Selecting the best model

In [ ]:

(1) Overfitting and underfitting (Sven)

Cross-validation framework

In [1]:
```
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing(as_frame=True)
data, target = housing.data, housing.target
```

In [2]:
```
# To simplify future visualization, let’s transform the prices from the 100 (k$) range to the thousand dollars
target *= 100
target.head()
```

Out[2]:
```
0    452.6
1    358.5
2    352.1
3    341.3
4    342.2
Name: MedHouseVal, dtype: float64
```

Training error vs testing error

Trying without train/test split

In [3]:
```
from sklearn.tree import DecisionTreeRegressor

regressor = DecisionTreeRegressor(random_state=0)
regressor.fit(data, target)
```

Out[3]:
```
▾          DecisionTreeRegressor
DecisionTreeRegressor(random_state=0)
```

In [4]:
```
from sklearn.metrics import mean_absolute_error

target_predicted = regressor.predict(data)
score = mean_absolute_error(target, target_predicted)
print(f"On average, our regressor makes an error of {score:.2f} k$")
```

```
On average, our regressor makes an error of 0.00 k$
```

In [5]:
```
from sklearn.model_selection import train_test_split

data_train, data_test, target_train, target_test = train_test_split(
    data, target, random_state=0
)
```

In [6]:
```
regressor.fit(data_train, target_train)
```

Out[6]:
```
▾          DecisionTreeRegressor
DecisionTreeRegressor(random_state=0)
```

In [7]:
```
target_predicted = regressor.predict(data_train)
score = mean_absolute_error(target_train, target_predicted)
print(f"The training error of our model is {score:.2f} k$")
```

```
The training error of our model is 0.00 k$
```

In [8]:
```
target_predicted = regressor.predict(data_test)
score = mean_absolute_error(target_test, target_predicted)
print(f"The testing error of our model is {score:.2f} k$")
```

```
The testing error of our model is 47.28 k$
```

Cross-validation: estimate robustness of a predictive model by repeating the splitting procedure. Gives several training and testing errors and therefore we can estimate how much the model generalization performance varies (as a an approximation to the "new" data that arrive in production).

shuffle split

- make copy of dataset where order of observations are shuffled
- make train/test split
- train model on train, evaluate on test

In [9]:
```
from sklearn.model_selection import cross_validate
from sklearn.model_selection import ShuffleSplit

cv = ShuffleSplit(n_splits=40, test_size=0.3, random_state=0)
# this creates 40 cv splits, where 30% of records are in test data set
cv_results = cross_validate(
    regressor, data, target, cv=cv, scoring="neg_mean_absolute_error"
)
# the ShuffleSplit repeatedly spits out the shuffled train/test splits and lets sklearn train a new model on th
```

In [10]:
```
# just some intuition for what ShuffleSplit does
for train_index, test_index in cv.split(data): # TODO: this is not course material. is it useful anyway?
    print("%s %s" % (train_index, test_index))
```

```
[ 1989   256  7887 ...  9845 10799  2732] [14740 10101 20566 ... 10211  2445 17914]
[ 3364 16548  7361 ...  5154 14875  2751] [ 9959 16306 17662 ... 20549  1212 17484]
[12518 17324  1688 ...  9972  1146   747] [20340 18230 15362 ...  2035  6919  6461]
[15912   555  5108 ... 14722  5851 12988] [ 5313  7640 18918 ...  5008  3312 18057]
[10604  1016  3266 ...   526  6465  9283] [ 7357  4713 10050 ...  1544 11416  6346]
[20356 13340  7263 ... 10572 16502 14054] [10502 14461 10915 ...  8940  5024  3583]
[ 9803  7879 11052 ... 13881  1853 13042] [13182  1320  2362 ... 13276 18258  7491]
[17232 19201 13640 ... 10609  1597 11674] [ 3344   302 20417 ... 12118 15635  5754]
[18558  3153  8256 ...  2442 16057 18358] [ 5997   295  2025 ...  4540 15907  1990]
[10931  6653 19020 ... 19800  6174 10120] [ 6580 14013 12559 ... 14312 13338  7646]
[ 7407 10308 16190 ...  8158 20388  3002] [10768 10864  2580 ... 11360  4314  4105]
[  719 18917  5337 ...    83 10723  3340] [14615  4459  3043 ... 13013 17231    48]
[ 7916 17976 16972 ... 13638 11310 18215] [12227  8164 11921 ...  2912 18550   269]
[14941   378 17191 ... 13560   304 18033] [13109 10203  4911 ... 15997  9224 11830]
[16051  6446 20563 ... 12010  1748 13479] [13112  1550 20295 ...    34 10931 17784]
[13919 14622  6811 ... 15831 10253 17573] [ 7372  8571 11945 ...  9870 18844  8361]
[ 1791 17192 10357 ... 15953 13791  3742] [14731 17561 13050 ...  5145 11292   733]
[ 5853 17054 12736 ...  8291 15284  3873] [11290 15345 13201 ... 16713  3203 15694]
[ 9490  5774 19005 ...  4644 11890  6751] [  545  5914  1194 ... 19346 13214  1783]
[ 2087  8761  5309 ... 18256 16849 11852] [10610 15048  1166 ... 12674  7613 13254]
[ 1353  3845 15756 ... 17553  8977 18026] [ 3342 13283 15765 ... 17635  6815  9131]
[19921  8458 18889 ... 12996 19215  3625] [12369 15433  6410 ... 11275  1788  8574]
[ 9858 19480  1710 ... 18942 12814  8309] [15948 17464 13225 ...  8562  9418  1854]
[ 4099 12535  4149 ... 18020  4205 20338] [ 3979  2931 15403 ... 15676 18914     4]
[10799  3500 10904 ...  9093 10248  6202] [16001 17417 16190 ... 18034 20071  2928]
[ 3245 15245 12179 ... 11924  2313 19602] [15497 20250 20520 ... 14176 13589 13315]
[14964 10997 14474 ... 11342 12264 19262] [15761 11288  1119 ...  7547  3744  9514]
[11498 15182  9381 ...   857 11206  9792] [18657 11556  9115 ...  7267 16544 17702]
[15275   230  3093 ... 14022  7373  7798] [12422 16605  5433 ...  5975 16744 11690]
[16382  4066 16136 ...  6781  9674  1093] [19057  2027  1634 ... 18047 19451  2893]
[12872 11579 11378 ... 18729 15918 16465] [  204  2357 19173 ...  9591  6166 10713]
[20612 11934  2136 ...  8504  7905   230] [17398  9740  2045 ...    19  5071  7451]
[ 1925 11503 18202 ... 19998 15865  2596] [16510 15666  8228 ... 19052 17070 11509]
[18431 19175 11668 ... 16476 13214  1746] [ 2487 17152 17042 ...  2966 13231 10180]
[ 4762 18773  2883 ...  7474  2329 18605] [ 8476  3899   109 ... 11022 10168   894]
[ 5628 15157 16360 ...  6438   746 19740] [ 1918   564  7406 ...  7996  5692 20514]
[20519  5774 12116 ...  8024 20261  7364] [10167  9473   828 ... 13285  6920 12035]
[18416 18319 15482 ... 17311 20589 13162] [10930 15771  1142 ...  7600  2554 13790]
[ 1597 14406  8513 ...  6052 16273 15483] [17854  3190  1959 ... 20537  8135  4888]
[ 8683  9855 13795 ...  6342 10717  3752] [14369  8483  6552 ... 15390 17427  8695]
```

