splitting,
  - select a feature j and cutpoint s such that splitting the predictor space into the "regions" $\{X|X_j < s\}$ and $\{X|X_j \geq s\}$ leads to the smallest loss
  - 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
  - …

Decision Trees: How it works

More formally, 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

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

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

Python

Decision Trees

- 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

Python

Decision Trees

- 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 performance

- Open again your notebook `PerformancePrediction` and save it under a different name `PerformancePredictionTree`
- 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 interpet your findings
- Compute the mean squared error and R^2
  - on the training set and
  - on the test set
- Save the notebook

Optimizing Prediction Quality

- But one can go beyond merely assessing the quality of an algorithm by measuring the accuracy of the prediction out-of-sample
- We should aim at finding a specification (i.e. a version of the algorithm) that yields the best out-of-sample prediction
- That is, for instance, find the specification that yields the lowest MSE when applied to new data
- To do so, it is useful to consider **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) $D_1, D_2, \ldots D_K$
- Repeat for $k = 1, 2, \ldots, K$:
  - For k-th fold, train the model on data from all other folds, i.e. $D \backslash D_k$
  - Calculate prediction error on D_{k} to obtain MSE_k

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

- 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

- Often, we consider models with hyperparameters α such as the number of neighbors in KNN regression or the depth of a tree
- We train an algorithm for a specific hyperparameter α (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 α
- 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 α 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

Python

Cross Validation and Hyperparameter Tuning

- 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')
```

- 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

Python

Grid Search

- Scikit-Learn also provides a class for automatic hyperparameter tuning

```
from sklearn.model_selection import GridSearchCV
```

- 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_)
```

