Model selection / Parameter tuning

Holdout method with Test/Validation/Train split

Holdout method

- The **holdout method** is the simplest idea for model selection
  - Split in **train (for model training)** and **test (for model evaluation)**
  - **Problem:** Test set becomes part of the model selection if we repeat this over many models! This is an instance of **information leakage/bleeding.** It can lead to overestimating model performance
- The better version of the holdout method (*always do this!*):
  - Split in **train (for model training)** and **validation (for model evaluation)** and **test (for final performance evaluation)**
  - After we selected the best model, we typically retrained on train+validation set

Holdout method – issues

- Sensitive to how we do the train/validation/test split
  - One remedy (that we will use for CA3) is to average over several different random splits
  - `03_knn_hyperparameters_validation_set.ipynb`
  - Also see `CA3_Workflow.pdf`

- After the holiday break, we will learn a more robust method for model selection: *k-fold cross-validation*

Preprocessing and Feature Selection

Building Good Training Sets

see Ch. 04 in book “Python Machine Learning” by Raschka & Mirjalili

Building Good Training Sets – Overview

- **Removing** or **imputing** missing values in the data set
- Handling **categorical** data (“*feature encoding*”)
- Bringing **features** onto the same **scale** (“*feature scaling*”)

- **Selecting meaningful features** (“*feature selection*”)
- Sequential backward selection algorithm for feature selection
- **Assessing feature importance** with random forests

Preprocessing and Feature Selection

Preprocessing data

Building Good Training Sets

see Ch. 04 in book “Python Machine Learning” by Raschka & Mirjalili

Building Good Training Sets – Overview

- **Key factors** that determine **how well** a machine learning algorithm can learn
  - **Quality** of the data
  - **Amount** of **useful information** that data contains

- It is **critical (!!)** that we **examine** and **pre-process** a dataset **before** we feed it to a machine learning algorithm
- In the following, we discuss the **essential pre-processing techniques** for building datasets that are well-prepared for machine learning tasks

Dealing with missing values – Removal

- NaN: Not a Number
- NULL

	A	B	C	D
0	1.00	2.00	3.00	4.00
1	5.00	6.00	nan	8.00
2	10.00	11.00	12.00	nan

`04_missing_values.ipynb`

Dealing with missing values – Removal

- First solution: remove samples or features with NaN values, e.g. (see script for more)

	A	B	C	D
0	1.00	2.00	3.00	4.00
1	5.00	6.00	nan	8.00
2	10.00	11.00	12.00	nan

	A	B
0	1.00	2.00
1	5.00	6.00
2	10.00	11.00

- Problem for small data set or in the case of bad data quality:
  - We might **remove too much** of the data and don't have enough information left to train a good machine learning model

`04_missing_values.ipynb`

Dealing with missing values – Imputation

- Replace missing value with a reasonable value so we can keep samples that maybe only have a small number of missing feature values
- Scikit-learn: `SimpleImputer`: mean, most frequent value, median

	A	B	C	D
0	1.00	2.00	3.00	4.00
1	5.00	6.00	nan	8.00
2	10.00	11.00	12.00	nan

	0	1	2	3
0	1.00	2.00	3.00	4.00
1	5.00	6.00	7.50	8.00
2	10.00	11.00	12.00	6.00

```
SimpleImputer(
    strategy='mean',
)
```

`04_missing_values.ipynb`

Dealing with outliers

- Difficult task since it is very **dataset-specific**
- **Outliers**
  - can result from faulty instruments or errors in the data (→ we may want to remove them from training (and prediction). Requires domain/expert knowledge about the type of data.
    - Example: A weight measurement of an object must be positive

      → **Remove** samples/features **manually** after expert judgement
  - can represent **rare** data points
    1. We might not be interested in having a good performance for rare unimportant events
    2. The opposite can also be the case: we want to perform well on outliers as well because they represent some rare but important events

Dealing with outliers

- Difficult task since it is very **dataset-specific**
- In the case **we are sure** that **we are not interested** in **performing well on outliers**, we may use z-score to remove all outliers that are, for example, 3 sample standard deviations away from the mean. However, note that this method only **works well** for **normal-distributed data**
- **Z-score:** ***(x – mean)/stddev***
  - Where x: feature value; mean: mean of features values; stddev: standard deviation of feature values (values in one column)
  - Can be positive or negative
  - Z-score of 3 indicated very unlikely events

`04_outliers_zscore.ipynb`

Preprocessing and Feature Selection

Preprocessing: Encoding categorical data

Building Good Training Sets

see Ch. 04 in book “Python Machine Learning” by Raschka & Mirjalili

Categorical features

- Features that can assume one value among a finite set of values
- Examples:
  - T-shirt size: {“S”, “M”, “L”, “XL”} (ordinal)
  - Patient is wearing T-shirt?: {“True”, “False”} (binary, nominal)
  - T-shirt colour: {“Red”, “Blue”, “Green”} (nominal)
- **Ordinal features:** Sets that have an intrinsic order (e.g. T-shirt size)
- **Nominal features:** Sets that have no order (e.g. T-shirt colour)

Categorical features

Example data set with **nominal** and **ordinal** categorical features

(and a numerical feature with is also ordinal)

	nominal	ordinal	numeric & ordinal	nominal
	color	**size**	**price**	**class label**
0	green	M	10.100000	class1
1	red	L	13.500000	class2
2	blue	XL	15.300000	class1

```
04_categorical_data_encoding.ipynb
```