Note: some of these things are covered in the "Overfitting and underfitting section", thus it may not be necessary to cover them again/in detail

What does the `ShuffleSplit` do? parameters

- https://scikit-learn.org/stable/modules/cross_validation.html#random-permutations-cross-validation-a-k-a-shuffle-split
- https://stackoverflow.com/questions/34731421/whats-the-difference-between-kfold-and-shufflesplit-cv
- it creates train and test samples randomly in each iteration. this means that a single observation can be multiple times in the test set.

- `n_splits` : the number of train-test splits
- `test_size` : size of test data, as a fraction of the full data set
- `random_state` : makes the results reproducible b/c it fixes tells the random number generator where to start randomizing

We can pass this input to the cross_validate function below as the `cv` argument.

In [11]:
```
import pandas as pd

cv_results = pd.DataFrame(cv_results)
cv_results.head()
```

Out[11]:

	fit_time	score_time	test_score
0	0.123644	0.002460	-46.909797
1	0.121718	0.002373	-46.421170
2	0.129612	0.002224	-47.411089
3	0.123873	0.002409	-44.319824
4	0.119278	0.002245	-47.607875

Lingo: score vs error

- score = higher values mean better results
- error = lower values mean better results

`scoring` parameter in `cross_validate` expects a function that is a score (why does that matter? does it not just calculate the error/score and return it? does it do something with it?)

so, the mean abs error is an error, but we need a score -- so we take the negative. But to look at the mean absolute error for our purpose, we need to again re-convert the score to an error.

In [12]:
```
cv_results["test_error"] = -cv_results["test_score"]
```

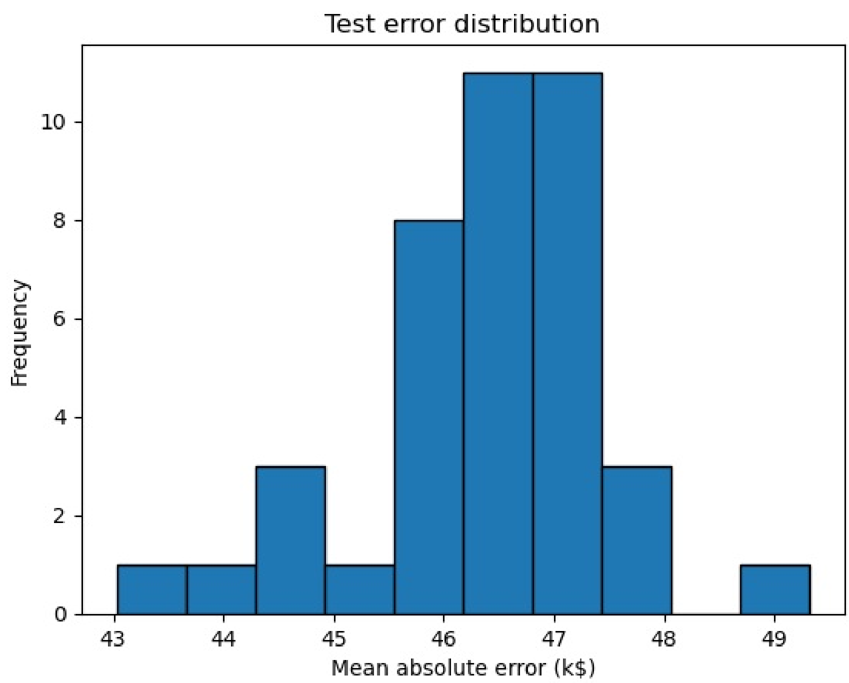
In [13]:
```
cv_results.head()
```

Out[13]:

