Mudcard

- **Why do we need a validation set if we already have a test set?**
  - the validation set is used for model selection
  - the test set is used to estimate how well the model is expected to perform on previously unseen data points once it is deployed
- **Is there a best practice for reading the Lasso and Ridge regressions directly from the graph?**
  - I'm not sure what you are refering to.
  - Please expand on it and submit your question on Ed
- **Wondering why do we want coeffs to be exactly zero for feature selection?**
  - If a coefficient is 0, the corresponding feature does not contribute to the prediction of the linear model
  - y' = <w,X> = sum (w1 * X1 + w2 * X2 + w3 * X3 + ...)
  - If w_i is 0, the corresponding feature X_i is not used in the prediction because w_i * X_i = 0 if w_i = 0
- **How can we interpret the learned coefficients from LogisticRegression?**
  - See PS5 problem 1 for the answer

Lecture 9: CV and Intro to Interpretability

By the end of this lecture, you will be able to

- perform basic hyperparameter tuning
- apply GridSearchCV
- describe why interpretability is important

The supervised ML pipeline

0. Data collection/manipulation: you might have multiple data sources and/or you might have more data than you need

- you need to be able to read in datasets from various sources (like csv, excel, SQL, parquet, etc)
- you need to be able to filter the columns/rows you need for your ML model
- you need to be able to combine the datasets into one dataframe

1. Exploratory Data Analysis (EDA): you need to understand your data and verify that it doesn't contain errors

- do as much EDA as you can!

2. Split the data into different sets: most often the sets are train, validation, and test (or holdout)

- practitioners often make errors in this step!
- you can split the data randomly, based on groups, based on time, or any other non-standard way if necessary to answer your ML question

3. Preprocess the data: ML models only work if X and Y are numbers! Some ML models additionally require each feature to have 0 mean and 1 standard deviation (standardized features)

- often the original features you get contain strings (for example a gender feature would contain 'male', 'female', 'non-binary', 'unknown') which needs to be transformed into numbers
- often the features are not standardized (e.g., age is between 0 and 100) but it needs to be standardized

4. Choose an evaluation metric: depends on the priorities of the stakeholders

- often requires quite a bit of thinking and ethical considerations

5. Choose one or more ML techniques: it is highly recommended that you try multiple models

- start with simple models like linear or logistic regression
- try also more complex models like nearest neighbors, support vector machines, random forest, etc.

6. Tune the hyperparameters of your ML models (aka cross-validation or hyperparameter tuning)

- ML techniques have hyperparameters that you need to optimize to achieve best performance
- for each ML model, decide which parameters to tune and what values to try
- loop through each parameter combination
  - train one model for each parameter combination
  - evaluate how well the model performs on the validation set
- take the parameter combo that gives the best validation score
- evaluate that model on the test set to report how well the model is expected to perform on previously unseen data

7. Interpret your model: black boxes are often not useful

- check if your model uses features that make sense (excellent tool for debugging)

- often model predictions are not enough, you need to be able to explain how the model arrived to a particular prediction (e.g., in health care)

By the end of this lecture, you will be able to

- **perform basic hyperparameter tuning**
- apply GridSearchCV
- describe why interpretability is important

Let's put everything together!

import packages

load your dataset

create feature matrix and target variable

for i in random_states:

- split the data
- preprocess it
- decide which hyperparameters you'll tune and what values you'll try
- for combo in hyperparameters:
  - train your ML algo
  - calculate training scores
  - calculate validation scores
- select best model based on the mean and std validation scores
- predict the test set using the best model
- return your test score (generalization error)
- return the best model

The deliverables of an ML pipeline are:

- at the very least:
  - the n number of best models you selected based on the vaidation scores
    - yes, you need ALL of them
  - the n number of test scores (or you can calculate mean and stdev test scores)
- in some cases, you need your whole pipeline in a reproducable format like a container
  - others might want to see how you preprocess and split your data
  - what models you try
  - your evaluation metric
  - they might want to see your train and validation scores to verify the range of your hyperparameters

- etc.