Mapping ordinal features

Data with an **inherent order (ordinal data)** can be **converted** to **numerical integer data (A)**. If we don't want to weigh/quantify the order by a number, we can also use **a threshold encoding (B)** approach using multiple new features (e.g. "x > M" + "x > L" each {0,1})

	color	size	price	class label
0	green	M	10.100000	class1
1	red	L	13.500000	class2
2	blue	XL	15.300000	class1

	color	size	price	class label
0	green	1	10.100000	class1
1	red	2	13.500000	class2
2	blue	3	15.300000	class1

(A)

```
size_mapping = {'XL': 3, 'L': 2, 'M': 1}
df['size'] = df['size'].map(size_mapping)
```

04_categorical_data_encoding.ipynb

	color	price	class label	x > M	x > L
0	green	10.100000	class1	0	0
1	red	13.500000	class2	1	0
2	blue	15.300000	class1	1	1

(B)

Encoding class labels

Although class labels are usually not ordinal, all classifiers we learn about in this course do not consider any ordering of class labels. **Common practice: use integers**

	color	size	price	class label
0	green	1	10.100000	class1
1	red	2	13.500000	class2
2	blue	3	15.300000	class1

	color	size	price	class label
0	green	1	10.100000	0
1	red	2	13.500000	1
2	blue	3	15.300000	0

```
class_mapping = {label: i for i, label in enumerate(np.unique(df['class label']))}
df['class label'] = df['class label'].map(class_mapping)
```

04_categorical_data_encoding.ipynb

One-hot encoding for nominal data

- **Idea:** Replace categorical feature with (*k*) possible states into bit-array of size (*k-1)* where each bit represents whether the corresponding category is assumed (***hot***: 1) or not.
- The value of one bit (*binary digit*) $\in \{0,1\}$ (alternatively represented as $\{\text{False}, \text{True}\}$; $\{\text{yes}, \text{no}\}$; $\{+, -\}$; $\{\text{on}, \text{off}\}$).
- Only *(k-1)* features are needed because the last state is implicitly defined by all bits 0.
- In statistics the new features are also known as "**dummy variables**"

(the "hot" bit is framed in green)

	color	size	price	class label
0	green	1	10.100000	0
1	red	2	13.500000	1
2	blue	3	15.300000	0

	size	price	class label	color_green	color_red
0	1	10.100000	0	True	False
1	2	13.500000	1	False	True
2	3	15.300000	0	False	False

blue

04_categorical_data_encoding.ipynb

```
df_dum = pd.get_dummies(df, drop_first=True)
show_dataset(df_dum)
```

Preprocessing and Feature Selection

Preprocessing: Scaling data

Building Good Training Sets

see Ch. 04 in book “Python Machine Learning” by Raschka & Mirjalili

Scaling data

- Two main ideas: **(1)** Scale to the **same range [min, max]** **(2) Standardize**
- Scikit-learn: (1) `MinMaxScaler`, (2) `StandardScaler`
- Min/Max-scaling is sensitive to outliers

(1) $$X_{\text{scaled}} = \frac{X - \min(X)}{\max(X) - \min(X)}$$

(2) $$X_{\text{scaled}} = \frac{(X - \mu)}{\sigma}$$

X_{scaled} are the new, transformed column values (a column-vector)
X is the original values

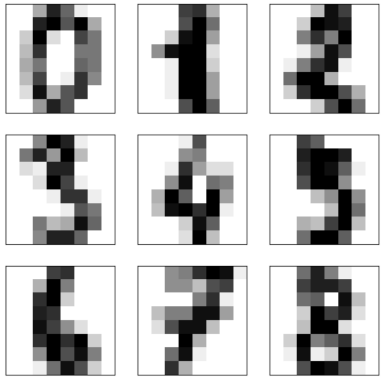
μ vector of the means of each column
σ vector of the standard deviations of each column

} feature-wise scaling

`04_scaling.ipynb`

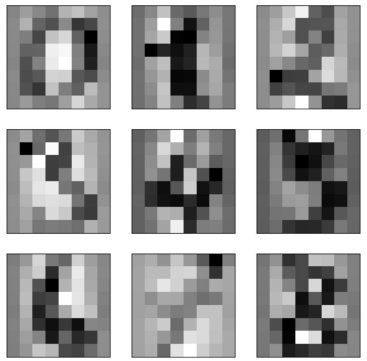
Scaling data

- Scaling each feature separately (as we did so far in the course) is usually a good idea. But depending on data and chosen ML model, we need to be careful.
- An **example** is **image data**: e.g. for the handwritten digits on the right, the intensity of each pixel is a feature (8*8 = 64 features per sample).
  - All pixel values use the same scale/unit and are comparable
  - Pixel values are correlated depending on position! Independent scaling leads to the loss of this information!
  - **In this case:** use mean/stddev of the whole data set (all pixels in all samples) for scaling instead of feature-wise independent scaling
- A classifier that treats each pixel independently (Logistic Regression) may not be affected by feature-wise scaling, while a classifier that can make use of differences between features (e.g. Kernel SVM) could be harmed by feature-wise standardization in this case