	fit_time	score_time	test_score	test_error
0	0.123644	0.002460	-46.909797	46.909797
1	0.121718	0.002373	-46.421170	46.421170
2	0.129612	0.002224	-47.411089	47.411089
3	0.123873	0.002409	-44.319824	44.319824
4	0.119278	0.002245	-47.607875	47.607875

In [14]:
```
import matplotlib.pyplot as plt

cv_results["test_error"].plot.hist(bins=10, edgecolor="black")
plt.xlabel("Mean absolute error (k$)")
_ = plt.title("Test error distribution")
```



What does this mean? Call-out?

> We notice that the mean estimate of the testing error obtained by cross-validation is a bit smaller than the natural scale of variation of the target variable. Furthermore, the standard deviation of the cross validation estimate of the testing error is even smaller.

- mean estimate of testing error is 47k
- what is the "natural scale of variation of the target variable"? -- it is just the range of the outcome, from around 50k to 500k
- but then I don't understand 47k is "a bit smaller than the range", which is 500k?

- small std of cross-validated error indicates good out-of-sample performance of the model

In [ ]:

In [ ]:

(2) Validation and learning curves

Comparing train and test errors

See the video/slides: https://inria.github.io/scikit-learn-mooc/overfit/learning_validation_curves_slides.html

Presentation notes

train vs test error

- black training data points
- fit blue prediction function
- orange test data (generalization error). -- fundamental goal of ML: good prediction of y given x on unseen data.
- we can contrast the train and test error to understand better how well it generalizes

train vs test error: increasing complexity

- polynomial: higher degree = higher complexity. first order = linear in x. second-order = quadratic in x. etc.
- compare the train and test error: average distance between the blue prediction line and the respective data points.
- on average, train error < test error b/c it's easier to remember the training data
- degree 2 polynomial: lowers both train and test error
- degree 5 poly: train error decreases further; but it seems it fits some noise in the training data that are not present in the test data (the very right of the figure). thus, test error increases.
- degree 9 poly: gets extreme.
- in sum: get an intuition of how a more complex model impacts train and test error: low degree poly -> train and test error are high. higher complexity -> "sweet spot" test error is minimal. higher complexity -> test error gets higher again. *overfitting*

train vs test error: varying the sample size

- fix the complexity of the model; see how a bigger training set impacts the test error
- few data points -> function varies.
- adding data points -> function gets smoother. test error goes down. intuition: the model can approximate the true data generating process better; noise in the training data has smaller impact on the prediction. (train error also increases; but reason not clear)
- at some point, the train and test error ar the same, suggesting we're close to the optimal model
- going beyond it -> no more changes; "diminishing returns". if computation is costly, this is mostly adding costs without adding value
- *with large enough training data, the degree 9 poly does not overfit anymore*
- general pattern: model complexity *relative* to the training data
- we cannot bring the test error to 0: "Bayes error rate" / "irreducible error" -- arises from the random noise that is in the data generating process. *best we can do*

model families

- in ML, we always have a statistical model, and an unknown data-generating process
- by choosing the statistical model, we want to match the DGP
- so far, we had polynomials for both DGP and the stat model
- we can also choose a different model family. what happens then?
- decision tree: piece-wise linear: predict constant y for a range of x. (blue line)
- they have different inductive biases and different notions of complexity
- inductive bias = "the set of assumptions that the learner uses to predict outputs of given inputs that it has not encountered."
- there is always an inductive bias *unless* we choose the correct model class that matches the DGP
- model families have different notions of complexity: higher degrees in polynomial = higher number of splits in the decision tree. both lead to overfitting.
- regularization = favor smooth, simple functions over complex functions. this reduces overfitting. *change parameters while keeping the model fixed*

take-home messages

- models overfit when number of examples in the training set is too small compared to the complexity of the model
- we can detect overfitting when the testing error is much bigger than the training error (but not necessarily? -- better to "minimize test error"?)
- models underfit if they fail to capture the shape of the training set -> even the training error is large. BUT: it could also be noise in the DGP.

We'll see how this works in practice now.

Overfit-generalization-underfit

We now want to use the cross-validation approach to quantify training and testing errors. This helps us to find out whether our model generalizes, overfits or underfits.

In [15]:
```
# already done above
# housing = fetch_california_housing(as_frame=True)
# data, target = housing.data, housing.target # we just extract the explanatory variables and the outcome
# target *= 100 # rescale the target to thousands of dollars (the original data are in hundreds of thousands of
# # https://inria.github.io/scikit-learn-mooc/python_scripts/datasets_california_housing.html
```

In [16]:
```
# target.head()
```

In [17]:
```
# from sklearn.tree import DecisionTreeRegressor

# regressor = DecisionTreeRegressor()

# DELETE?
```

The decision tree regressor was "introduced" / used (but not explained) in the previous segment.