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder, OrdinalEncc
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

df = pd.read_csv('../data/adult_data.csv')

# let's separate the feature matrix X, and target variable y
y = df['gross-income'] # remember, we want to predict who earns more than 50
X = df.loc[:, df.columns != 'gross-income'] # all other columns are features

# collect which encoder to use on each feature
# needs to be done manually
ordinal_ftrs = ['education']
ordinal_cats = [[' Preschool',' 1st-4th',' 5th-6th',' 7th-8th',' 9th',' 10th
                ' Some-college',' Assoc-voc',' Assoc-acdm',' Bachelors',' Ma
onehot_ftrs = ['workclass','marital-status','occupation','relationship','rac
minmax_ftrs = ['age','hours-per-week']
std_ftrs = ['capital-gain','capital-loss']

# collect all the encoders into one preprocessor
preprocessor = ColumnTransformer(
    transformers=[
        ('ord', OrdinalEncoder(categories = ordinal_cats), ordinal_ftrs),
        ('onehot', OneHotEncoder(sparse_output=False,handle_unknown='ignore'
        ('minmax', MinMaxScaler(), minmax_ftrs),
        ('std', StandardScaler(), std_ftrs)])

prep = Pipeline(steps=[('preprocessor', preprocessor)]) # for now we only pr
```

Quiz

Let's recap preprocessing. Which of these statements are true?

Basic hyperparameter tuning

In [2]:
```
help(ParameterGrid)
```

```
Help on class ParameterGrid in module sklearn.model_selection._search:

class ParameterGrid(builtins.object)
 |  ParameterGrid(param_grid)
 |
 |  Grid of parameters with a discrete number of values for each.
 |
 |  Can be used to iterate over parameter value combinations with the
 |  Python built-in function iter.
 |  The order of the generated parameter combinations is deterministic.
 |
 |  Read more in the :ref:`User Guide <grid_search>`.
 |
 |  Parameters
 |  ----------
 |  param_grid : dict of str to sequence, or sequence of such
 |      The parameter grid to explore, as a dictionary mapping estimator
 |      parameters to sequences of allowed values.
 |
 |      An empty dict signifies default parameters.
 |
 |      A sequence of dicts signifies a sequence of grids to search, and is
 |      useful to avoid exploring parameter combinations that make no sense
 |      or have no effect. See the examples below.
 |
 |  Examples
 |  --------
 |  >>> from sklearn.model_selection import ParameterGrid
 |  >>> param_grid = {'a': [1, 2], 'b': [True, False]}
 |  >>> list(ParameterGrid(param_grid)) == (
 |  ...    [{'a': 1, 'b': True}, {'a': 1, 'b': False},
 |  ...     {'a': 2, 'b': True}, {'a': 2, 'b': False}])
 |  True
 |
 |  >>> grid = [{'kernel': ['linear']}, {'kernel': ['rbf'], 'gamma': [1, 1
0]}]
 |  >>> list(ParameterGrid(grid)) == [{'kernel': 'linear'},
 |  ...                               {'kernel': 'rbf', 'gamma': 1},
 |  ...                               {'kernel': 'rbf', 'gamma': 10}]
 |  True
 |  >>> ParameterGrid(grid)[1] == {'kernel': 'rbf', 'gamma': 1}
 |  True
 |
 |  See Also
 |  --------
 |  GridSearchCV : Uses :class:`ParameterGrid` to perform a full parallelize
d
 |      parameter search.
 |
 |  Methods defined here:
 |
 |  __getitem__(self, ind)
 |      Get the parameters that would be ``ind``th in iteration
 |
 |      Parameters
 |      ----------
```

```
 |      ind : int
 |          The iteration index
 |
 |      Returns
 |      -------
 |      params : dict of str to any
 |          Equal to list(self)[ind]
 |
 |  __init__(self, param_grid)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __iter__(self)
 |      Iterate over the points in the grid.
 |
 |      Returns
 |      -------
 |      params : iterator over dict of str to any
 |          Yields dictionaries mapping each estimator parameter to one of i
ts
 |          allowed values.
 |
 |  __len__(self)
 |      Number of points on the grid.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables
 |
 |  __weakref__
 |      list of weak references to the object
```

In [3]:
```
# let's train a random forest classifier

# we will loop through nr_states random states so we will return nr_states t
nr_states = 5
test_scores = np.zeros(nr_states)
final_models = []

# loop through the different random states
for i in range(nr_states):
    print('randoms state '+str(i+1))

    # first split to separate out the training set
    X_train, X_other, y_train, y_other = train_test_split(X,y,train_size = 0

    # second split to separate out the validation and test sets
    X_val, X_test, y_val, y_test = train_test_split(X_other,y_other,train_si

    # preprocess the sets
    X_train_prep = prep.fit_transform(X_train)
    X_val_prep = prep.transform(X_val)
    X_test_prep = prep.transform(X_test)
```

```
    # decide which parameters to tune and what values to try
    # the default value of any parameter not specified here will be used
    param_grid = {
                 'penalty': ['l1'],
                 'C': np.logspace(-3,3,7), # only the inverse of the regula
                 'solver': ['saga'],
                 'max_iter': [10000]
                 }
    print(param_grid)