good

Feature-wise Standardization

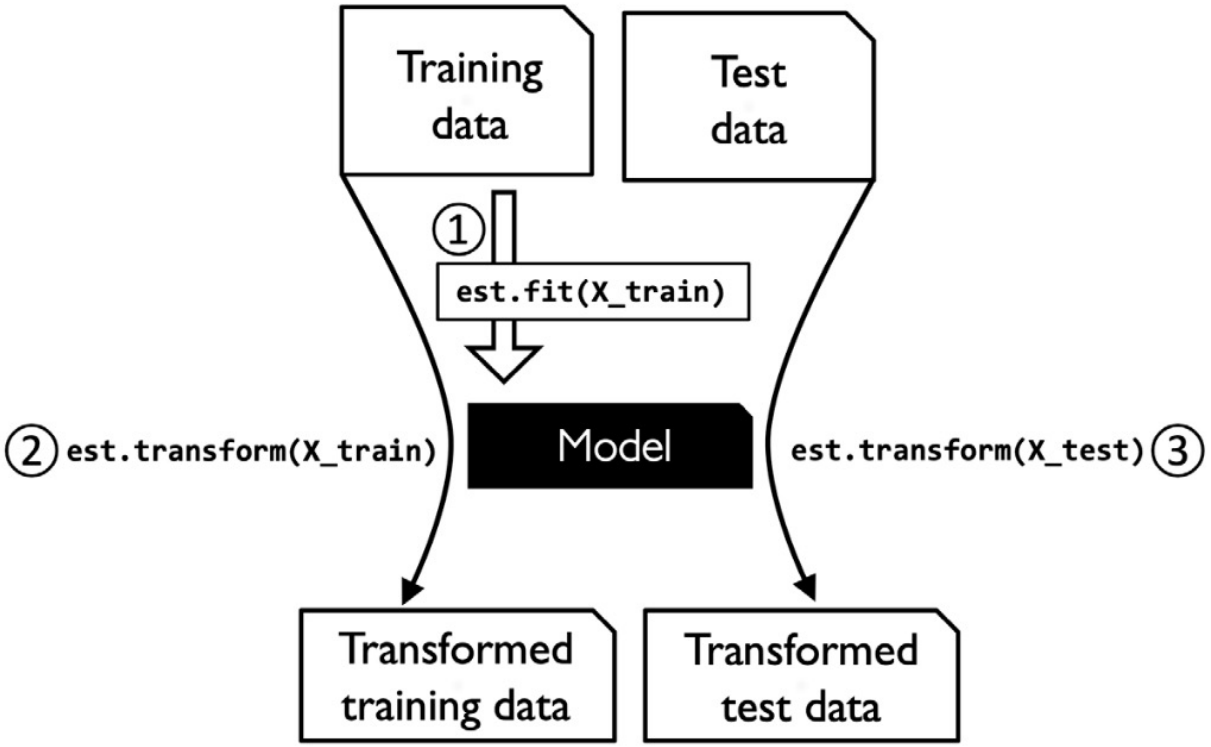


bad

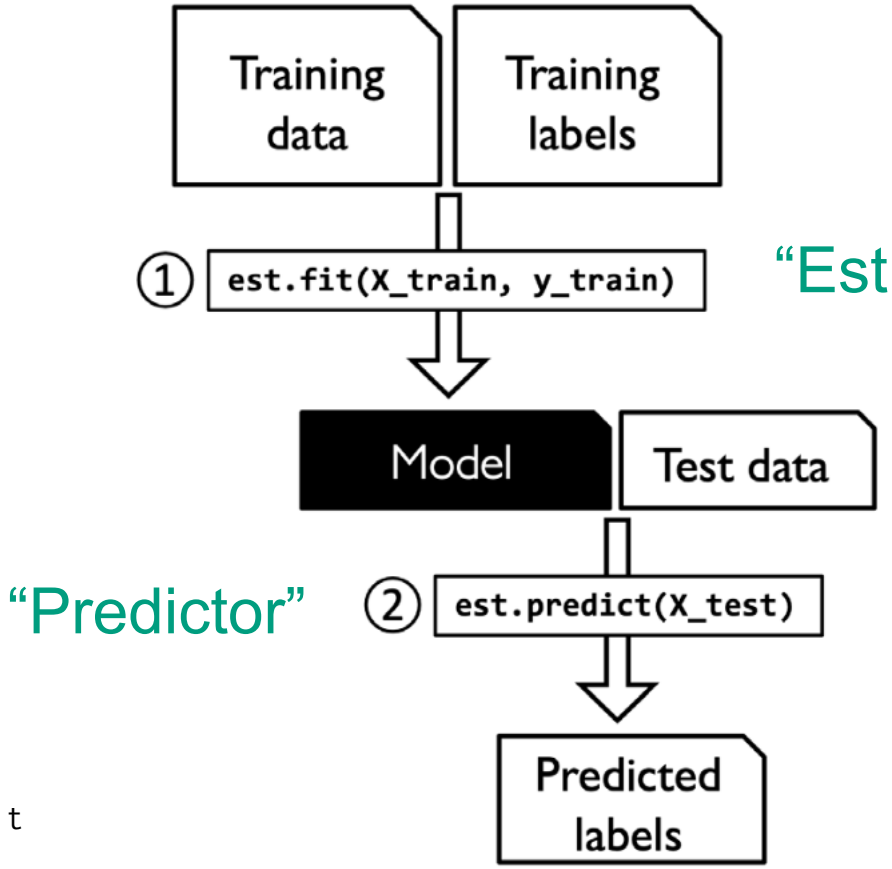
`04_scaling_digits.ipynb`

Transformer API and Estimator/Predictor API (sci-kit learn)

"Transformer"



"Estimator"



"Predictor"

Remark: The alternative interface `est.fit_transform(X_train)` performs both `fit` and `transform` of the training set (1+2) in one function call. Subsequent calls to `transform` have the same effect as if the first call would have been only `fit`.