Overfitting vs underfitting

In order to understand how our model generalizes, we want to compare the testing and the training error. To compute the error on the test set with the `cross_validate` function

In [18]:
```
# already imported above
# from sklearn.model_selection import cross_validate, ShuffleSplit
```

In [19]:
```
cv = ShuffleSplit(n_splits=30, test_size=0.2, random_state=0)
```

In [ ]:
```

```

In [20]:
```
cv_results = cross_validate(
    regressor,
    data,
    target,
    cv=cv,
    scoring="neg_mean_absolute_error",
    return_train_score=True,
    n_jobs=2,
)
```

In [21]:
```
cv_results
```

Out[21]:
```
{'fit_time': array([0.17909646, 0.17803574, 0.17157149, 0.18492937, 0.17681456,
        0.17953753, 0.17122507, 0.18224573, 0.17118764, 0.18390369,
        0.17098308, 0.17956877, 0.16909313, 0.18139696, 0.16651297,
        0.17918992, 0.17719746, 0.1906023 , 0.1698103 , 0.19037414,
        0.16757512, 0.18488216, 0.17270684, 0.18379045, 0.17240763,
        0.18100095, 0.16896176, 0.18912959, 0.18846846, 0.14204884]),
 'score_time': array([0.00231862, 0.00270486, 0.00240159, 0.00259829, 0.00250983,
        0.00254583, 0.00247121, 0.00260639, 0.00265813, 0.00260091,
        0.00238299, 0.00263357, 0.00263333, 0.00255966, 0.00217819,
        0.00261736, 0.00230765, 0.00254941, 0.00308895, 0.00264406,
        0.00235724, 0.00252104, 0.00256324, 0.00272179, 0.00242782,
        0.0026629 , 0.00222826, 0.00317383, 0.00234175, 0.00206113]),
 'test_score': array([-46.92525848, -46.6962173 , -45.06884133, -43.57765528,
        -48.05169307, -44.58310441, -44.42208236, -45.1216984 ,
        -44.96031032, -45.14051914, -47.27880063, -46.76900751,
        -46.07751163, -45.55437984, -47.20800945, -44.38683794,
        -45.99953682, -46.76782049, -45.11137815, -47.4055906 ,
        -43.39771245, -46.00505475, -45.36389753, -46.79342878,
        -46.32596754, -45.71983503, -44.41470058, -46.21278246,
        -45.68218314, -47.66419259]),
 'train_score': array([-1.51747693e-14, -3.29796483e-15, -9.32587341e-15, -1.26376084e-14,
        -8.81293316e-15, -1.25859702e-14, -1.44070802e-14, -3.74549659e-15,
        -1.12123919e-14, -3.45976477e-15, -3.38747118e-15, -1.57935680e-14,
        -1.52126374e-14, -1.00763497e-14, -1.23622043e-14, -1.36772592e-14,
        -1.33811997e-14, -3.24976910e-15, -3.21190103e-15, -3.79369232e-15,
        -1.50542800e-14, -1.21074554e-14, -1.42452802e-14, -3.64566258e-15,
        -3.58369665e-15, -1.52952586e-14, -3.42189670e-15, -1.32193997e-14,
        -1.28544892e-14, -3.09141171e-15])}
```

In [22]:
```
cv_results = pd.DataFrame(cv_results)
```

In [23]:
```
cv_results
```

Out[23]:

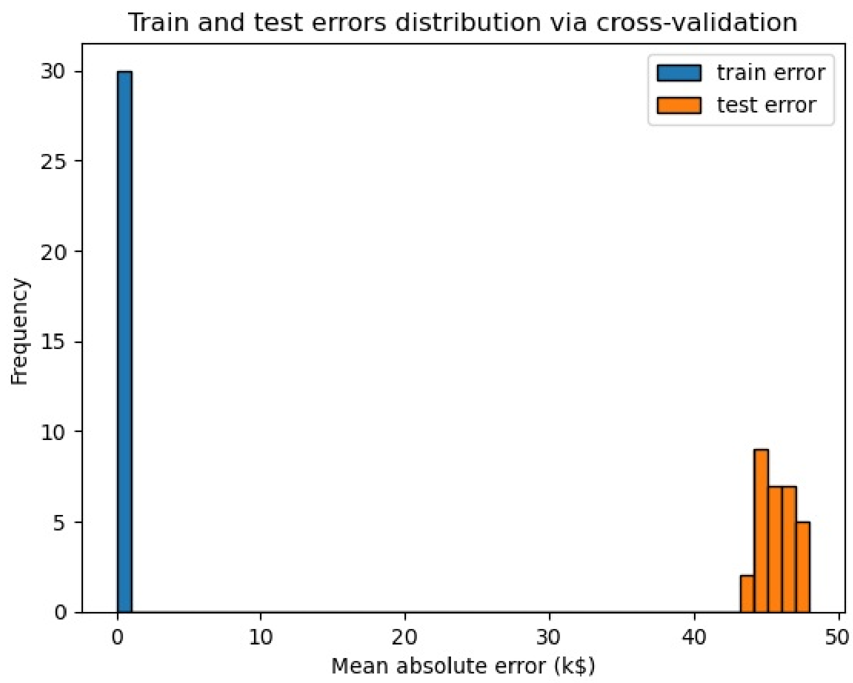
	fit_time	score_time	test_score	train_score
0	0.179096	0.002319	-46.925258	-1.517477e-14
1	0.178036	0.002705	-46.696217	-3.297965e-15
2	0.171571	0.002402	-45.068841	-9.325873e-15
3	0.184929	0.002598	-43.577655	-1.263761e-14
4	0.176815	0.002510	-48.051693	-8.812933e-15
5	0.179538	0.002546	-44.583104	-1.258597e-14
6	0.171225	0.002471	-44.422082	-1.440708e-14
7	0.182246	0.002606	-45.121698	-3.745497e-15
8	0.171188	0.002658	-44.960310	-1.121239e-14
9	0.183904	0.002601	-45.140519	-3.459765e-15
10	0.170983	0.002383	-47.278801	-3.387471e-15
11	0.179569	0.002634	-46.769008	-1.579357e-14
12	0.169093	0.002633	-46.077512	-1.521264e-14
13	0.181397	0.002560	-45.554380	-1.007635e-14
14	0.166513	0.002178	-47.208009	-1.236220e-14
15	0.179190	0.002617	-44.386838	-1.367726e-14
16	0.177197	0.002308	-45.999537	-1.338120e-14
17	0.190602	0.002549	-46.767820	-3.249769e-15
18	0.169810	0.003089	-45.111378	-3.211901e-15
19	0.190374	0.002644	-47.405591	-3.793692e-15
20	0.167575	0.002357	-43.397712	-1.505428e-14
21	0.184882	0.002521	-46.005055	-1.210746e-14
22	0.172707	0.002563	-45.363898	-1.424528e-14
23	0.183790	0.002722	-46.793429	-3.645663e-15
24	0.172408	0.002428	-46.325968	-3.583697e-15
25	0.181001	0.002663	-45.719835	-1.529526e-14
26	0.168962	0.002228	-44.414701	-3.421897e-15
27	0.189130	0.003174	-46.212782	-1.321940e-14
28	0.188468	0.002342	-45.682183	-1.285449e-14
29	0.142049	0.002061	-47.664193	-3.091412e-15

we used the negative mean abs error (higher is better) $\rightarrow$ let's transform it into a positive mean abs error (smaller is better)

- this is the same as we did in the previous segment on overfitting and underfitting

In [24]:
```
scores = pd.DataFrame()
scores[["train error", "test error"]] = -cv_results[
    ["train_score", "test_score"]
]
```

In [25]:
```
import matplotlib.pyplot as plt
scores.plot.hist(bins=50, edgecolor="black")
plt.xlabel("Mean absolute error (k$)")
_ = plt.title("Train and test errors distribution via cross-validation")
```



What do we see here? Call-out?

- train error is very small, actually 0. thus, the model is certainly not underfitting
- test error is much larger -- indicates that we are overfitting.
- **intuition**: the model is memorized too many observations from the training set. because there is some extent of noise in all observations, this means that the memorized observations do not replicate in the test dataset, leading to high test error.

Validation curve

hyperparameters:

- parameters that are not directly learned in the training process, but are chosen by us and can impact the performance of the model
- for instance: the number of neighbors in a k-nearest neighbors model
- the degree of a polynomial

Often, choosing the right hyperparameters is crucial to move a model towards a better test set performance (be it moving away from underfitting or from overfitting)

We can find how the training and test erorrs behave as a function of the hyperparameters with the **validation curve**.

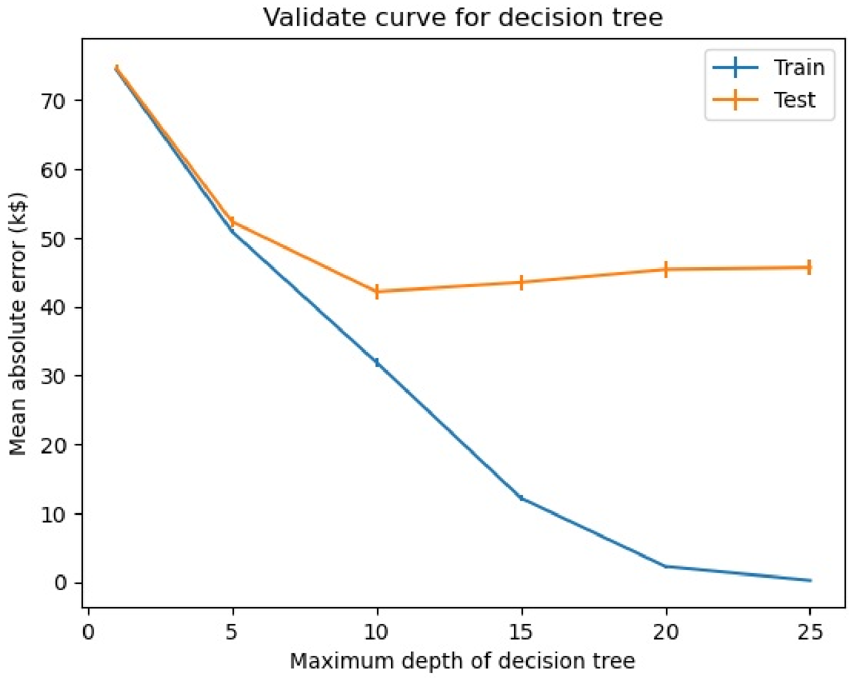
Let's apply to curve to our example from above: we can use the `max_depth` hyperparameter and see how the trade-off between over- and underfitting behaves.