    # we save the train and validation scores
    # the validation scores are necessary to select the best model
    # it's optional to save the train scores, it can be used to identify hig
    train_score = np.zeros(len(ParameterGrid(param_grid)))
    val_score = np.zeros(len(ParameterGrid(param_grid)))
    models = []

    # loop through all combinations of hyperparameter combos
    for p in range(len(ParameterGrid(param_grid))):
        params = ParameterGrid(param_grid)[p]
        print('   ',params)
        clf = LogisticRegression(**params,random_state = 42*i) # initialize
        clf.fit(X_train_prep,y_train) # fit the model
        models.append(clf) # save it
        # calculate train and validation accuracy scores
        y_train_pred = clf.predict(X_train_prep)
        train_score[p] = accuracy_score(y_train,y_train_pred)
        y_val_pred = clf.predict(X_val_prep)
        val_score[p] = accuracy_score(y_val,y_val_pred)
        print('   ',train_score[p],val_score[p])

    # print out model parameters that maximize validation accuracy
    print('best model parameters:',ParameterGrid(param_grid)[np.argmax(val_s
    print('corresponding validation score:',np.max(val_score))
    # collect and save the best model
    final_models.append(models[np.argmax(val_score)])
    # calculate and save the test score
    y_test_pred = final_models[-1].predict(X_test_prep)
    test_scores[i] = accuracy_score(y_test,y_test_pred)
    print('test score:',test_scores[i])
```

```
randoms state 1
{'penalty': ['l1'], 'C': array([1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+
02, 1.e+03]), 'solver': ['saga'], 'max_iter': [10000]}
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.001)}
    0.803030303030303 0.8015970515970516
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.01)}
    0.8421375921375921 0.8444410319410319
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.1)}
    0.8487919737919738 0.8539619164619164
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(1.0)}
    0.8510954135954136 0.8539619164619164
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
0.0)}
    0.8511977886977887 0.8541154791154791
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
00.0)}
    0.8509930384930385 0.8541154791154791
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
000.0)}
    0.8509930384930385 0.8541154791154791
best model parameters: {'solver': 'saga', 'penalty': 'l1', 'max_iter': 1000
0, 'C': np.float64(10.0)}
corresponding validation score: 0.8541154791154791
test score: 0.8524489482573315
randoms state 2
{'penalty': ['l1'], 'C': array([1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+
02, 1.e+03]), 'solver': ['saga'], 'max_iter': [10000]}
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.001)}
    0.80246723996724 0.8023648648648649
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.01)}
    0.8422911547911548 0.8372235872235873
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.1)}
    0.8510954135954136 0.8487407862407862
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(1.0)}
    0.8530405405405406 0.8513513513513513
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
0.0)}
    0.8529381654381655 0.851044226044226
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
00.0)}
    0.8530917280917281 0.8508906633906634
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
000.0)}
    0.8530917280917281 0.8508906633906634
best model parameters: {'solver': 'saga', 'penalty': 'l1', 'max_iter': 1000
0, 'C': np.float64(1.0)}
corresponding validation score: 0.8513513513513513
test score: 0.8541378780899739
```

```
randoms state 3
{'penalty': ['l1'], 'C': array([1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+
02, 1.e+03]), 'solver': ['saga'], 'max_iter': [10000]}
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.001)}
    0.8028767403767404 0.7959152334152334
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.01)}
    0.8447993447993448 0.8390663390663391
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.1)}
    0.851044226044226 0.8481265356265356
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(1.0)}
    0.853552416052416 0.8493550368550369
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
0.0)}
    0.8538083538083538 0.8488943488943489
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
00.0)}
    0.854013104013104 0.8487407862407862
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
000.0)}
    0.854013104013104 0.8487407862407862
best model parameters: {'solver': 'saga', 'penalty': 'l1', 'max_iter': 1000
0, 'C': np.float64(1.0)}
corresponding validation score: 0.8493550368550369
test score: 0.8536772608628896
randoms state 4
{'penalty': ['l1'], 'C': array([1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+
02, 1.e+03]), 'solver': ['saga'], 'max_iter': [10000]}
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.001)}
    0.8069717444717445 0.7976044226044227
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.01)}
    0.8463349713349714 0.8389127764127764
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.1)}
    0.854013104013104 0.8465909090909091
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(1.0)}
    0.8545761670761671 0.8481265356265356
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
0.0)}
    0.8539619164619164 0.8468980343980343
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
00.0)}
    0.854013104013104 0.8468980343980343
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
000.0)}
    0.8539619164619164 0.8467444717444718
best model parameters: {'solver': 'saga', 'penalty': 'l1', 'max_iter': 1000
0, 'C': np.float64(1.0)}
corresponding validation score: 0.8481265356265356
test score: 0.8473821587594043
```

```
randoms state 5
{'penalty': ['l1'], 'C': array([1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+
02, 1.e+03]), 'solver': ['saga'], 'max_iter': [10000]}
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.001)}
    0.8003685503685504 0.8043611793611793
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.01)}
    0.8435196560196561 0.8453624078624079
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(0.1)}
    0.8495085995085995 0.856418918918919
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64
(1.0)}
    0.8517096642096642 0.8579545454545454
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
0.0)}
    0.851965601965602 0.8574938574938575
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
00.0)}
    0.8518632268632269 0.8570331695331695
    {'solver': 'saga', 'penalty': 'l1', 'max_iter': 10000, 'C': np.float64(1
000.0)}
    0.8518632268632269 0.8570331695331695
best model parameters: {'solver': 'saga', 'penalty': 'l1', 'max_iter': 1000
0, 'C': np.float64(1.0)}
corresponding validation score: 0.8579545454545454
test score: 0.8487640104406572
```

Things to look out for

- are the ranges of the hyperparameters wide enough?
  - if you are unsure, save the training scores and plot the train and val scores!
  - do you see underfitting? model performs poorly on both training and validation sets?
  - do you see overfitting? model performs very good on training but worse on validation?
  - if you don't see both, expand the range of the parameters and you'll likely find a better model
  - read the manual and make sure you understand what the hyperparameter does in the model
    - some parameters (like regularization parameters) should be evenly spaced in log because there is no upper bound
    - some parameters (like l1_ratio for the elastic net) should be linearly spaced because they have clear lower and upper bounds
  - **if the best hyperparameter is at the edge of your range, you definitely need to expand the range if you can**
- not every hyperparameter is equally important
  - some parameters have little to no impact on train and validation scores

    - visualize the results if in doubt
- is the best validation score similar to the test score?
    - it's usual that the validation score is a bit better than the test score
    - but if the difference between the two scores is significant over multiple random states, something could be off

Quiz

By the end of this lecture, you will be able to

- perform basic hyperparameter tuning
- **apply GridSearchCV**
- describe why interpretability is important

Hyperparameter tuning with folds

- the steps are a bit different

In [4]:
```
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import make_pipeline

df = pd.read_csv('../data/adult_data.csv')

# let's separate the feature matrix X, and target variable y
y = df['gross-income'] # remember, we want to predict who earns more than 50
X = df.loc[:, df.columns != 'gross-income'] # all other columns are features

ordinal_ftrs = ['education']
ordinal_cats = [[' Preschool',' 1st-4th',' 5th-6th',' 7th-8th',' 9th',' 10th
                ' Some-college',' Assoc-voc',' Assoc-acdm',' Bachelors',' Ma
onehot_ftrs = ['workclass','marital-status','occupation','relationship','rac
minmax_ftrs = ['age','hours-per-week']
std_ftrs = ['capital-gain','capital-loss']

# collect all the encoders
preprocessor = ColumnTransformer(
    transformers=[
        ('ord', OrdinalEncoder(categories = ordinal_cats), ordinal_ftrs),
        ('onehot', OneHotEncoder(sparse_output=False,handle_unknown='ignore'
        ('minmax', MinMaxScaler(), minmax_ftrs),
        ('std', StandardScaler(), std_ftrs)])

# all the same up to this point
```

In [5]:
```
# we will use GridSearchCV and the parameter names need to contain the ML al
# the parameters of some ML algorithms have the same name and this is how we

param_grid = {
```

```
            'logisticregression__penalty': ['l1'],
            'logisticregression__solver': ['saga'],
            'logisticregression__max_iter': [10000],
            'logisticregression__C': np.logspace(-3,3,7) # only the inverse
                 }

nr_states = 3
test_scores = np.zeros(nr_states)
final_models = []

for i in range(nr_states):
    # first split to separate out the test set
    # we will use kfold on other
    X_other, X_test, y_other, y_test = train_test_split(X,y,test_size = 0.2,

    # splitter for other
    kf = KFold(n_splits=4,shuffle=True,random_state=42*i)

    # the classifier
    clf = LogisticRegression(**params,random_state = 42*i) # initialize the

    # let's put together a pipeline
    # the pipeline will fit_transform the training set (3 folds), and transf
    # then it will train the ML algorithm on the training set and evaluate i
    # it repeats this step automatically such that each fold will be an eval
    pipe = make_pipeline(preprocessor,clf)