Figures from Python Machine Learning (Raschka & Mirjalili)

Preprocessing and Feature Selection

Feature selection

Building Good Training Sets

see Ch. 04 in book “Python Machine Learning” by Raschka & Mirjalili

Feature Selection

- If a model performs **much better** on a **training dataset** than on the **test dataset**, this may be **strong indication** of overfitting

- **Overfitting** [recap]
  - model fits the parameters too closely focusing on particular observations in the training dataset
  - model is too complex (too many degrees of freedom) for the given training data
  - model does not generalise well to new data
  - model has a high variance

- **Remedies** for reducing **generalisation** error (and prevent overfitting) [recap]
  - Collect **more** training data (*often not possible!)*
  - Introduce a **penalty** for complexity through **regularisation**
    - e.g. ℓ_1 (LASSO), ℓ_2 (Ridge), ℓ_1 + ℓ_2 (ElasticNet)
  - Choose a **simpler** model with **fewer** parameters
  - Reduce **dimensionality** of data

Feature Selection

- **Feature selection** is a way of **reducing dimensionality** by **removing features**

- Two main ways of **dimensionality reduction:**
  - **Feature selection**

    → select a subset of the original features (this lecture)
  - **Feature extraction** with **compression**

    → derive information from the feature set to construct a new feature subspace

    → e.g. PCA/LDA (next topic in class)

Feature Selection by regularization (recap)

See lectures slides on Overfitting and Regularization

- Add a **regularization** term to the **loss function**

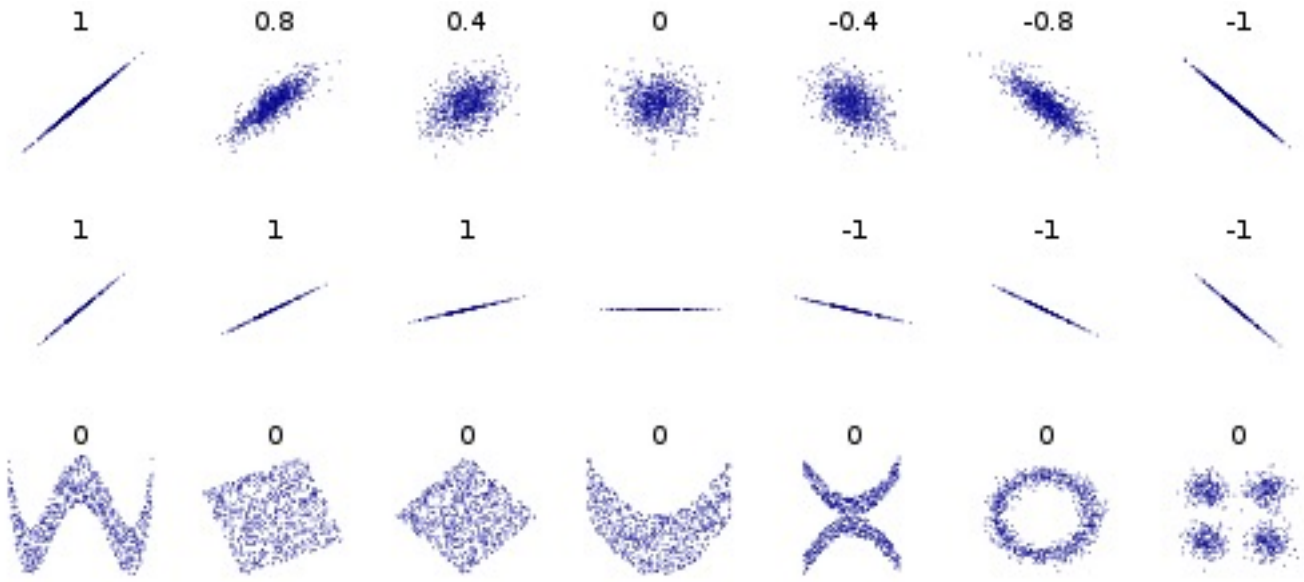
- ℓ_2 **(Ridge) – Regularization:** loss $+\frac{\lambda}{2}||\mathbf{w}||_2^2$ $\qquad ||\mathbf{w}||_2^2 = \sum_{i=1}^{m} w_i^2$
- Promotes **smaller weights** → Features corresponding to smallest weights do not contribute. For prediction, we can also omit features below a given threshold (speed-up)