In [26]:
```
import numpy as np
from sklearn.model_selection import ValidationCurveDisplay

max_depth = np.array([1, 5, 10, 15, 20, 25]) # this defines the depth of a tree

disp = ValidationCurveDisplay.from_estimator(
    regressor,
    data,
    target,
    param_name="max_depth", # I guess this needs to be a parameter of the respective regressor model?
    param_range=max_depth,
    cv=cv, # we re-use the shuffle split from above
    scoring="neg_mean_absolute_error",
    negate_score=True, ## this again re-converts the neg_mean_absolute_error to the mean_absolute_error
    std_display_style="errorbar", # show the variability of the mean absolute error
    n_jobs=2,
)

_ = disp.ax_.set(
    xlabel="Maximum depth of decision tree",
    ylabel="Mean absolute error (k$)",
    title="Validate curve for decision tree",
)
```



explain the regression tree? what is the depth?

- simply put: the deeper, the more flexible the tree because it uses more features for explaining the outcome

There are 3 areas in the validation curve

- first, we'd like to be in a sweet spot between neither over- nor underfitting. (mention bias-variance trade-off??)
- this is equivalent to the lowest possible test error, in this case with a depth of around 10. here we say "the model generalizes".
- the other regions are worse because the test error is larger
  - for `max_depth` < 10, we underfit: both train and test error are large
  - for `max_depth` > 10, we overfit: training error becomes very small, but test error increases. "memorizing" of observations and too much associated noise.

in how much detail should I explain this? use it as exercise?

Notes

- we should not only look at the mean errors, but also at the standard deviation of the errors
  - why? -- for two types of hyperparameters, we could have the same mean errors, but a much larger variance. in this case, we would like to go for the model with lower variance. this is actually displayed with the error bars in the figure above (with the `std_display_style` attribute. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.ValidationCurveDisplay.html)
  - in our case, the variance seems to be fairly stable across hyperparameter values
- we see also that the train and test error slightly diverge for `max_depth` = 10 already. Thus, the model might be overfitting. But this is the best we can do with changing the hyperparameter.

Summary

- how to identify whether a model is generalizing, overfitting or underfitting
- how to check the influence of a hyperparameter on the underfit/overfit tradeoff

Effect of the sample size in cross-validation

- before: how under-/overfitting and generalizing are impacted by hyperparameter values
- now: how they are impacted by the size of the data we have available

In [27]:
```
# already done above
# housing = fetch_california_housing(as_frame=True)
# data, target = housing.data, housing.target
# target *= 100

# we already have the DecisionTreeRegressor
```

Learning curve

- we can produce similar figures with test and train errors by varying the size of the dataset
- this called the **learning curve**
- it is informative about whether adding more training data improves the model's generalization performance

In [28]:

```
train_sizes = np.linspace(0.1, 1.0, num=5, endpoint=True) # we split the values from 0.1 to 1 into 5 equally-si
# endpoint=True includes also 1.0.
train_sizes
```

Out[28]:

```
array([0.1  , 0.325, 0.55 , 0.775, 1.   ])
```

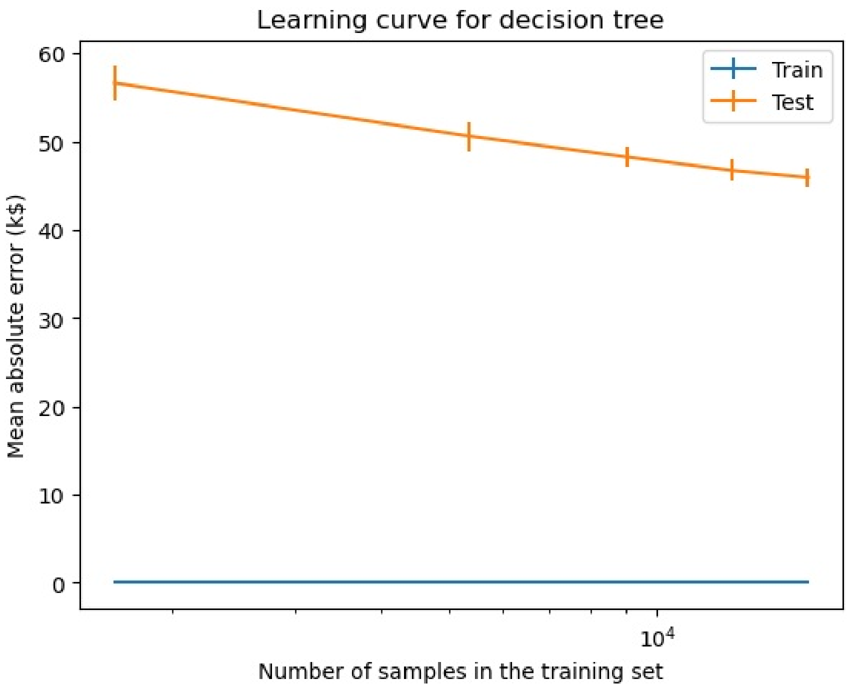
In [29]:

```
# again use a ShuffleSplit for cross-validation
cv = ShuffleSplit(n_splits=30, test_size=0.2)
```