Python

Grid Search

- 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 R²

```
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

Your Task — Predicting performance

- Open again your notebook `PerformancePrediction`
- Now try to find the optimal number of neighbors in the Knn regression (note: again use the standardized X matrix)
- Perform a grid search defining a parameter grid
  `param_grid={'n_neighbors': np.arange(2,20)}`
  - Note: `np.arange(3,20)` returns an array of values between 3 and 20 with steps of 1
  - (You could also specify larger steps with a third parameter such as `np.arange(2,20,2)`)
- What is the optimal number of neighbors?
- Compute the test mean squared error and R^2
- Save the notebook

Model Complexity/Flexibility

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 neighboorhood
- A decision tree is more flexible when it has higher depth…

Note:

- A more flexible model is more likely to be able to come close 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

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

- 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

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

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

- It is the mean squared deviation between predictions made by the same algorithm when trained with different training data sets D
- It is preferable to have a low variance!
  - Then the predictions are rather consistent. 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

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

where

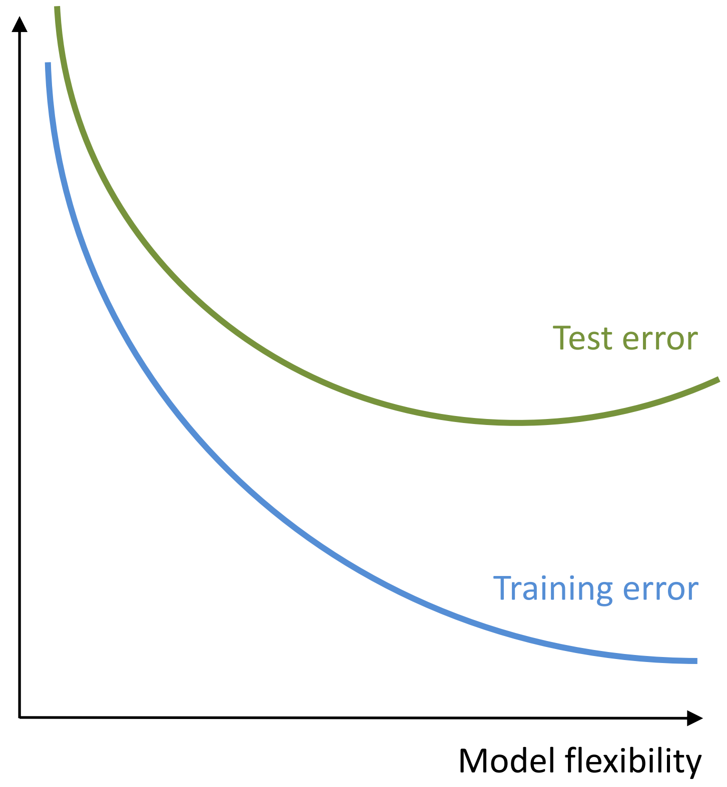
- $V[\epsilon]$ is the *irreducible error*
- $E\left[\hat{f}_D(X_0)\right] - f(X_0)$ is the *bias* of the algorithm
- $E\left[\hat{f}_D(X_0) - E\left[\hat{f}_D(X_0)\right]\right]$ is the *variance* of the algorithm

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
  – Here the **variance** increases
  – Model becomes so flexible that it will map random patterns in training data
  – A very complex model will "memorize" the training data & thus **overfit**



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 training data into two parts & fitting a decision tree to both halves, the results could be quite different
  - 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

Bootstrap: Example

Original Sample:			**Bootstrap Sample:**	
1	Anna		3	Peter
2	Mehmet		10	Rosa
3	Peter		9	Herbert
4	Chloé		4	Chloé
5	Marie		1	Anna
6	Huan		8	Pedro
7	Robert		1	Anna
8	Pedro		3	Peter
9	Herbert		6	Huan
10	Rosa		5	Marie

Random Forests: Bagging

Bagging:

(James/Witten/Hastie/Tibshirani (2022, pp. 340))

- Draw $b = 1, .., B$ different "bootstap 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 subset of the features to be used for the next split
- This subset is determined by a random draw
- The idea is to "decorrelate" the trees or to force them to learn where it is more difficult to learn

Random Forest: Using Subsets of Features at each Split

- Subset of features that may be used for a split randomly drawn at each node
- The size of this subset has to be set as a hyperparameter
- If there are M features and this maximal number of features is set to P:
  - At each node the random forest algorithm randomly randomly picks P out of the M features
  - The algorithm then seeks the best split (as usual when estimating a decision tree) but only within this subset of features
- Note:
  - When $P = \mathrm{N}$ then each split can look at all features in the data set and randomness only comes from bootstrapping (“bagging estimator”)
  - When $P = 1$ then each split could only use one randomly picked feature
- Recommendation: When there are M features, allow $\mathrm{P} = \sqrt{M}$ features (randomly picked at each node) to be used for the next split

Python

Random Forest

- 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
  - do a grid search with `GridSearchCV`, or
  - follow the convention to use the square root of M by specifying `max_features='sqrt'`
- You may also set (& tune) the tree hyperparameters such as `max_depth`

Your Task

Predicting performance

- Open again your notebook `PerformancePredictionTree`
- 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 mean squared error and test R^2
- Save the notebook as `PerformancePredictionForest`

Feature Importance

- We use ML methods to make predictions but often we also want to understand 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“
- One important tool to open the black box to some extent: Approaches to estimate the importance of the different features for the prediction
- Different ML algorithms have different methods to assess feature importance
- Here: Show one method that work across all algorithms:
  Permutation feature importance

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

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

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

Python

Permutation Feature Importance

- 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).plot(kind='barh')
```

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

Backup

Validation Set

Recall:

- If we use the test data for both model selection and assessment, we will tend to underestimate the true test error of the best model.
- The same holds with cross validation: When we use the whole data set for cross validation (and hyperparameter tuning) there is a risk of overfitting

To deal with this problem

- It is advisable to leave out a test set that you do not use in cross validation
- Use this only to estimate the generalization error

Note: Useful terminology to distinguish

- *Validation Set*: this data set is used to tune hyperparameters
- *Test Set:* this data set is used to obtain an unbiased estimate for the actual predictive power of the final algorithm

Random Forest: Out-of-Bag Error

- There is a very straightforward way to estimate the test error of a bagged model (without the need to perform cross-validation)
- Note: When a **bootstrap sample** is drawn, not all observations in the main sample are used
  - as it is drawn with replacement some observations are drawn several times
  - thus others are not drawn at all in that sample
- It has been shown that on average about 63% of the original observations show up in a bootstrap sample
- Hence, each observation in the data set will not be used in about 37% of the predictions $\hat{f}_b(x)$ and these can be used to estimate the test error
- The **Out-of-bag MSE** is just the average squared distance between the observed values y_i and the average of the predictions $\hat{f}_b(x_i)$ across the samples b that did not include observation i