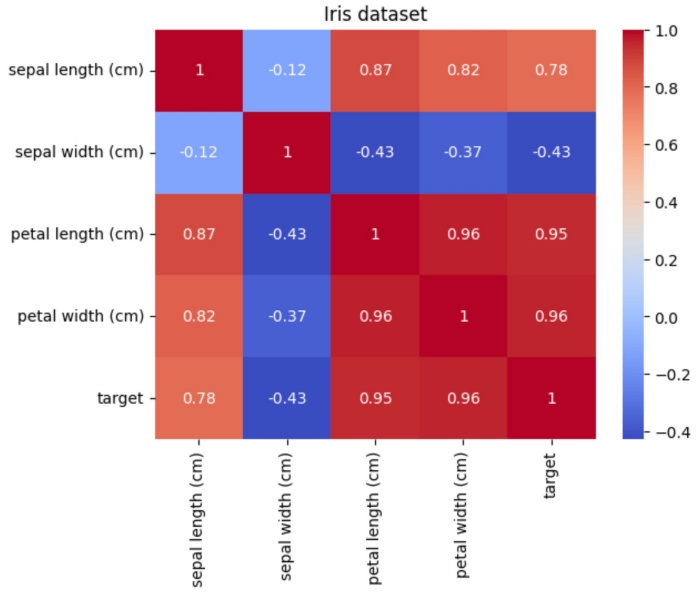
- ℓ_1 **(LASSO) – Regularization:** loss $+\lambda||\mathbf{w}||_1$ $\qquad ||\mathbf{w}||_1 = \sum_{i=1}^{m} |w_i|$
- Promotes **sparsity** (**some weights are zero**) → Features corresponding to zero weights do not contribute. For prediction, we can also omit these features (speed-up)

Detailed discussion of ℓ_2 vs ℓ_1 : *Section 3.4, The Elements of Statistical Learning, Trevor Hastie, Robert Tibshirani, and Jerome Friedman, Springer Science+Business Media, 2009*

Feature Selection based on correlation

- **Strongly correlated** features may contain redundant information, may be left out
- Completely unrelated / uninformative features w.r.t. target may be left out
- A **crude** tool to assess this is looking at the **correlation matrix**
  - **Crude** because this **only** detects **linear** relationships**!**





Correlation distance

```
04_correlationmap.ipynb
```

Sequential feature selection algorithms

- Sequential feature selection algorithms are an alternative way to reduce the complexity of the model and avoid overfitting

  → reducing the number of features using feature selection: reducing an initial d-dimensional feature space to a k-dimensional feature subspace where $k < d$

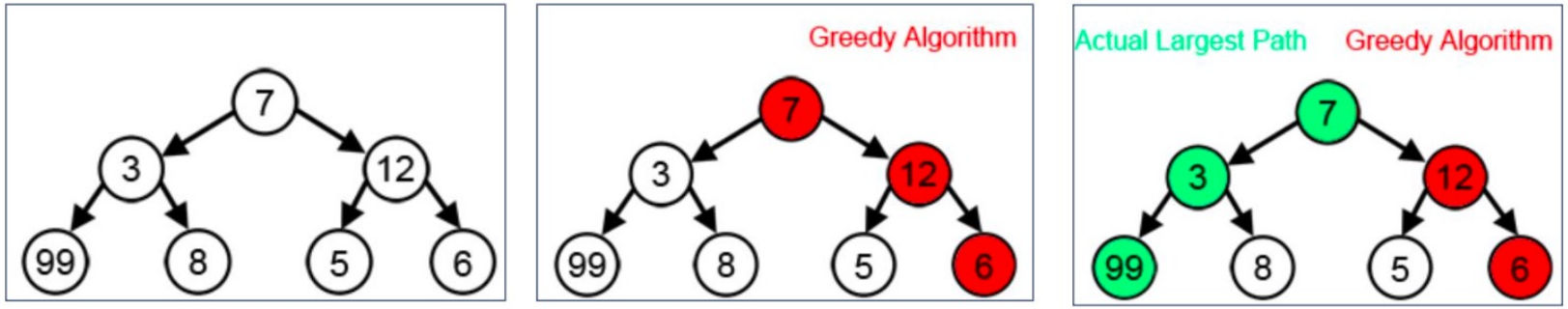
- Especially useful if algorithm doesn't support regularization
- Sequential feature selection is a family of **greedy** search algorithms
- Motivation behind feature selection algorithms
  - automatically select a **subset** of features that are **most relevant** to the problem
  - improve **computational efficiency**
  - Sometimes: **reduce the generalisation error** of the model by removing irrelevant features or noise (models without regularization option which has a similar effect)

Greedy versus exhaustive algorithms

- **Exhaustive** algorithms search for best solution among all possible solutions (relation to global optimization). This can be a **prohibitively expensive** task!
- **Greedy** algorithms use ***heuristics*** to make **locally optimal** choices at **each iteration** of a **combinatorial** search solved as a **sequential** problem
- **Greedy** algorithms generally yield a **suboptimal** solution to the problem, in contrast to exhaustive search algorithms which yield the **optimal** solution (analogously to global/local minima in optimization)
- **Greedy** algorithms allow for a **less complex**, computationally **more efficient** solution

Example

"Find the longest path in a weighted directed graph"