In [30]:

```
# let's do the experiment
from sklearn.model_selection import LearningCurveDisplay

display = LearningCurveDisplay.from_estimator(
    regressor,
    data,
    target,
    train_sizes=train_sizes,
    cv=cv,
    # score_type="both", # score both train and test errors -- this is the default, and not shown in the previo
    # to make it consistent, ignore it here
    scoring="neg_mean_absolute_error",
    negate_score=True,
    score_name="Mean absolute error (k$)",
    std_display_style="errorbar",
    n_jobs=2,
)
_ = display.ax_.set(xscale="log", title="Learning curve for decision tree")
```



What do we see?

- train error is 0 -- overfitting
- the variability of the error declines as we increase the size of the data set -- noise has less impact on the error
- test error declines as we increase sample size
- since it does not seem to "flatten out", we it may be useful to increase the size of the dataset further to reduce the test error
- in case we reached the "flat" part of the learning curve, then we may have reached the Bayes error rate (the irreducible error, ie $Var(\varepsilon)$) with the available model. To further reduce the test error, we'd have to try ot a more complex model.

Summary

- we can use the learning curve to check whether it's worth adding more data, or whether we may need to try out a more complex model

Exercise M2.01

For the questions, see the exercise document

explain what the SVM does?

for classification problems:

- in a p-dimensional space, tries to find a boundary in $p-1$-dimensional space to separate the data points
- give example with 2-dimensional space?
- it seeks to maximize the distance from the closest data points on either side of the boundary
  - intuition: this will also lead to low generalization error (since "new" data are likely to fall into the same area as the "existing" data)
- see wikipedia: https://en.wikipedia.org/wiki/Support_vector_machine -- the γ parameter is in the radial basis function

what is the γ parameter? it makes the classification boundary more non-linear.

Solution

In [31]:
```
blood_transfusion = pd.read_csv("../datasets/blood_transfusion.csv")
data = blood_transfusion.drop(columns="Class")
target = blood_transfusion["Class"]
```

In [32]:
```
data.head()
```

Out[32]:

	Recency	Frequency	Monetary	Time
0	2	50	12500	98
1	0	13	3250	28
2	1	16	4000	35
3	2	20	5000	45
4	1	24	6000	77

(1) Create a predictive pipeline

In [33]:
```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

model = make_pipeline(
    StandardScaler(), SVC(kernel="rbf")
)
```

In [34]:
```
model.get_params().keys() # this is how we can see the name of the parameters, particularly for the validation c
```

Out[34]:
```
dict_keys(['memory', 'steps', 'verbose', 'standardscaler', 'svc', 'standardscaler__copy', 'standardscaler__with
_mean', 'standardscaler__with_std', 'svc__C', 'svc__break_ties', 'svc__cache_size', 'svc__class_weight', 'svc__
coef0', 'svc__decision_function_shape', 'svc__degree', 'svc__gamma', 'svc__kernel', 'svc__max_iter', 'svc__prob
ability', 'svc__random_state', 'svc__shrinking', 'svc__tol', 'svc__verbose'])
```

let's use the `ShuffleSplit` scheme for cross validation

In [ ]:
```

```

(2) Evaluate how well the model generalizes.

In [35]:
```
cv = ShuffleSplit(random_state=0)
cv_results = cross_validate(model, data, target, cv=cv, n_jobs=2)
cv_results = pd.DataFrame(cv_results)
```

In [36]:
```
cv_results.shape
```

Out[36]:
```
(10, 3)
```

In [37]:
```
print(
    "Accuracy score of our model:\n"
    f"{cv_results['test_score'].mean():.3f} ± "
    f"{cv_results['test_score'].std():.3f}"
)
```

```
Accuracy score of our model:
0.765 ± 0.043
```

(3) Evaluate the effect of the parameter gamma by using `sklearn.model_selection.ValidationCurveDisplay`