    # use GridSearchCV
    # GridSearchCV loops through all parameter combinations and collects the
    grid = GridSearchCV(pipe, param_grid=param_grid,scoring = 'accuracy',
                        cv=kf, return_train_score = True, verbose=True)

    # this line fits the model on other and loops through the 4 different va
    grid.fit(X_other, y_other)
    # save results into a data frame. feel free to print it and inspect it
    results = pd.DataFrame(grid.cv_results_)
    #print(results)

    print('best model parameters:',grid.best_params_)
    print('validation score:',grid.best_score_) # this is the mean validatio
    # save the model
    final_models.append(grid)
    # calculate and save the test score
    y_test_pred = final_models[-1].predict(X_test)
    test_scores[i] = accuracy_score(y_test,y_test_pred)
    print('test score:',test_scores[i])
```

```
Fitting 4 folds for each of 7 candidates, totalling 28 fits
best model parameters: {'logisticregression__C': np.float64(100.0), 'logisti
cregression__max_iter': 10000, 'logisticregression__penalty': 'l1', 'logisti
cregression__solver': 'saga'}
validation score: 0.8529253685503686
test score: 0.8479963150621833
Fitting 4 folds for each of 7 candidates, totalling 28 fits
best model parameters: {'logisticregression__C': np.float64(1.0), 'logisticr
egression__max_iter': 10000, 'logisticregression__penalty': 'l1', 'logisticr
egression__solver': 'saga'}
validation score: 0.8498925061425061
test score: 0.8581298940580377
Fitting 4 folds for each of 7 candidates, totalling 28 fits
best model parameters: {'logisticregression__C': np.float64(1.0), 'logisticr
egression__max_iter': 10000, 'logisticregression__penalty': 'l1', 'logisticr
egression__solver': 'saga'}
validation score: 0.852311117936118
test score: 0.8506064793489944
```

In [6]:
```
results
```

Out[6]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_logisticregre
0	0.141838	0.011799	0.009326	0.000240	
1	0.296977	0.028689	0.009311	0.000110	
2	1.522729	0.272064	0.009537	0.000155	
3	8.284712	0.586564	0.009746	0.000292	
4	16.500764	1.776865	0.009721	0.000058	
5	21.036766	2.211653	0.009714	0.000211	
6	21.543215	2.167504	0.009744	0.000130	

7 rows × 22 columns

Things to look out for

- less code but more stuff is going on in the background hidden from you
  - looping over multiple folds
  - .fit_transform and .transform is hidden from you
- nevertheless, GridSearchCV and pipelines are pretty powerful
- working with folds is a bit more robust because the best hyperparameter is selected based on the average score of multiple trained models

Quiz

Can we use GridSearchCV with sets prepared by train_test_split in advance? Use the sklearn manual or stackoverflow to answer the question.

By the end of this lecture, you will be able to

- perform basic hyperparameter tuning
- apply GridSearchCV
- **describe why interpretability is important**

Intro to interpretability

Example 1:

- A bank uses ML to review loans.

	Requested	Received	Interest Rate
Person A	10k	8k	4%
Person B	10k	10k	5%
Person C	10k	2k	5%

- Which person received the worst outcome?
- All three applicants have similar financial backgrounds
- Person C is a member of the protected class but persons A and B are not
- Person C sues the bank for discrimination
- How can the bank check whether the algorithm does not discriminate against protected classes of people?

Example 2:

- An ML model predicts that a patient is at risk of heart disease
- Doctors will ask why?
- The 'why' is very important here
- Should the prediction be trusted?
  - If the model predicts heart disease because of the patient's zip code, the clinician will push back
    - bias or some spurious correlation is suspected, model needs to be revised
  - If model predict high cholesterol is the main reason, that aligns with medical knowledge
- An interpretable outcome can inform next steps!

    - If high cholesterol is the main reason for the predicted heart disease risk, diet, life style changes, and statin might be the solution
    - If family history is the main driver of the prediction, earlier screenings might be the solution

Interpretable or explainable ML (XML or XAI)

There are two main types:

- global explanations
    - does the model make predictions based on reasonable features?
    - one value per feature, it is a vector of shape (n_{ftrs})
    - it describes how important each feature is generally
    - good start but cannot be used to explain predictions of one datapoint!
- local explanations
    - can we trust the model's prediction for one specific data point?
    - one value per feature and data point, it is a 2D array with a shape of (n_{points},n_{ftrs}) - the same shape as your feature matrix
    - it describes how important each feature is for predicting one particular data point
    - required when working with human data!

Mud card