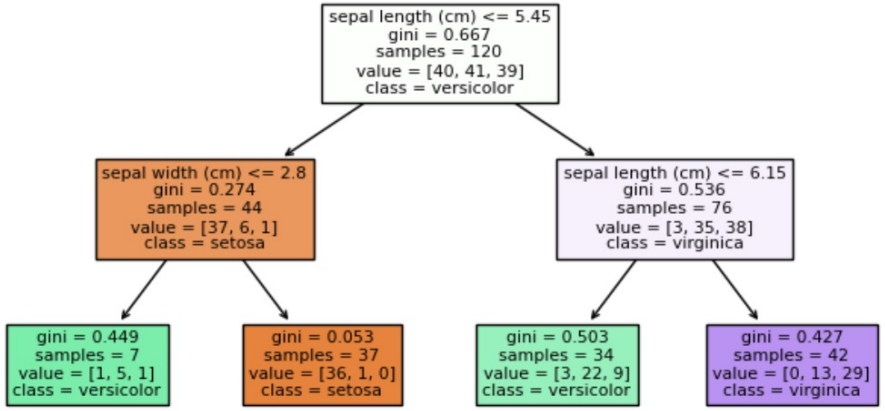


Greedy versus exhaustive algorithms

- **Exhaustive** algorithms search for best solution among all possible solutions (relation to global optimization). This can be a **prohibitively expensive** task!
- **Greedy** algorithms use ***heuristics*** to make **locally optimal** choices at **each iteration** of a **combinatorial** search solved as a **sequential** problem
- **Greedy** algorithms generally yield a **suboptimal** solution to the problem, in contrast to exhaustive search algorithms which yield the **optimal** solution (analogously to global/local minima in optimization)
- **Greedy** algorithms allow for a **less complex**, computationally **more efficient** solution

Example 2

"Find the best splitting strategy in a decision tree"



Decision tree learning is an example of a greedy search algorithm.

We find the best split locally (without looking if this also results in the best split later)

Sequential feature selection algorithms

- A classic sequential feature selection algorithm is **Sequential Backward Selection (SBS)**
- SBS aims to **reduce the dimensionality** of the initial feature subspace with a **minimum decay** in **performance** of the classifier to improve upon computational efficiency
- In certain cases, SBS **can** even **improve** the **predictive performance** of the model (if a model suffers from overfitting):
  - **Example:** k-NN typically suffer from overfitting in high dimensions due to the **curse of dimensionality**. Feature selection can help improve the **predictive performance** by reducing the feature space dimension and therefore susceptibility to overfitting

Sequential Backward Selection (SBS)

Main idea behind SBS algorithm

- SBS **sequentially removes features** from the full feature subset d until the new feature subspace contains the desired number of k features ($k < d$)
- **Which feature** is to be removed at each stage? → need to **define** a loss function *L* that is to be minimized
- The loss function can simply be the **decrease in performance** of the classifier **before** and **after** the **removal of a particular feature**
- The **feature to be removed** at each stage would be the feature that minimizes this loss → in each stage we **eliminate the feature** that **causes the least performance loss** after removal (performance loss negative would mean we increase performance!)

Sequential Forward Selection (SFS)

The SFS algorithm (versus SBS)

- Same concept as SBS but instead of removing features we start with one feature and sequentially add features. We add the feature that improves the model performance the most
- The default “flavour” of the algorithm for sci-kit learn’s `SequentialFeatureSelector` is `direction=forward`

Code example:

```
04_sequential_feature_selection.ipynb
```

Sequential Forward Selection (SFS)

Code example conclusions:

`04_sequential_feature_selection.ipynb`

- Using **less than a 4** of the original 21 features in the cancer dataset, the prediction accuracy on the test set **declined slightly**
- This **may indicate** that those three features **do not** provide less discriminatory information than the **original** dataset
- **The way** the dataset is split into training and test subsets, and how the training dataset is **split further** into a training and validation **subset** may influence the results
- The performance of the K-NN increased a bit by reducing the number of features the size of the dataset was reduced substantially
- Can be useful in real world applications that may involve expensive data collection steps
  - Fewer features means obtaining simpler models, which are easier to interpret

Sequential feature selection in sci-kit learn

- Many more feature selection algorithms available in scikit-learn (not syllabus)

`sklearn.feature_selection`: Feature Selection

The `sklearn.feature_selection` module implements feature selection algorithms. It currently includes univariate filter selection methods and the recursive feature elimination algorithm.

User guide: See the Feature selection section for further details.

`feature_selection.GenericUnivariateSelect`([...])	Univariate feature selector with configurable strategy.
`feature_selection.SelectPercentile`([...])	Select features according to a percentile of the highest scores.
`feature_selection.SelectKBest`([score_func, k])	Select features according to the k highest scores.
`feature_selection.SelectFpr`([score_func, alpha])	Filter: Select the pvalues below alpha based on a FPR test.
`feature_selection.SelectFdr`([score_func, alpha])	Filter: Select the p-values for an estimated false discovery rate.
`feature_selection.SelectFromModel`(estimator, *)	Meta-transformer for selecting features based on importance weights.
`feature_selection.SelectFwe`([score_func, alpha])	Filter: Select the p-values corresponding to Family-wise error rate.
`feature_selection.SequentialFeatureSelector`(...)	Transformer that performs Sequential Feature Selection.
`feature_selection.RFE`(estimator, *[, ...])	Feature ranking with recursive feature elimination.
`feature_selection.RFECV`(estimator, *[, ...])	Recursive feature elimination with cross-validation to select features.
`feature_selection.VarianceThreshold`([threshold])	Feature selector that removes all low-variance features.

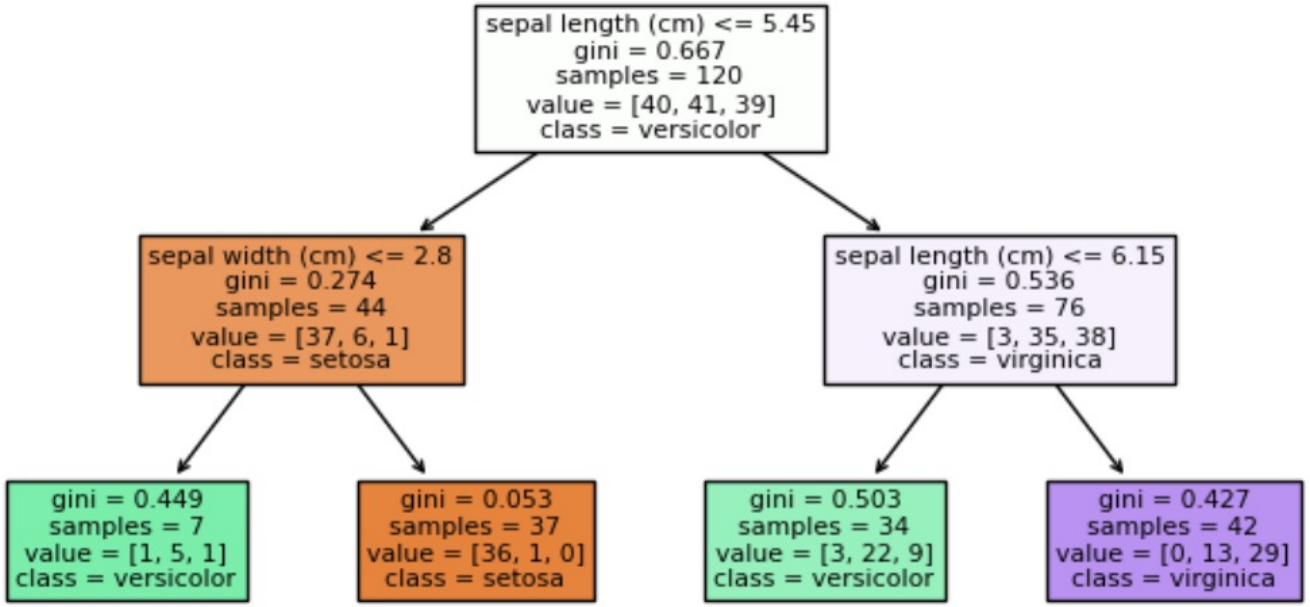
`feature_selection.chi2`(X, y)	Compute chi-squared stats between each non-negative feature and class.
`feature_selection.f_classif`(X, y)	Compute the ANOVA F-value for the provided sample.
`feature_selection.f_regression`(X, y, *[, ...])	Univariate linear regression tests returning F-statistic and p-values.
`feature_selection.r_regression`(X, y, *[, ...])	Compute Pearson's r for each features and the target.
`feature_selection.mutual_info_classif`(X, y, *)	Estimate mutual information for a discrete target variable.
`feature_selection.mutual_info_regression`(X, y, *)	Estimate mutual information for a continuous target variable.

https://scikit-learn.org/stable/modules/classes.html#module-sklearn.feature_selection

https://scikit-learn.org/stable/modules/feature_selection.html#sequential-feature-selection

Feature selection using random forests

- Train a random forest classifier on the dataset
- Measure the **feature importance** as the **averaged impurity decrease (information gain)** computed from **all decision trees** in the forest (*Impurity-based feature importance*)
  - For a given feature, over all trees
    - Check for splits that involve ou
    - Compute impurity decrease
    - Weight by number of samples
  - Average over all trees
  - Repeat for all features
  - Normalize such that feature import



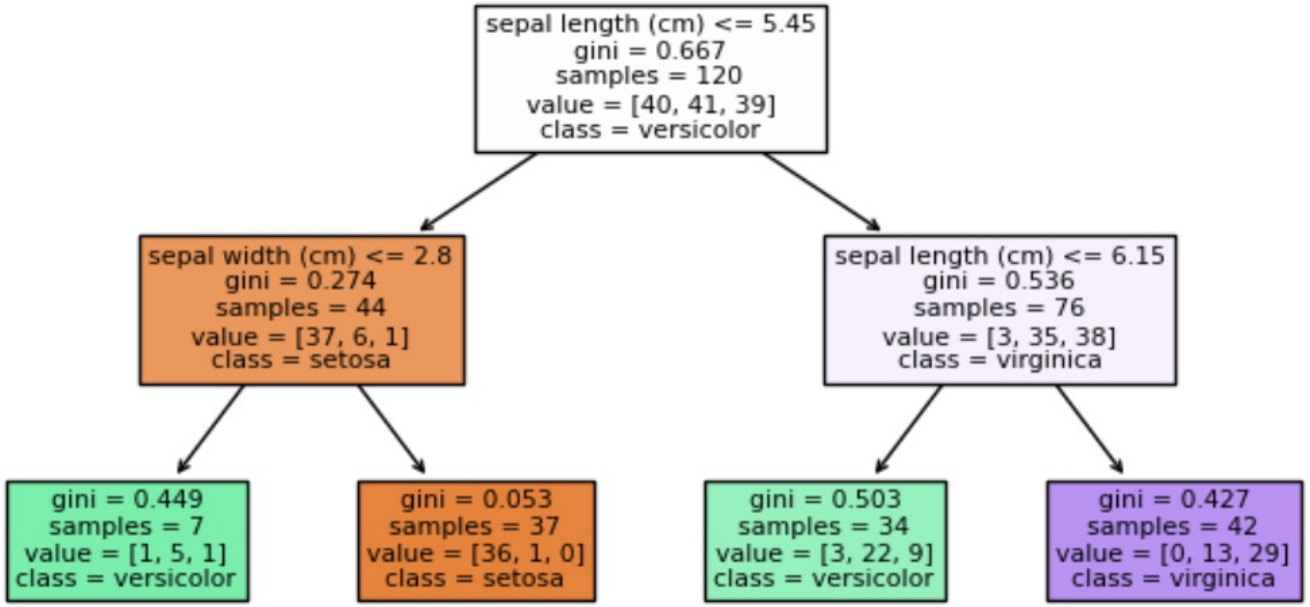
Feature selection using random forests

The random forest classifier in scikit-learn collects the feature importance values automatically:

→ Access the `feature_importances_` attribute after fitting a `RandomForestClassifier`

Code example from sci-kit learn:

https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html



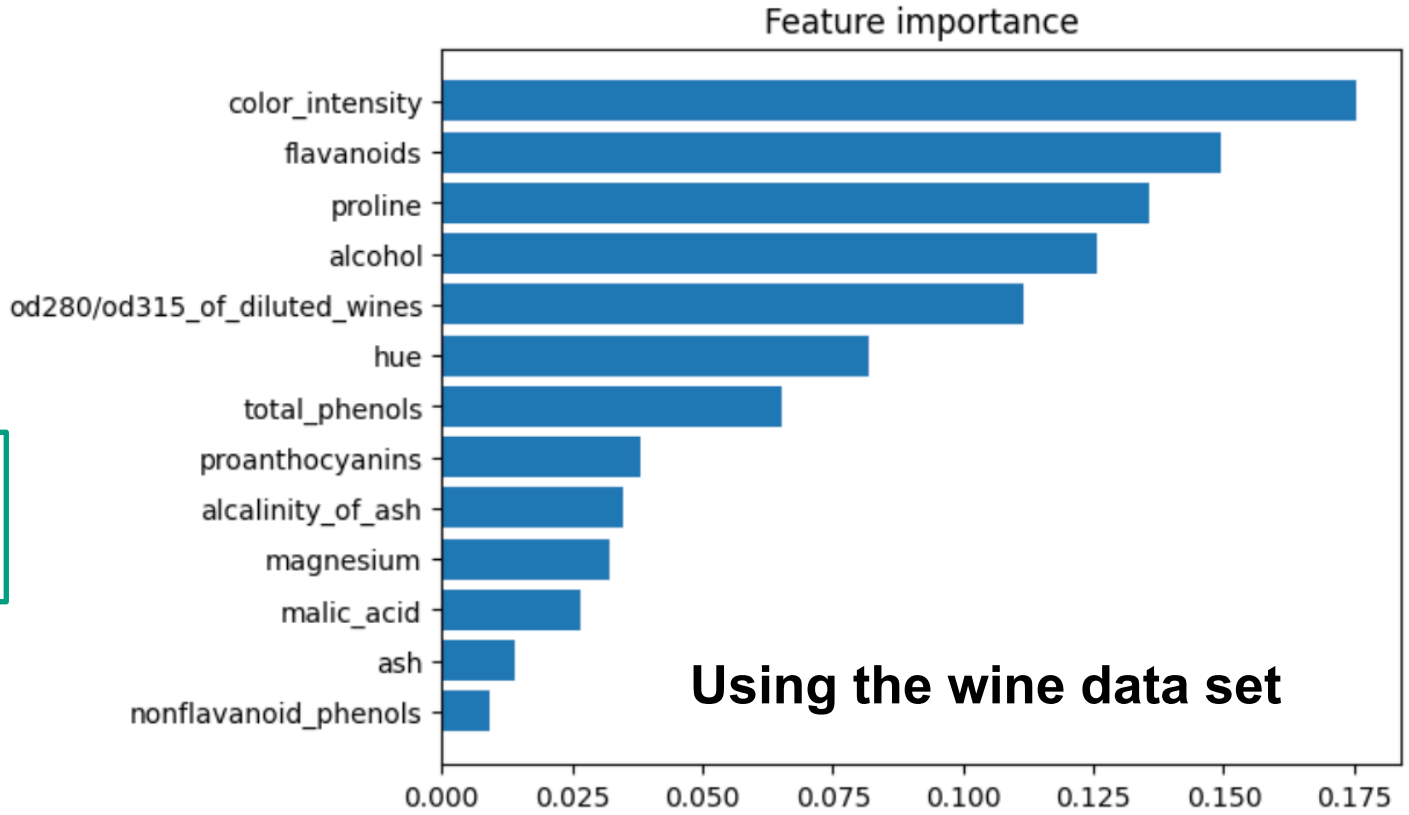
Feature selection using random forests

The random forest classifier in scikit-learn collects the feature importance values automatically:

→ Access the `feature_importances_` attribute after fitting a `RandomForestClassifier`

Code example:

```
04_randomforest_feature_importance.ipynb
```



Feature selection using random forests

- Decision trees **implicitly** assess **feature importance** by finding the best split at a given node
- Remark 1: if **two** or **more** features are **highly correlated** the feature importance is split among the features making each feature possibly appear unimportant. This problem also occurs for categorical data with many unique values (e.g. encoded by one-hot encoding)
- Remark 2: The problem in remark 1 is important when **interpreting feature importance**. When the focus of the analysis is **on predictive performance**, it is **not so important**.
- Remark 3: There is a **more robust method** to assess feature importance in random forests called ***permutation importance*** (not syllabus). It is based on permuting the order of values in one feature column an inspect the out-of-bag error (mean classification error where for each samples, we only use those trees in the forest that weren't trained on that sample).