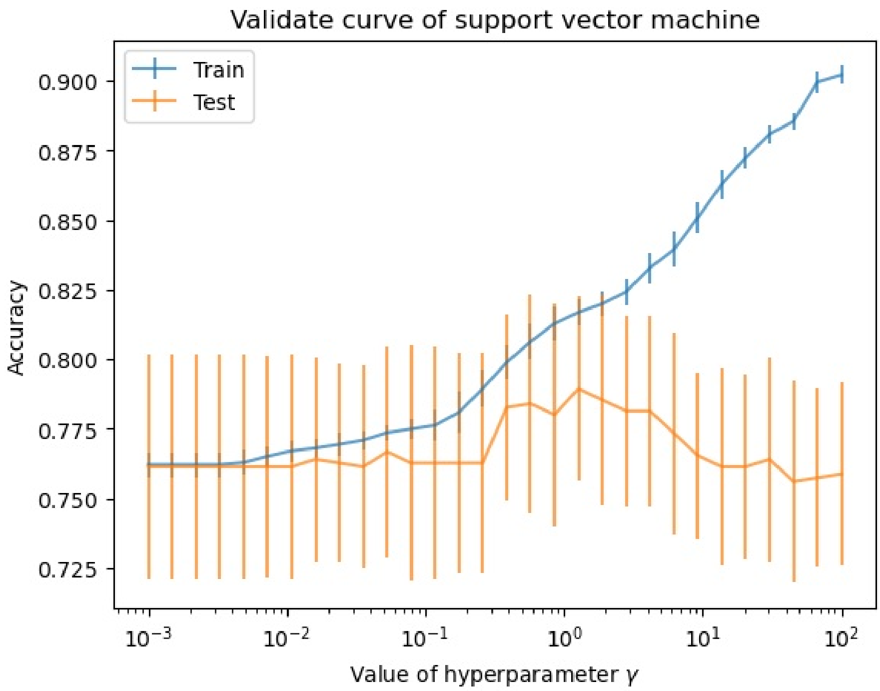
Try out different values for γ

In [38]:
```
gamma_values = np.logspace(-3, 2, num=30)

disp = ValidationCurveDisplay.from_estimator(
    model,
    data,
    target,
    param_name="svc__gamma", # we need svc__gamma instead of just gamma because we are manipulating a `Pipeline
```

```
    param_range=gamma_values,
    cv=cv, # we re-use the shuffle split from above
    scoring="accuracy",
    score_name="Accuracy",
    std_display_style="errorbar", # show the variability of the mean absolute error
    errorbar_kw={"alpha": 0.7},  # transparency for better visualization
    n_jobs=2,
)

_ = disp.ax_.set(
    xlabel="Value of hyperparameter $\gamma$",
    ylabel="Accuracy",
    title="Validate curve of support vector machine",
)
```



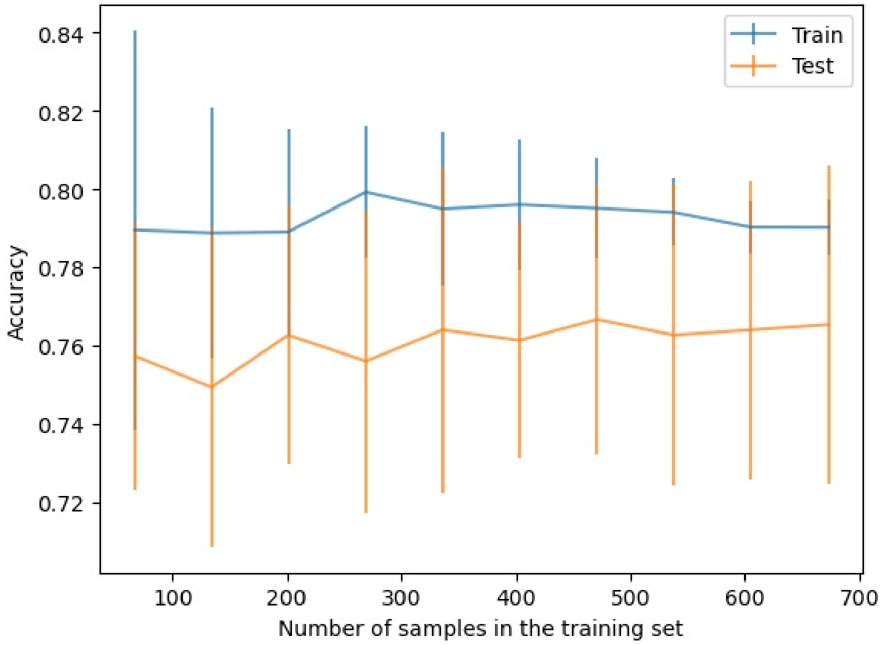
What do we see?

- overfitting starting around $\gamma = 1$? (remember higher accuracy is better)

(4) Compute the learning curve (using `sklearn.model_selection.LearningCurveDisplay`) by computing the train and test scores for different training dataset size.

In [39]:
```
train_sizes = np.linspace(0.1, 1.0, num=10, endpoint=True)
```

In [40]:
```
display = LearningCurveDisplay.from_estimator(
    model,
    data,
    target,
    train_sizes=train_sizes,
    cv=cv,
    scoring="accuracy",
    score_name="Accuracy",
    std_display_style="errorbar",
    errorbar_kw={"alpha": 0.7},  # transparency for better visualization
    n_jobs=2,
)
_ = disp.ax_.set(title="Learning curve for support vector machine")
```



Observations

- adding more samples does not improve the accuracy much -- it's always around 0.77
- note that this is about the fraction of records that are in class `"not donated"`. Therefore, even without seeing any data, the we could have a similar performance

Interpretation

- this suggests that this model here is probably too simple
- features themselves are not informative, for any model
- the other default hyperparameter values of `SVC` are not good
- the model itself may be wrong

In [ ]:

Quiz M2.02

see the exercise document

"Quiz: over- and underfitting and learning curves (5 minutes, in pairs; if time-permitting)"

Solutions 1.b, 2.a, 3.d, 4.b, 5.c/d, 6.b

In [ ]:

In [ ]: