```
import sklearn
sklearn.show_versions()
```

- Displays the versions of the dependencies for scikit-learn

```
pydoc modules
```

—MODULE 0: INTRODUCTION TO MACHINE LEARNING CONCEPTS

§ 0.1.1 What is Machine Learning?

- https://www.youtube.com/watch?v=f0b11x2tAZw
- Identifying Irises
  - ¬ Describe flowers with numbers and identify with mathematical rules
  - ¬ petal size and sepal size
- Identifying rich people from census data

- Rule creation is automated and built from the data
- Predictive analysis
  - ¬ relies on statistical tools, but go beyond standard stats
  - ¬ many sournces of variability
  - ¬ noise
    - ◊ a performance review may be affected by the mood of the boss...
- Memorizing
  - ¬ given a new individual, predict based on the closest match that's already in the database
  - ¬ nearest neighbor predictor
  - ¬ error rate
    - ◊ test set: all data that we're examining is within the data set
- Generalizing
  - ≠ to Memorizing
  - test data ≠ training data
  - ¬ now use that model on data that has not been used to develop the model
  - ¬ error will increase
- Data matrix
  - ¬ samples
    - ◊ rows are observations
  - ¬ features
    - ◊ columns are descriptors
- Supervised machine learning
  - ¬ data matrix,
    - ◊ X,
    - ◊ with n samples
    - ◊ and m features
  - ¬ target
    - ◊ y,
    - ◊ a property of each observation
  - ¬ goal is to predict y
- Unsupervised learning
  - ¬ No labels
    - ◊ not covered in class for now
  - ¬ Algorithm finds trends on its own
  - ¬ matrix, X,
  - ¬ no available target

- ¬ extract some goal from the structure or patterns in the data
  - ◊ not covered
- Regression and classificaiton
  - ¬ Classification
    - ◊ y is discrete
    - ◊ what kind of iris: setosa, versicolor, virginica
  - ¬ Regression
    - ◊ y is continuous, numerical quantity
    - ◊ wage prediciton

—MODULE 1.1: TABULAR DATA EXPLORATION

Predictive Modeling Pipeline

- Preparing the data for analysis:
  - ¬ Loading the data
  - ¬ Differentiating the variables of the data set
    - ◊ Numerical
    - ◊ Categorical
  - ¬ Visualizing the distribution of the variables

§ 1.1.1 First look at our dataset

- 01_tabular_data_exploration.ipynb
- Loading the adult census dataset
  - ¬ CSV file
  - ¬ pandas will read it into the kernel

```
adult_census = pd.read_csv("../datasets/adult-census.csv")
```

  - ¬ Goal:
    - ◊ predict yes or no:
      - † does someone earn more than 50K
      - † based on
        - » age
        - » emmployment
        - » education
        - » family status
- The variables (columns) in the dataset
  - ¬ Pandas dataframe
    - ◊ where imported data is stored
    - ◊ two dimensions
      - † rows: samples
      - † columns: features

  - ¬ **X**
    - ◊ data
    - ◊ columns in data
      - † feature
      - † variable
      - † attribute
      - † covariate
    - ◊ rows in data
      - † sample
      - † record

† instance
† observation

¬ **y**
◊ column in data
† target

¬ Class imbalance
◊ more or fewer samples in one class than the others
◊ may require special techniques

¬ Defining columns (variables) as numerical or categorical:

```
numerical_columns = [
    "age", "education-num", "capital-gain", "capital-
loss",
    "hours-per-week"]
categorical_columns = [
    "workclass", "education", "marital-status",
"occupation",
    "relationship", "race", "sex", "native-country"]
all_columns = numerical_columns + categorical_columns
+ [target_column]

adult_census = adult_census[all_columns]
```

¬ Check the size of samples and features:

```
adult_census.shape[1]
```

† but subtract one, because one of the columns is the target

```
adult_census.shape[0]
```

• **Visual inspection of the data**

¬ Look for:
◊ can it be solved without machine learning?
◊ is the required information in the dataset?
◊ peculiarities: missing data, malfunctioning sensor, capped values

¬ Quick look:

```
adult_census.head()
```

¬ Plot a histogram

```
_ = adult_census.hist(figsize=(20, 14))
```

◊ Numerical data distrubution
† The underscore variable is used by convention as a garbage variable
◊ Observations:
† Min or max values along x-axis:
» eg: few points over 70 (retired people filtered out)
† Peak values or zeros of y-axis
» eg: education-num peaks twice at 10 and again at 13
» eg: hours per week peaks at 40
» eg: capital gain and loss both have one peak then near zero elsewhere

¬ Display counts

```
adult_census["sex"].value_counts()
```

◊ Categorical variable distribution
† eg: gender or education level counts

¬ Compare two variables directly

```
pd.crosstab(index=adult_census["education"],
    columns=adult_census["education-num"])
```

◊ checking for duplicate data
† if the data shows up in one spot for each column and row

† variables with duplicate data should be eliminated
» ML has problems with duplicated or highly correclated data

¬ Compare two variables against the target

`seaborn.pairplot(...)`

◊ interactions between the different variables

† plots along the diagonal just show regular histograms of the data

`seaborn.scatterplot(...)`

† shows just a single plot

◊ Find regions which contain mostly one class

† sketch in some rough handdrawn boundaries

† decision trees make similar divisions in regions for analysis

¬ Limitations with ML

◊ imbalanced number of samples in a target variable category

◊ redundant or highly correlated columns (variables)

◊ linear models only accurately predict linear relationships

—MODULE 1.2: FITTING A SCIKIT-LEARN MODEL ON NUMERICAL DATA

§ 1.2.1 First model with scikit-learn

• 02_numerical_pipeline_introduction.ipynb

• Loading the dataset and check

```
adult_census = pd.read_csv("../datasets/adult-census-numeric.csv")
adult_census.head()
```

• Separate the data and the target

¬ Columns can be dropped as a list of names

```
columns=[target_name, ]
```

¬ Check number of rows

```
data.shape[0]
```

• K-nearest neighbors

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier()
```

• Train the model

```
model.fit(data, target)
```

◊ fit has two elements

† learning algorithm

» takes training data and target as input

† some model states

» used to predict (or transform)

• Data and target

¬ X and y

• Predict

```
target_predicted = model.predict(data)
```

• Display predictions and actual (target)

```
target_predicted[:5]
target[:5]
```

• Compare predictions to actual

```
target[:5] == target_predicted[:5]
```

◊ displays subset of true or false

• Compare total percentage correct

```
(target == target_predicted).mean()
```

¶ **Ideas and concepts**

- How to build model with:
  - ¬ tabular datasets
  - ¬ only numerical features
- **Train-test data split**
  - ¬ However, data used to build the model was also used to test
    - ◊ using data separated from training data, better assessment of prediction accuracy
    - ◊ separated data: data_test and target_test

```
accuracy = model.score(data_test, target_test)
```

- Predictive Model Assessment:
  - ¬ **generalization performance**
    - ◊ also:
      - † **predictive performance**
      - † **statistical performance**
    - ◊ test score or test error
    - ◊ comparing predicted to true targets
  - ¬ **computational performance**
    - ◊ assess computational costs of training or predicting when using a certain model

¶ **Excercise M1.02**

- 02_numerical_pipeline_cross_ex_00.ipynb

```
sklearn.neighbors.KNeighborsClassifier
```

- ◊ rarely used in practice, but intuitive to understand
- ◊ all scikit-learn models can be created without arguments
  - † simplicity and accessibility in ML

¬ Scikit-learn

```
.fit(X, y)
```
trainingax2 = sns.scatterplot(data=true_v_predict_ridge, x="measured_power", y="predicted_power", color="black", alpha=0.5)

xpoints = ypoints = plt.xlim()
ax2.plot(xpoints, ypoints, color="blue", lw=3, alpha=.5)
_ = ax2.set_title(f"Ridge: measured vs. predicted")

```
.predict(X)
```
predictions

```
.score(X, y)
```
evaluate a model

¬ Documentation in a notebook

```
KNeighborsClassifier?
```

§ 1.2.3 <u>Working with numerical data</u>

- 02_numerical_pipeline_hands_on.ipynb
- Identifying numerical data in the dataset
  - ¬ `df.dtypes`
- Separating numerical data out

```
numerical_columns =
    ["age", "capital-gain", "capital-loss", "hours-per-week"]
data[numerical_columns].head()
```

- Data range for a certain attribute
  - ¬ `df['column'].describe()`
- Train/test split helper function

```
data_train, data_test, target_train, target_test =
sklearn.model_selection.train_test_split(
        data_numeric, target, random_state=42, test_size=0.25)
```

» function returns a list of four outputs

`random_state`
† **deterministic randomization**
» repeatability
» but accuracy is dependent upon the particular split (see cross-validation)
`test_size`
† chooses how much of the data to set aside for testing
† eg: the above code takes 25% of the samples for the test set
`shuffle=False`
† eg: chooses the last 25% of the set for the test data

• Logistic Regression

```
from sklearn import set_config
set_config(display='diagram')  # to display nice model diagram
model = sklearn.linear_model.LogisticRegression()
```

◊ more commonly used in production than K-nearest...

• Train and evaluate the model

```
model.fit(data_train, target_train)
model.score(data_test, target_test)
```

§ 1.2.4 Exercise M1.03: Dummy Classifier

• 02_numerical_pipeline_ex01.ipynb
• Creates a model (like K-nearest or linear regression)
¬ However, this model will predict random or constant outcomes
¬ Type is set by `strategy`
• Dummy Classifier

```
dumdum = DummyClassifier(strategy="constant", constant=" >50K")
```

• Model is then run past methods fit and score

```
dumdum.fit(data_train, target_train)
dumdum.score(data_test, target_test)
```

◊ **baseline classifier**
† gives reference to answer question: is our statistical model accuracy good?

§ 1.2.5 Preprocessing for numerical features

• 02_numerical_pipeline_scaling.ipynb
• **Preprocessing**
¬ Adjusting the dataset to standardize the data
¬ **scaling / standardizationn**
◊ a dataset's different columns usually span different ranges
◊ **normalization**
† a transformer shifts and scales each feature to:
» mean = 0
» a unit standard deviation
¬ Different models benefit differently from scaling

• Normalizing transformer

```
scaler = sklearn.preprocessing.StandardScalar()
```

◊ beneficial to k-Nearest Neighbors and Logistic Regression
◊ not useful for Decision Trees, but not harmful either
◊ Returns arrays containing means and SDs for each column

```
scaler.fit(data_train)
```

◊ uses only a single argument: Data/X
◊ no target column
◊ generates object attributes learned from data
◊ end in underscore
`mean_` array of means for each column (attribute, feature)
`scale_` array of standard deviations for each column

```
data_train_scaled = scaler.transform(data_train)
```

◊ returns a new data set
† a numpy array, not a dataframe
◊ data transformed into scaled data
† smilar to `.predict` step with predictors
† just that it uses a function and takes the model state and data as input
† outputs transformed data (2-d array) instead of predictions (1-d array)

```
data_train_scaled = scaler.fit_transform(data_train)
```

◊ performs both actions in one step

• Reformat as dataframe and relabel columns

```
data_train_scaled = pd.DataFrame(data_train_scaled,
                                 columns=data_train.columns)
```

• Visual display of how StandardScalar changes the data:

```
num_points_to_plot = 300
sns.jointplot(data=data_train[:num_points_to_plot], x="age",
      y="hours-per-week", marginal_kws=dict(bins=15))
sns.jointplot(data=data_train[:num_points_to_plot], x="age",
      y="hours-per-week", marginal_kws=dict(bins=15))
```

◊ Distribution is the same, but scale of the axes is different

• Make a scaling pipeline

```
from sklearn.pipeline import make_pipeline
model = make_pipeline(StandardScaler(), LogisticRegression())
```

◊ automatically names each object, eg: `standardscaler`

```
model.named_steps
```

† chains together operations
† from library `sklearn.pipeline`
† Returns a model with the same methods (.fit, .predict, .score...)

• Calling fit on a pipeline scales and trains a model

```
model.fit(data_train, target_train)
```

◊ Executes three methods all together:

	Argument:
`transformer.fit()`	`data_train`
`transformer.transform()`	`data_train`
`predictor.fit()`	`data_train, target_train`

• Then do some predictions on some test data

```
predicted_target = model.predict(data_test)
```

◊ Executes two methods:

```
transformer.transform()
predictor.predict()
```

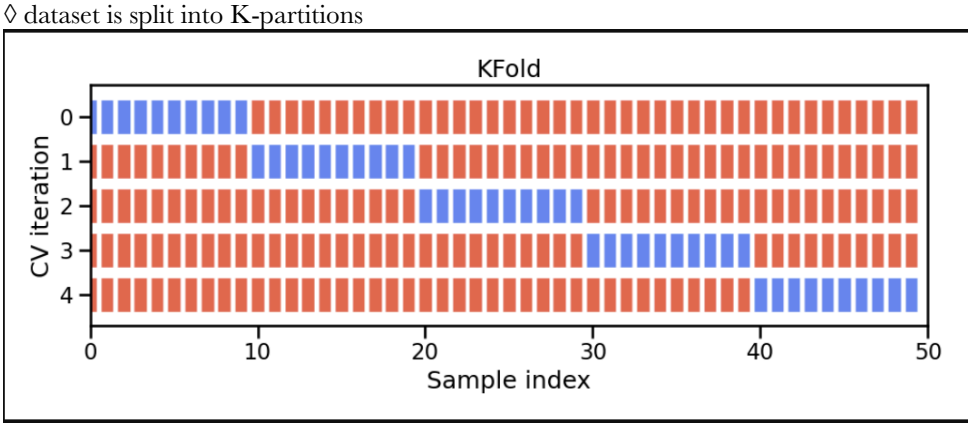
◊ Test data must be scaled before predictions can be performed

• Reasons to scale:
¬ Non-scaled data warning
◊ using non-scaled data could cause the required number of iterations to exceed `max_iter`
◊ this would leave an unfinished result as the answer

§ 1.2.6 Model evaluation using Cross-validation

• 02_numerical_pipeline_cross_validation.ipynb
• Why use cross-validation?
¬ small data sets produce very small training or test sets
¬ greater confidence in the result
• Why not?
¬ Computationally expensive
• Cross-validation does multiple splits, and multiple models, and aggregates the results
¬ estimates variability in the statistical performance
• K-fold
¬ one of several cross-validation strategies

◊ dataset is split into K-partitions



• Cross-validated, scaled linear regression:

```
model = make_pipeline(StandardScaler(), LogisticRegression())
cv_result = cross_validate(model, data_numeric, target, cv=5)
```

◊ parameter `cv` defines the number of splits
◊ takes untrained model, attributes and target sets as the other arguments
◊ Returns a dictionary, with entries for each fold
† training time
† prediction time
† score for each fold

• By default, does NOT produce a model
¬ Estimates the generalization performance
◊ trained with the full training set
¬ Estimates the variability
◊ ie: uncertainty of the generalization accuracy

• Average accuracy is:

```
scores = cv_result["test_score"]
f"{scores.mean():.3f} +/- {scores.std():.3f}"
```

◊ mean with uncertainty (standard deviation)

• Standard deviation estimates model uncertainty

— MODULE 1.3: HANDLING CATEGORICAL DATA

§ 1.3.1 Encoding of categorical variables

• 03_categorical_pipeline.ipynb
• Identify categorical variables

```
data.dtypes
```

¬ eg: int64 or object

• Select features based on data type

```
sklearn.compose.make_column_selector(dtype_include=object)
categorical_columns = categorical_columns_selector(data)
```

◊ selects columns based on their type
◊ Returns an Object which can then takes datasets as arguments
» which then returns a list of column labels

```
data_categorical = data[categorical_columns]
```

◊ removes numeric types

¶ Strategies to encode categories

• Encoding ordinal categories
    `encoder = sklearn.preprocessing.OrdinalEncoder()`
    `data_encoded = encoder.fit_transform(data_categorical)`
        `.fit` Analyzes categories to determine numerical equivalents
        `.transform` Applies the new numerical labels to the data
        ◊ Numpy array
        ◊ number of output columns matches columns inputted
    `encoder.categories_`
        ◊ this attribute shows the mapping between categories and numerical value
    ¬ Lexicographical mapping strategy
        ◊ numerical order follows alphabetical order
            † eg: size labels would be ordered:
                L < M < S < XL
    ¬ `categories`
        ◊ constructor argument allows for explicit designation
    ¬ Caution:
        ◊ Predictive models tend to assume values are ordered
            † ie: 0 < 1 < 2 < 3...
        ◊ if no ordering actually exists, eg, countries, one-hot encoding is preferred
• Encoding nominal categories
    `encoder = OneHotEncoder(sparse=False)`
        ◊ prevents downstream models from making an assumption about ordering
        ◊ creates new columns
            † as many as there are categories
            † values are 1 for the intended category and 0 for everything else
        `sparse=False`
            † is used just for easier visualization of the data
            † sparse matrices are efficient data structures when most of the matrix is zero
    `encoder.get_feature_names(data_categorical.columns)`
        ◊ informative column names can be provided by encoder object
            † then passed to a dataframe: `columns=columns_encoded`
        ◊ eg, this made the number of features 10 times largers
    `data_encoded = encoder.fit_transform(data_categorical)`
        ◊ creates a data set with 102 columns
    `columns_encoded = encoder.get_feature_names_out`
    `(data_categorical.columns)`
    `pd.DataFrame(data_encoded, columns=columns_encoded).head()`
        ◊ Wraps NumPy array in a Dataframe
        ◊ Column names generated by the one-hot encoder
• Choosing an encoding strategy
    ¬ Linear models
        ◊ impacted by ordering
            † use One Hot
            † or use Ordinal but make sure that:
                » original categories do have an order
                » encoded categories follow the same ordering
    ¬ Tree-based
        ◊ not impacted by inaccurate ordering
        ◊ use Ordinal
    ¬ **Cardinality**
        ◊ number of unique values in a column
            † high cardinality causes low computational efficiency for tree-based
                » creates gigantic, mostly empty matrices
            † avoid one-hot even if categories are unordered
• Evaluate our predictive pipeline

¬ Rare instances in a category
◊ attribute categories with very few occurances could be skipped in the train/test split
† if that sample ended up only in the test set, classifier would be unable to encode
◊ Two solutions
† list all possible categories and give to encoder with `categories` argument
† use parameter `handle_unknown`
» `OrdinalEncoder` also has this to handle rare categories

```
model = make_pipeline(OneHotEncoder(handle_unknown="ignore"),
                      LogisticRegression(max_iter=500))
```

◊ example:
`max_iter=500`
» otherwise `ConvergenceWarning`
† one-hot does NOT benefit from scaling
» all values are on the same scale: 1 or 0

• Create categorical logistic regression and check generalization performance:

```
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression

model = make_pipeline(
    OneHotEncoder(handle_unknown="ignore"),
LogisticRegression(max_iter=500)
)

from sklearn.model_selection import cross_validate
cv_results = cross_validate(model, data_categorical, target)
cv_results
```

§ 1.3.3 Exercise M1.04: Arbitrary Integer Encoding

• 03_categorical_pipeline_ex_01.ipynb
• Performance comparison between OrdinalEncoder and OneHotEncoder (from previous secion)
• Load the data

```
import pandas as pd

adult_census = pd.read_csv("../datasets/adult-census.csv")
target_name = "class"
target = adult_census[target_name]
data = adult_census.drop(columns=[target_name, "education-num"])
```

• Filter dataset for categorical features only

```
from sklearn.compose import make_column_selector as selector

categorical_columns_selector = selector(dtype_include=object)
categorical_columns = categorical_columns_selector(data)
data_categorical = data[categorical_columns]
```

• Choose OrdinalEncoder and classifier

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import OrdinalEncoder
from sklearn.linear_model import LogisticRegression

model =
make_pipeline(OrdinalEncoder(handle_unknown="use_encoded_value",
unknown_value=-1), LogisticRegression(max_iter=500))
```

• Score the accuracy

```
from sklearn.model_selection import cross_validate

results = cross_validate(model, data_categorical, target,
error_score="raise")
results
score_array = results["test_score"]
print(f"{score_array.mean():.3f} +/- {score_array.std():.3f}")
```

- Choose and score OneHotEncoder with classifier

```
from sklearn.preprocessing import OneHotEncoder

model2 = make_pipeline(OneHotEncoder(handle_unknown="ignore"),
LogisticRegression(max_iter=500))
results2 = cross_validate(model2, data_categorical, target,
error_score="raise")
results2
score_array2 = results2["test_score"]
print(f"{score_array2.mean():.3f} +/- {score_array2.std():.3f}")
```

§ 1.3.4 Using numerical and categorical variables together

- 03_categorical_pipeline_column_transformer.ipynb
- Selection based on data types

```
make_column_selector(dtype_exclude=object)
        OR:         (dtype_include=object)
```

  - ◊ Caution:
    - † object data type could include dates which relate to a quantity of elapsed time
    - † ultimately, data types must be manually inspected
- Dispatch columns to a specific processor

```
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer

categorical_preprocessor = OneHotEncoder(handle_unknown="ignore")
numerical_preprocessor = StandardScaler()
preprocessor = ColumnTransformer([
    ('one-hot-encoder', categorical_preprocessor,
categorical_columns),
    ('standard_scaler', numerical_preprocessor,
numerical_columns)])
```

  - ◊ takes a list of tuples as an argument, each listing:
    - † a name
    - † the preprocessor
    - † a list of column names to apply it to
  - ◊ column transformer then:
    - † splits the columns of the original dataset
    - † transforms each subset (calls either `fit_transform` or `transform`)
    - † concatinates the transformed subsets
  - ◊ it can be combined with a classifier in a pipeline

```
model = make_pipeline(
                preprocessor, LogisticRegression(max_iter=500))
```

  - ◊ model returned can call methods:

```
model.fit(data_train, target_train)
model.predict(data_test)
model.score(data_test, target_test)
```

- Evaluation of the model with cross-validation

```
cv_results = cross_validate(model, data, target, cv=5)
scores = cv_results["test_score"]
```

```
    f"{scores.mean():.3f} +/- {scores.std():.3f}")
```

- Fitting a more powerful model

```
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.preprocessing import OrdinalEncoder

categorical_preprocessor =
OrdinalEncoder(handle_unknown="use_encoded_value",
                                            unknown_value=-1)

preprocessor = ColumnTransformer([
    ('categorical', categorical_preprocessor,
categorical_columns)],
    remainder="passthrough")

model = make_pipeline(preprocessor,
HistGradientBoostingClassifier())
```

  - ◊ Gradient Boosting Trees
    - † Scaling of numerical features not needed
    - † Ordinal encoding works even with inappropriate ordering
  - ◊ effective for datasets with:
    - † large number of samples
    - † limited number of features
    - † mix of categorical and numerical
  - ◊ popular with data science practitioners

§1.3.5 Exercise M1.05

- 03_categorical_pipeline_ex_02.ipynb
- Reference pipeline
  - ¬ No numerical scaling
  - ¬ Integer-coded categories
    - ◊ The mean cross-validation accuracy is: 0.872 +/- 0.003 with a fitting time of 5.867
    - ◊ Trees can handle integer-coded categories as long as they are deep enough
- Scaling numerical features
  - ¬ Numerical scaling
    - ◊ The mean cross-validation accuracy is: 0.873 +/- 0.003 with a fitting time of 5.448
- One-hot encoding of categorical variables
  - ¬ 1 or 0 coded with columns for every category
    - ◊ The mean cross-validation accuracy is: 0.873 +/- 0.002 with a fitting time of 12.508
    - ◊ Much longer training times due to about 10 times more features
    - ◊ However, the implementation of `HistGradientBoostingClassifier` is incomplete

§ 1.3.6 How to define a scikit-learn pipeline and visualize it

- video_pipeline.ipynb
- Table has some missing data
  - ¬ NaN
- But only columns with no missing data are used
  - ¬ some numeric and some categorical

`SimpleImputer(strategy='median')`

- ¬ any missing data will be replaced with the median of that column

`OneHotEncoder(handle_unknown='ignore')`

- ¬ ignore:
  - ◊ missing values are assigned zeroes everywhere

```
ColumnTransformer(transformers=[
('num', numeric_transformer, numeric_features),
('cat', categorical_transformer, categorical_features),])
```

¬ applies the transformation

—MODULE 2: SELECTING THE BEST MODEL

§ Overview

- Underfitting and Overfittitting
  - ¬ ML models are imperfect
  - ¬ Fundamental trade-off
    - ◊ modeling flexibility
    - ◊ limited dataset size
- Train error vs. Test error
- Statistical variance and bias

—MODULE 2.1: OVERFITTING AND UNDERFITTING

§ Overfitting and Underfitting

- https://www.youtube.com/watch?v=xErJGDwWqys

§ 2.1.1 Cross-validation framework

- cross_validation_train_test.ipynb
- Predicting median housing values
  - ¬ continuous variable instead of discrete
  - ¬ regression

¶ Training error vs testing error

```
from sklearn.tree import DecisionTreeRegressor
regressor = DecisionTreeRegressor(random_state=0)
```

- ¬ random_state
  - ◊ setting to an integer makes the model deterministic

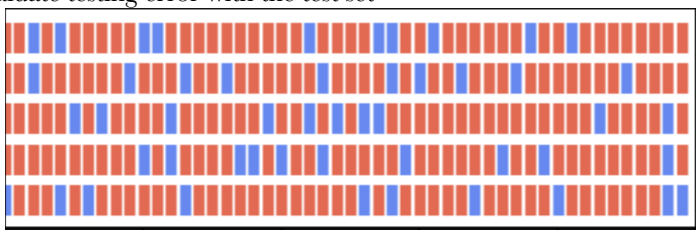
```
from sklearn.metrics import mean_absolute_error
target_predicted = regressor.predict(data)
score = mean_absolute_error(target, target_predicted)
```

- ¬ error score is in the units of the target
- ¬ Returns an error of 0
  - ◊ we made predictions on the training data
- ¬ methodological problem
  - ◊ too optimistic prediction error

- **empirical error**
  - ¬ **training error**
  - ¬ eg: we just trained a model which minimizes the training error; it was 0
    - ◊ our model had memorized the training set
  - ¬ New addition to our evaluation process
- **generalization error**
  - ¬ **testing error**
  - ¬ error that comes from predictions on test data not seen during training
  - ¬ score from `mean_absolute_error` is in the units of target

¶ Stability of the cross-validation estimates

- Small test sets
  - ¬ testing error estimate is unstable
  - ¬ won't reflect "true error rate" of the same model with lots of data
- Cross-validation
  - ¬ estimate of the variability in the generalization performance
- **Shuffle-split**
  - ¬ Randomly shuffle the order
  - ¬ Split the shuffled set
  - ¬ Train
  - ¬ Evaluate testing error with the test set



```
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)
cv_results = cross_validate(
regressor, data, target, cv=cv, scoring="neg_mean_absolute_error")
```

  - ¬ Returned as a dictionary

`cv`

  - ◊ where the ShuffleSplit object is added to a cross-validation

`n_split`

  - ◊ computation cost rises with the number of splits

`scoring="neg_mean_absolute_error"`

  - ◊ The parameter `score` always expects a function that is a score
    - † Score Higher values are better results
    - † Error Lower values are better results
  - ◊ All error metrics can be transformed into a score by making them negative

`cv_results["test_error"] = -cv_results["test_score"]`

  - ◊ change negative-valued score results to positive-value error

- Dictionary -> Dataframe

`pd.DataFrame(cv_results)`

  - ¬ makes a nice display

```
f"{cv_results['test_error'].mean():.2f} k$")
f"{cv_results['test_error'].std():.2f} k$")
```

  - ¬ The mean testing error and standard deviation
  - ¬ eg: 46.36 +/- 1.17 k$

```
f"{target.std():.2f} k$")
```

  - ¬ Now find the standard deviation of the target column
  - ¬ Note that the error is smaller than this SD
    - ◊ So, smaller than the natural scale of variation in the target variable
  - ¬ And the SD of the testing error estimate is even smaller
- However more is needed to determine if generalization performance is good enough
  - ¬ Observation of the range of target values show a lower end of 50 k$
  - ¬ The mean testing error is far too large compared to that target value
  - ¬ mean absolute percentage error might be better

¶ More detail regarding cross_validate

`cv_results = cross_validate(regressor, data, target, return_estimator=True)`
- ¬ will retrieve all fitted models produced by cross-validate
- ¬ returns them as entries in the dictionary
- ¬ this allows inspection of the internal fitted parameters

`scores = cross_val_score(regressor, data, target)`
- ¬ just returns the `test_score` list entry in the cross-validate dictionary

§ 2.1.2 Overfit-generalization-underfit
- cross_validation_validation_curve.ipynb
- Using training and testing errors to determine overfit/generalizing/underfit

¶ Overfitting vs. underfitting
- Comparison of testing and training error
- Using `cross_validate` with:
  - `neg_mean_absolute_error`
  - `ShuffleSplit`
  - `return_train_score=True`
    - ◊ returns training score as well as test score
- Change negative error into positive

```
scores[["train error", "test error"]] =
                        -cv_results[["train_score", "test_score"]]
```

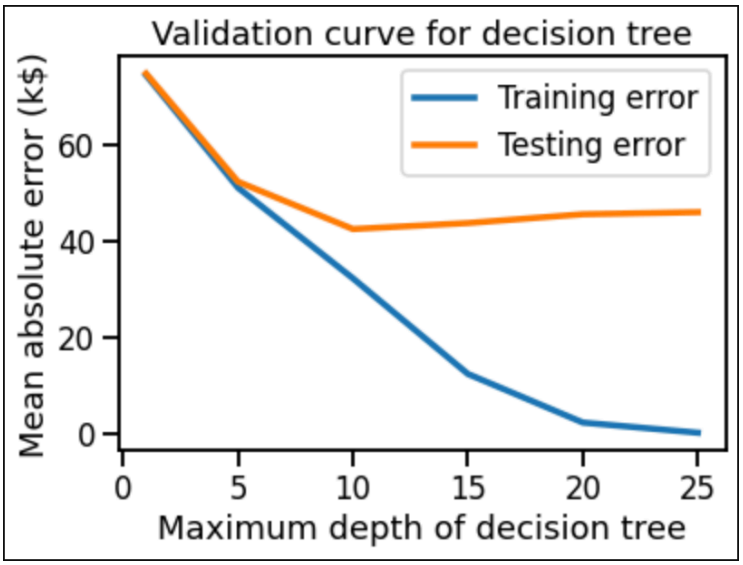
- Plot both training and testing error
  - ¬ Very small training error
    - ◊ basically 0
    - ◊ Not underfitting
      - † flexible enough to capture training set variability
  - ¬ Larger testing error
    - ◊ on the same order of magnitude as the smallest values in the dataset
    - ◊ the model memorized the many variations in the training set
      - † noisy
      - † Do not generalize well
    - ◊ Is overfitting

¶ Validation curve

```
from sklearn.model_selection import validation_curve
from sklearn.model_selection import ShuffleSplit

cv = ShuffleSplit(n_splits=30, test_size=0.2)

max_depth = [1, 5, 10, 15, 20, 25]
train_scores, test_scores = validation_curve(
            regressor, data, target, param_name="max_depth",
            param_range=max_depth, cv=cv,
            scoring="neg_mean_absolute_error", n_jobs=2)
train_errors, test_errors = -train_scores, -test_scores
```

- ¬ Model hyperparameters can be adjusted to go from underfitting to overfitting
- ¬ A validation curve plots training and testing error over various maximum depths
  - `param_range`
    - ◊ array of numbers to be used as parameters for each individual model
  - `'gamma'`
    - ◊ array of numbers to be used as parameters
  - `'svc__gamma'`
    - ◊ required when using a pipeline instead of just the regressor

¬ Use `model.get_params().keys()` to figure out parameters available



• Curve has three areas:

<10	Underfitting	Both training and testing error are high Model is too constrained, cannot capture variability
=10	Generalizing	Flexible enough to capture variability Avoids memorizing the "noise" in the target data
<10	Overfitting	Training error shrinks but testing error increases Creates decisions for noisy samples harming generalization

• At =10
  ¬ A little overfitting, shown as the gap between errors
  ¬ Also shows underfitting, since error is still far from zero
  ¬ However, best compromise by tuning just this parameter

```
plt.errorbar(max_depth, train_errors.mean(axis=1),
     yerr=train_errors.std(axis=1), label='Training error')
plt.errorbar(max_depth, test_errors.mean(axis=1),
     yerr=test_errors.std(axis=1), label='Testing error')
```

    ◊ Standard deviation
      † Just looking at mean errors (above) is limited
      † Add SD bars to more accurately graph the analysis

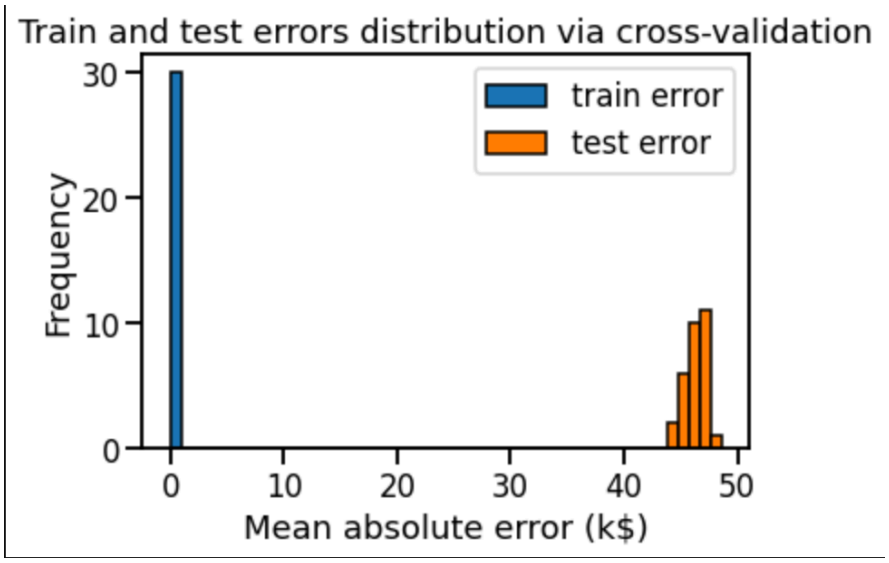
• Let's go back and find the tree depth of the original model
  ¬ Now that we know tree depth determines error levels
  ¬ original `DecisionTreeRegressor` showed a mean testing error of 47.28
    ◊ it also showed 0 for training error (see below)
  ¬ The range of errors in the graph showed that error level was near a depth of 8... maybe.
  ¬ The array of testing errors also shows the same

```
test_errors.mean(axis=1)
```

  ¬ Looking at docs, `DecisionTreeRegressor` exposes a method to return depth:

```
regressor.get_depth()
```

    ◊ requires regressor be fitted with `.fit` first
  ¬ Gives a number of 36... which means we're way off the right side of the graph.
    ◊ and makes sense if training error is 0...



§ 2.1.3 Effect of the sample size in cross-validation

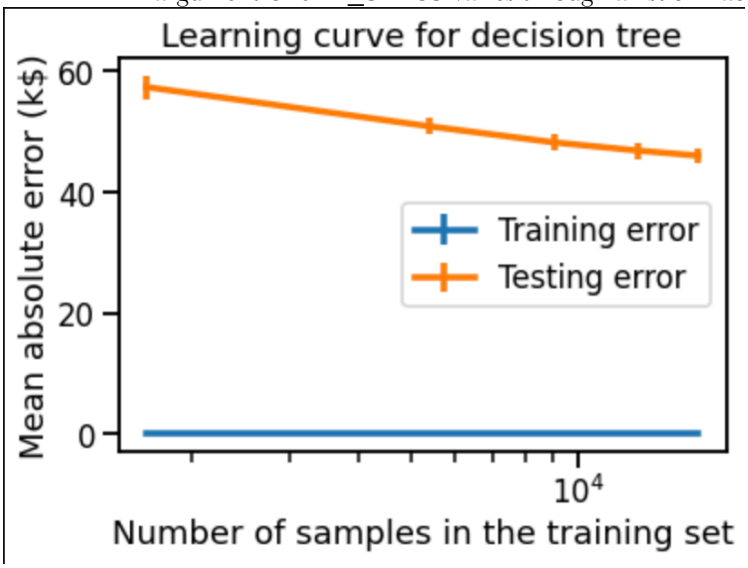
- cross_validation_learning_curve.ipynb

¶ Learning curve

- training and testing scores plotted against various training set sizes

```
results = learning_curve(
    regressor, data, target, train_sizes=train_sizes, cv=cv,
    scoring="neg_mean_absolute_error", n_jobs=2)
```

  - ◊ argument `train_sizes` varies through a list of fractions between .1 and 1



¬ Training error shows overfitting

¬ Testing error decreases as more samples are added

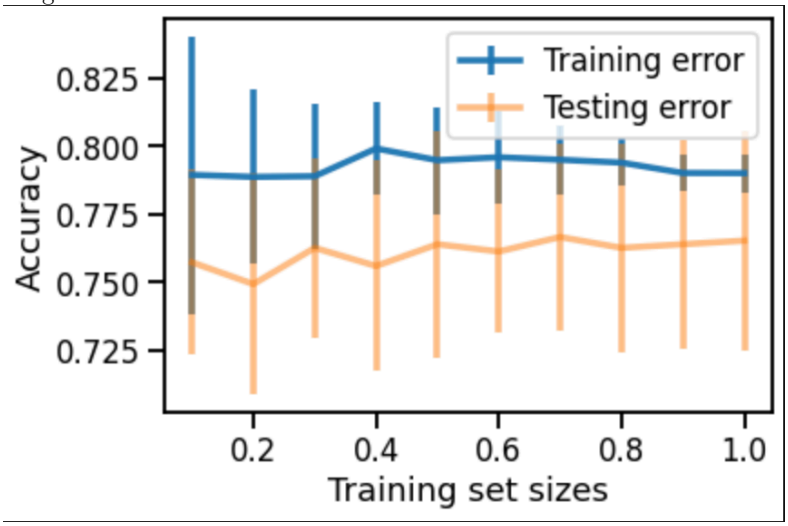
◊ there should eventually be a plateau

◊ which would show limits of the available model:

† **Bayes error rate**

§ 2.1.4 Excercise M2.01

• Retrieve parameter names from a model
```
model.get_params().keys()
```
• Regular intervals including the endpoints
```
np.linspace(0.1, 1.0, num=5, endpoint=True)
```
• Logarithmic scale regular intervals
```
np.logspace(-3, 2, num=30)
```
• Line transparency and little error bars
```
plt.errorbar(x, y, yerr=d, alpha=0.5)
```
• Logarithmic graphing
```
matplotlib.pyplot.xscale("log")
```
• Learning Curve

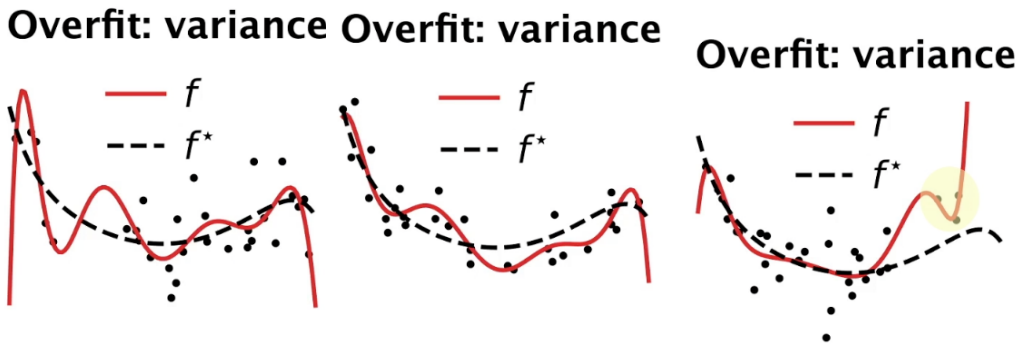


¬ Adding new samples does not really improve accuracy for either line
¬ Testing oscillates around 76%
◊ Turns out 76% of the samples belong to "not donated"
¬ DummyClassifier set to "not donated" would achieve an accuracy of 76%
◊ Maybe the small pipeline is not able to use input features to improve
◊ Maybe the input features are not very informative
◊ Maybe the hyperparameter value of the SVC was poor
◊ Maybe the choice of SVC wasn't the best option

—MODULE 2.3: BIAS VS VARIANCE TRADE-OFF

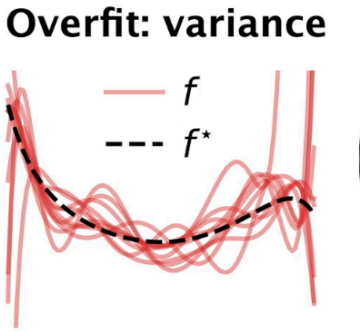
§ 2.3.1 Bias vs Variance

• https://www.youtube.com/watch?v=VOeTTiML1pY&feature=youtu.be
• Resampling the training set
¬ Limited amount of training data
¬ Training set is a small, random subset of all possible
¬ How does choosing the set effect the learned prediction outcome?
• Overfit



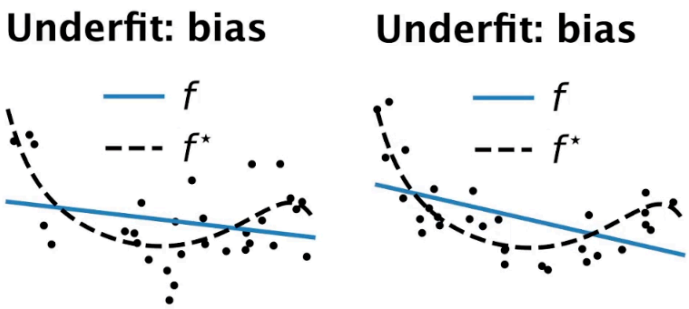
¬ Degree nine function modeling
- ◊ But if we access to a different training set
- ◊ We'd get a different model
  - † and different predictions
- ◊ but prediction error would be about the same

¬ Exessive overfitting
- ◊ yellow patch
- ◊ the weird shape around those three points captures noise
  - † but isn't representative of the overall trends

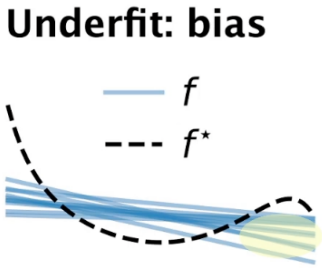


¬ The average model might be pretty good
- ◊ but each individual is overfitting and is bad

• Underfit

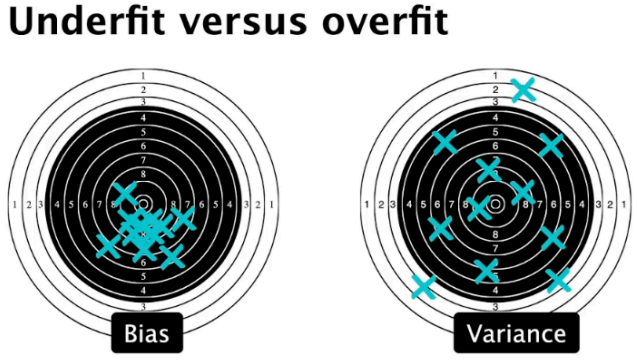


¬ if we change the sample of the training set
◊ the model does not change much
◊ slope is basically the same



¬ They make the same kind of errors
◊ they all make a systematic variance
† in the yellow patch, they all underestimate
† in the center, they all overestimate
◊ bias

• Underfit vs Overfit



        ¬ Bias/Underfit
            ◊ **mispecified models**
            ◊ All models make a systematic prediction error
            ◊ model prefers to ignore some aspects of the data
            ◊ On average they are not good
        ¬ Variance/Overfit
            ◊ **unstable models**
                † too much flexibility
            ◊ On average, they can be good
                † but prediction errors are without obvious structure
            ◊ Have high sensitivity to the training set used
            ◊ also have bad individual test errors
    • Bias-variance decomposition of the Mean Squared Error
        ¬ see Wikipedia
        ¬ the average of the squares of the errors
            ◊ variance
                † how widely spread the estimates are from one data sample to another
            ◊ bias
                † how far off the average estimated value is from the true value
        ¬ root mean square deviation
            ◊ square root of the MSE
            ◊ same units as the quantity being estimated
            ◊ for unbiased estimators, it is the standard deviationn

§ 2.4 Quiz

- Show all possible values in a column

```
target.value_counts()
target.unique()
```

- Returns just the values for the `'test_score'` key from the cross_validate dictionary

```
cross_val_score(model, X, y)
```

- Returns a more accurate assessment of a dummy_classifier from cross_validate/cross_val_score

```
scoring="balanced_accuracy"
```

- Scaling required for KNeighborsClassifier
    - ¬ Classifier compares the y-values of the K-closest neighbors to the test data
    - ¬ Model computes distances between data points
        price: 0-10000000
        temperature: 0-30
    - ¬ Distances between rows will be mostly impacted by price
        - ◊ temperature will be ignored

§ Wrap-up

- Overfitting caused by:
    - ¬ limited size of training set
    - ¬ noise in data
    - ¬ high flexibility of ML model
- Underfitting / systematic errors caused by:
    - ¬ model and parameter choice
        - ◊ lack of flexibility can't capture structure of the true data
- Fixed training set
    - ¬ minimize test error by adjusting model and parameters
- Fixed model and parameters
    - ¬ increasing training set size can:
        - ◊ decrease overfitting
        - ◊ increase underfitting
- Model which is neither overfitting nor underfitting

¬ high error remains if:
◊ variations in the target cannot be largely determined by the features
¬ **label noise**
◊ irreducible error
◊ critical feature data missing

—MODULE 3: HYPERPARAMETER TUNING

§ Overview

• **hyperparameters**
¬ control the learning process
¬ often manually tuned
¬ cannot be estimated from the data
• Do not confuse hyperparameters with parameters inferred during the training
¬ parameters define the model itself
¬ eg, coefficients in a linear model, `model.coef_`
• Objectives
¬ How to get and set hyperparameter values
¬ Fine tune full predictive modeling pipeline
¬ understand and visualize improved parameter combinations

§ 3.1.1 Set and get hyperparameters in scikit-learn

• parameter_tuning_manual.ipynb
• New pipeline function

```
model = Pipeline(steps=[
    ("preprocessor", StandardScaler()),
    ("classifier", LogisticRegression())
])
```

• Setting parameter C when creating the model

```
LogisticRegression(C=1e-3)
```

• Setting parameter C after model is generated

```
model.set_params(classifier__C=1e-3)
```

• Pipeline parameter names:

```
<model_name>__<parameter_name>
```

• List all parameter names and values

```
model.get_params()
```

◊ Get only parameter names

```
for parameter in model.get_params():
    print(parameter)
```

◊ Get only the value of a parameter

```
model.get_params()['classifier__C']
```

• Varying parameter C manually

```
for C in [1e-3, 1e-2, 1e-1, 1, 10]:
    model.set_params(classifier__C=C)
    cv_results = cross_validate(model, data, target)
    scores = cv_results["test_score"]
    print(f"Accuracy score via cross-validation with C={C}:\n"
          f"{scores.mean():.3f} +/- {scores.std():.3f}")
```

• Warning
¬ When we evaluate a family of models on test data and pick the best:
◊ Prediction accuracy cannot be trusted
◊ Test data has been used to select the model and is no longer independent

§ 3.1.2 **Excercise M3.01**

- parameter_tuning_ex_02.ipynb
-

§ 3.2.1 **Hyperparameter tuning by grid-search**

- parameter_tuning_grid_search.ipynb
- Categorical features
  - ¬ Select categorical columns

```
categorical_columns_selector =
                        make_column_selector(dtype_include=object)
categorical_columns = categorical_columns_selector(data)
```

- Usinng a tree-based model means:
  - ¬ Numerical variables don't need scaling
  - ¬ Categorical variables can be dealt with an OrdinalEncoder
    - ◊ even though coding order has no meaning
    - ◊ OrdinalEncoder avoids high-dimensional representations (ie: OneHot)
- Unknown values for OrdinalEncoder should be set to -1

```
categorical_preprocessor = OrdinalEncoder(
        handle_unknown="use_encoded_value", unknown_value=-1)
```

- GridSearchCV accomplishes search similar to the two `for` loops

```
from sklearn.model_selection import GridSearchCV
param_grid = {
    'classifier__learning_rate': (0.01, 0.1, 1, 10),
    'classifier__max_leaf_nodes': (3, 10, 30)}
model_grid_search = GridSearchCV(model, param_grid=param_grid,
                                 n_jobs=2, cv=2)
model_grid_search.fit(data_train, target_train)
```

  - ¬ Returns the best model
    - ◊ which still needs to be fitted
  - ¬ Pass model along with test data to cross_validate to score
  - ¬ `param_grid` needs to be a dictionary
    - ◊ key names in grid dict must come from `model.get_params().keys()`
  - ¬ `n_jobs` sets the number of cores for simultaneous processing
    - ◊ `-1` will use all available cores
  - ¬ `gridsearch.cv_results_`
    - ◊ shows all grid search data
- Model from grid-search has attribute which displays the chosen parameters

```
model_grid_search.best_params_
```

- Score results were similar to the for loop method
  - ¬ but it still came up with a different answer from the grid
- All scoring results from all permutations are stored in an attribute

```
model_grid_search.cv_results_
```

- Heatmap
  - ¬ Create a dataframe of all scores

```
cv_results = pd.DataFrame(
        model_grid_search.cv_results_).sort_values(
        "mean_test_score", ascending=False)
```

  - ¬ Column name management
    - ◊ Add `"param_"` to the beginning of each of the `param_grid` keys

```
column_results =
    [f"param_{name}" for name in param_grid.keys()]
```

    - ◊ then include names for columns of mean score, SD score, and rank

```
column_results +=
    ["mean_test_score", "std_test_score","rank_test_score"]
```

    - ◊ Use those to make a new dataframe

```
cv_results = cv_results[column_results]
```

◊ Drop the `param_classifier__` from the name

```
if "__" in param_name:
    return param_name.rsplit("__", 1)[1]
```

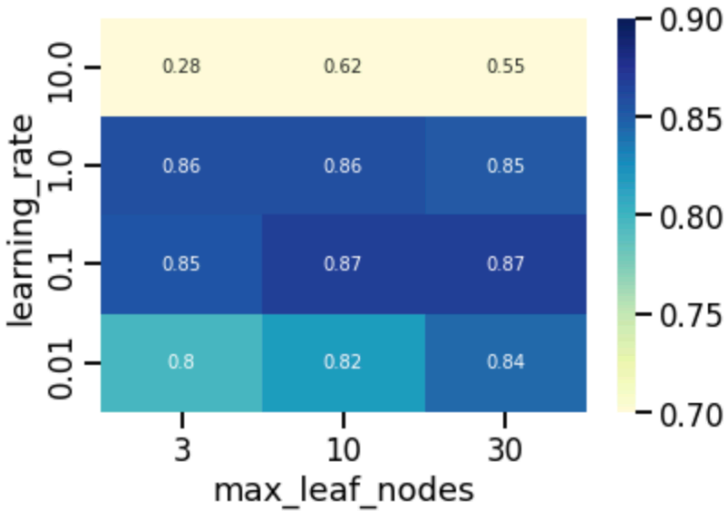
◊ Transform two parameters and mean test scores into pivot table

```
pivoted_cv_results = cv_results.pivot_table(
    values="mean_test_score", index=["learning_rate"],
    columns=["max_leaf_nodes"])
```

◊ Graph a heatmap

```
import seaborn as sns

ax = sns.heatmap(pivoted_cv_results, annot=True,
cmap="YlGnBu", vmin=0.7,vmax=0.9)
ax.invert_yaxis(s)
```

• Visual interpretation



¬ For very High values of `learning_rate`:

◊ Adjusting `max_leaf_nodes` cannot help

¬ Otherwise:

◊ As `max_leaf_node` increases, `learning_rate` must decrease

• No unique optimal parameter setting

¬ 4 models out of 12 configurations reach maximal accuracy

◊ ± random fluxuations caused by sampling

§ 3.2.2 Hyperparameter tuning by randomized-search

• parameter_tuning_randomized_search.ipynb

• Grid-search shortcomings

¬ Does not scale when number of parameters to tune is increasing

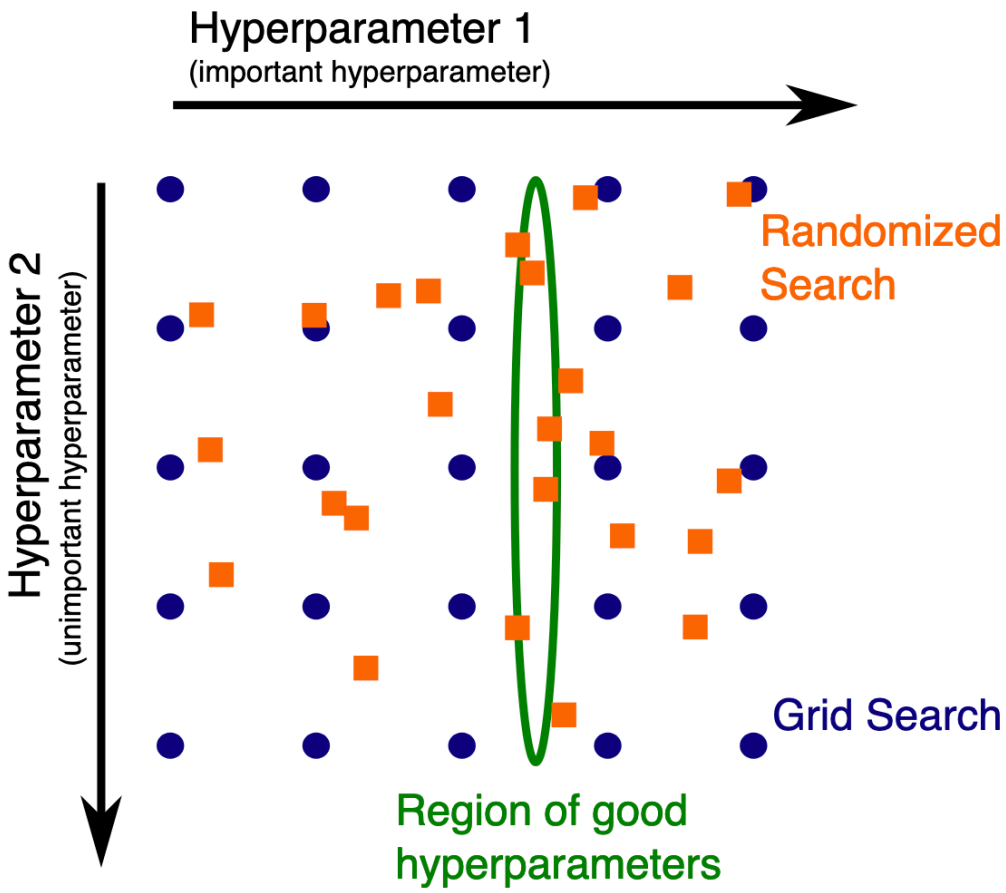
◊ they must be specified explicitly, which becomes unworkable in very large numbers

¬ Imposes search regularity (vs. randomness) via intervals which can be problematic

◊ ideal values might lie between the intervals

¶ Our predictive model

• Random parameter candidates on top of grid search candidates:



- ¬ adding more evaluations oncreases resolution in each directions
- ¬ because hyperparameter 2 is unimportant, region of good parameters aligns with grid
- ¬ **stochastic search**
  - ◊ Use of randomness in the optimization algorithm

• RandomizedSearchCV works like GridSearchCV
- ¬ But sampling distributions instead of parameter values
- ¬ eg: log-uniform distribution
  - ◊ the parameters of interest have positive values and natural log scaling
  - ◊ 0.1 is as close to 1 as 10 is

• Three new parameters are specified in addition to learning_rate and max_leaf_nodes:

`l2_regularization`	¬ strength of regularization
`min_samples_leaf`	¬ min number of samples req'd in a leaf
`max_bins`	¬ max number of bins for histogram
`learning_rate`	¬ speed that gradient-boosting corrects residuals
`max_leaf_nodes`	¬ max number of leaves for each tree

• Object that produces logarithmically-even distribution of random integers

```
from scipy.stats import loguniform

class loguniform_int:
    """Integer valued version of the log-uniform
distribution"""
    def __init__(self, a, b):
        self._distribution = loguniform(a, b)

    def rvs(self, *args, **kwargs):
        """Random variable sample"""
        return self._distribution.rvs(*args,
**kwargs).astype(int)

loguniform_int.rvs(0, 100)
```

- RandomSearchCV implemented
  - ◊ Bottom three hyperparameters need integers

```
%%time
from sklearn.model_selection import RandomizedSearchCV

param_distributions = {
    'classifier__l2_regularization': loguniform(1e-6, 1e3),
    'classifier__learning_rate': loguniform(0.001, 10),
    'classifier__max_leaf_nodes': loguniform_int(2, 256),
    'classifier__min_samples_leaf': loguniform_int(1, 100),
    'classifier__max_bins': loguniform_int(2, 255),
}

model_random_search = RandomizedSearchCV(
    model, param_distributions=param_distributions,
n_iter=10,
    cv=5, verbose=1,
)
model_random_search.fit(data_train, target_train)
```

- Then compute the accuracy with the test set

```
accuracy = model_random_search.score(data_test, target_test)
```

- Display results with modified formatting

```
def shorten_param(param_name):
    if "__" in param_name:
        return param_name.rsplit("__", 1)[1]
    return param_name

# get the parameter names
column_results = [
    f"param_{name}" for name in param_distributions.keys()]
column_results += [
    "mean_test_score", "std_test_score", "rank_test_score"]

cv_results = pd.DataFrame(model_random_search.cv_results_)
cv_results = cv_results[column_results].sort_values(
    "mean_test_score", ascending=False)
cv_results = cv_results.rename(shorten_param, axis=1)
cv_results
```

- Evaluation
  - ¬ Best model is substantially better then the second to best
    - ◊ the difference between the two mean test scores is almost 3 times the SD of the best

```
cv_results = cv_results.set_index("rank_test_score")
cv_results["mean_test_score"][1] -
                                cv_results["mean_test_score"][2]
3 * cv_results["std_test_score"][1]
```

- Remember that better parameter sets could be possible but weren't tested in the search
  - ¬ a search with `n_iter=200` can produce even more parameter results
  - ¬ this data shows there is high overlap of mean ± SD between top performing models
    - ◊ Therefore, no unique best set of parameters

§ 3.2.3 Analysis of hyperparemeter search results

- Scatterplot / heatmap
  - ¬ Sometimes can see bands in the data
- Parallel Coordinate Plot
  - ¬ Tool can show what ranges of parameters produce certain results

```
import plotly.express as px
fig = px.parallel_coordinates(cv_results.apply({...}), color=...
fig.show
```

§ 3.2.4 Analysis of hyperparameter search results

- Heatmap
  - ¬ only works with two parameters for visualization
  - ¬ can be done pair-wise, but this leads to wrong interpretation of the data
- Parallel Coordinate Plot
  - ¬ axis values transformed with $\log_{10}$ and $\log_2$
  - ¬ improves readability
  - ¬ don't forget to transform back to understand the actual parameter values indicated
- Example
  - ¬ Selecting learning_rate = -1.5 to -0.5 and max_bins = 5 to 8
    - ◊ all other parameters will still select the top performing models
      - † other parameters are not very sensitive

§ 3.2.5 Evaluation and hyperparameter tuning

- parameter_tuning_nested.ipynb
- Always return dense matrices from ColumnTransformer

```
ColumnTransformer(sparse_threshold=0,...)
```

- Best parameters from a GridSearchCV

```
model_grid_search.best_params_
```

- Evaluation of generalization performance
  - ¬ Mean/SD of grid-search/cross-validation scores might not be good generalization estimates
  - ¬ When refitting a model using the best hyperparameters on the full dataset

```
model_grid_search.fit
```

    - † method automatically performs this refit by default
    - ◊ Knowledge from the full dataset is therefore used to both:
      - † decide hyperparameters
      - † train model
  - ¬ Best practice is then to keep an external test set held out for final evaluation of refitted model
    - ◊ Basically:
      - † Use full data for GridSearchCV

```
model_grid_search = GridSearchCV(
        model, param_grid=param_grid, n_jobs=2, cv=2
```

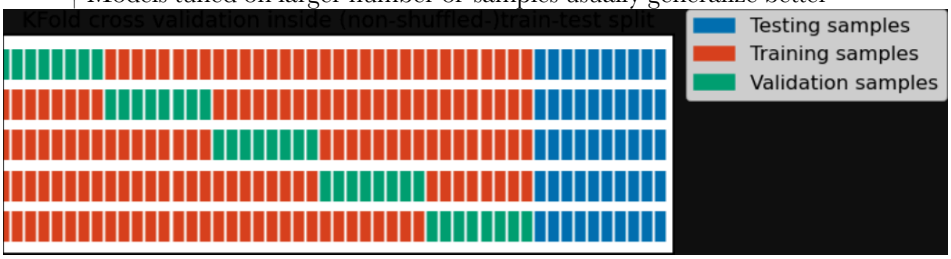
        - ‡ Which doesn't actually calculate anything
      - † Split and use train_data for fitting

```
model_grid_search.fit(data_train, target_train)
```

        - ‡ Actually performs the grid-search
      - † Use test_data for scoring

```
accuracy = model_grid_search.score(
    data_test, target_test)
```

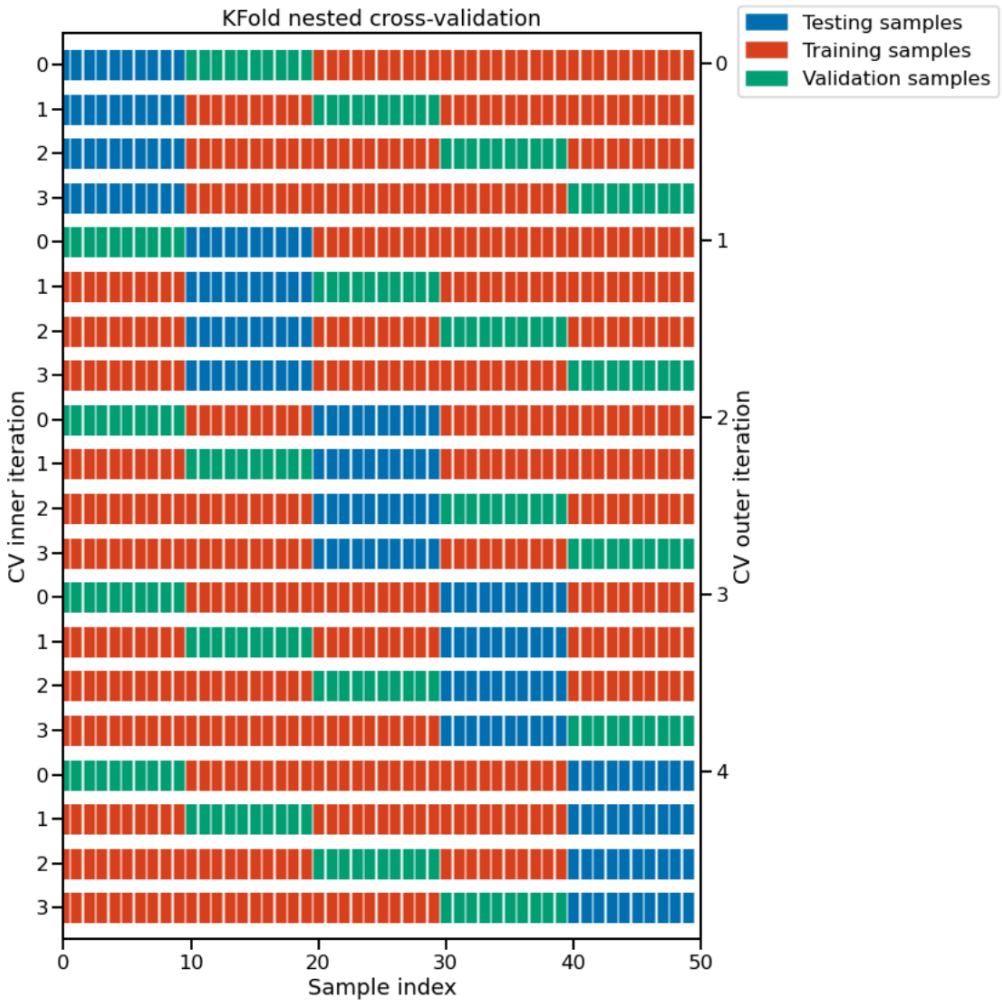
- Score analysis
  - ¬ Final test set score is almost in the range of the best internal CV score
    - ◊ This means tuning procedure did not cause overfitting
      - † Otherwise final score would have significantly lower than internal CV score
      - † No overfitting is expected, however, since few hyperparameters used in grid-search
  - ¬ Final score is higher than expected with internal CV loop/grid-search
    - ◊ Expected because the final fit is done on the entire red/green set
      - † Each CV iteration is done on a single green/red line
      - † Models tuned on larger number of samples usually generalize better



    - ◊ K-fold
      - † each line trains on red, evaluates with green
        - » best hyperparameters are chosen from these intermediate evaluation scores
      - † final model with those hyperparameters is fitted using entire red/green set
        - » evaluated with blue samples
  - ¬ **Nested cross-validation**
    - ◊ Provides a range of uncertainly to complement the single-point generalization estimate
    - ◊ Instead of using `.fit` and `.score` on the GridSearchCV model
    - ◊ Use GridSearchCV model as argument for cross_validate

```
cv_results = cross_validate(
    model_grid_search, data, target, cv=5, n_jobs=2,
    return_estimator=True)
```

      - ‡ Inner cross-validation for the hyperparameter selection
      - ‡ Outer cross-validation for generalization performance evaluation



`cv_inner = KFold(n_splits=4)`
» indexed on the left side
» each split trains a model on the red samples
» then evaluates the hyperparameter quality with the green samples

`cv_outer = KFold(n_splits=5)`
» indexed on right side
» best hyperparameters are chosen from their validation scores using green samples
» model is refitted using combined red/green samples of that outer CV iteration

† Generalization performance of 5 models from outer CV loop are evaluated with blue

¬ Comparison of the best hyperparameters

◊ Check values of the best hyperparameters

`return_estimator=True`
» must be used as an argument in the cross_validate

`estimator_in_fold.best_params_`

◊ If all values are similar, it means we can expect performance close to measured

◊ If different tuning sessions give different results, then any value will work

† Try a parallel coordinate plot to observe if certain hyperparameters don't matter

◊ Or, just use all the outputted models and give each a vote

† but very computationally expensive

§ 3.2.6 Exercise M3.02 and Quiz 3

- parameter_tuning_ex_03.ipynb
- Creates dataframe with only certain columns and dropping data samples with missing values

```
penguins_non_missing = penguins[columns + [target_name]].dropna()
```

- Box-Cox method
  - ¬ Common preprocessing strategy for positive values

```
PowerTransformer(method="box-cox")
```

— MODULE 4.1: LINEAR REGRESSION

§ Intuitions on linear models

- https://www.youtube.com/watch?v=ksEGivkPP7I
- Find a relationship between output variable and input variables
  - ¬ linear combination of those input variables
  - ¬ approximate output by taking the weighted sum of the input variable
    - ◊ eg, feature data * coefficient + feature data * coefficient + ... + intercept
- Machine Learning
  - ¬ these coefficients are learned (or tuned) from the training set

§ 4.1.1 Linear regression without scikit-learn

- linear_regression_without_sklearn.ipynb
- Using penguin dataset with flipper length and body mass
- The object returned by calling scatterplot can be used to add additional labels

```
ax = sns.scatterplot(data=penguins, x=feature_name, y=target_name,
                     color="black", alpha=0.5)
ax.set_title("Flipper length in function of the body mass")
```

- Regression problem
  - ¬ Penguin mass
  - ¬ Continuous variable
  - ¬ Range: 2700g and 6300g
- Linear relationship

```
body_mass = weight_flipper_length * flipper_length +
            intercept_body_mass
```

  - ¬ variable `weight_flipper_length` is a weight (parameter) applied to flipper length
    - ◊ positive weight: body mass goes up with flipper length
  - ¬ the coefficient (weight/parameter) has units of g/mm
  - ¬ y-intercept hits 1500g at 0mm flipper length

§ 4.1.2 Exercise M4.01

- linear_models_ex_01.ipynb
- My regression guesses:
  weights = [45, 52, 45.5, 51, 50, 49, 48, 47, 46.5, 46]
  intercepts = [-5000, -5900, -5000, -5900, -5800, -5600, -5500, -5300, -5200, -5500]
- numpy ravel turns both pandas series and pandas dataframes into 1-dimensional arrays

```
np.ravel(true_values)
```

§ 4.1.3 Linear regression using scikit-learn

- linear_regression_in_sklearn.ipynb
- Parametrization of a linear model
  - ¬ Varying parameters give different models, with varying accuracy
- Brute-force approach
  - ¬ check an array of parameters and pick the best one
- Closed-form solution
  - ¬ Best parameter values can be found by solving an equation
  - ¬ Avoids brute-force search

- Instead, use sklearn.linear_model.LinearRegression to find the coefficients and intercepts

```
linear_regression = LinearRegression()
linear_regression.fit(data, target)
weight_flipper_length = linear_regression.coef_[0]
intercept_body_mass = linear_regression.intercept_
```

- Mean squared error
  - ¬ Computes the goodness of fit of a model
  - ¬ Difficult to interpret, since value is unrelated to data

```
from sklearn.metrics import mean_squared_error
inferred_body_mass = linear_regression.predict(data)
model_error = mean_squared_error(target, inferred_body_mass)
```

- Mean absolute error
  - ¬ Value is in units of the data
  - ¬ eg, MAE = 310.00
    - ◊ on average the model made an error of ± 313 grams when predicting penguin body mass

§ 4.2.1 Exercise M4.02

- linear_models_ex_02.ipynb
- Scikit-learn models require data in column format
  - ¬ DataFrames for data
  - ¬ Series for target

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(pd.DataFrame(data), pd.Series(target))
```

    - ◊ numpy array reshaped to `(100, 1)` would also work, though
- Metrics seem to handle basic arrays just fine

```
from sklearn.metrics import mean_squared_error
error = mean_squared_error(target, predictions)
```

§ 4.2.2 Linear regression for a non-linear features-target relationship

- linear_regression_non_linear_link.ipynb
- Linear models can be made more expressive by engineering additional features
  - ¬ enhances performance on non-linear data
  - ¬ Combine a non-linear feature engineering step followed by linear regression
- numpy reshape method argument "-1"
  - ¬ means shape is automatically inferred from the length of data and the remaining dimensions

```
data = data.reshape((-1, 1))
```

    - ◊ will take the shape of a single column, as set by the second dimension
    - `data.shape` returns `(100, 1)`, eg

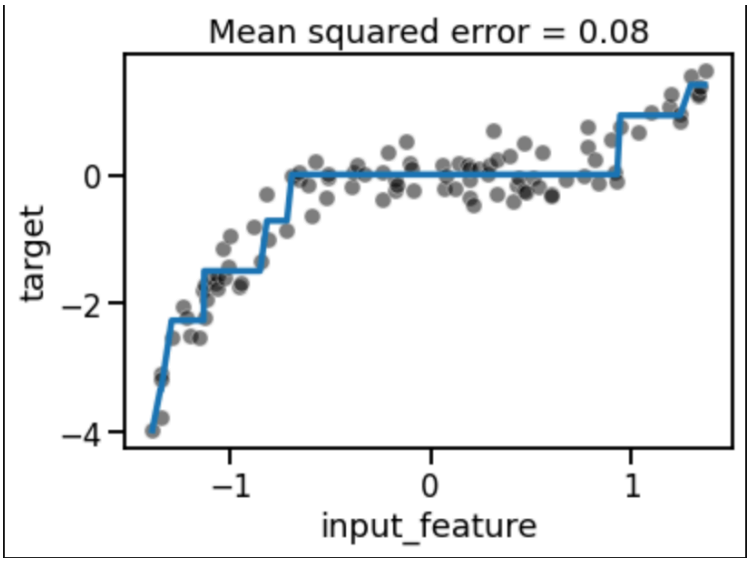
¶ 3 approaches to addressing non-linearity

- Choose a model that can natively deal with non-linearity
- Enginner a richer set of features
  - ¬ Use expert knowledge which can be directly used by the linear model
- Use a "kernel" to have a locally-based decision function
  - ¬ instead of global linear decision function

¶ Choose a model which handles non-linearity

- Decision tree regressor

```
from sklearn.tree import DecisionTreeRegressor

tree = DecisionTreeRegressor(max_depth=3).fit(data, target)
target_predicted = tree.predict(data)
mse = mean_squared_error(target, target_predicted)
```



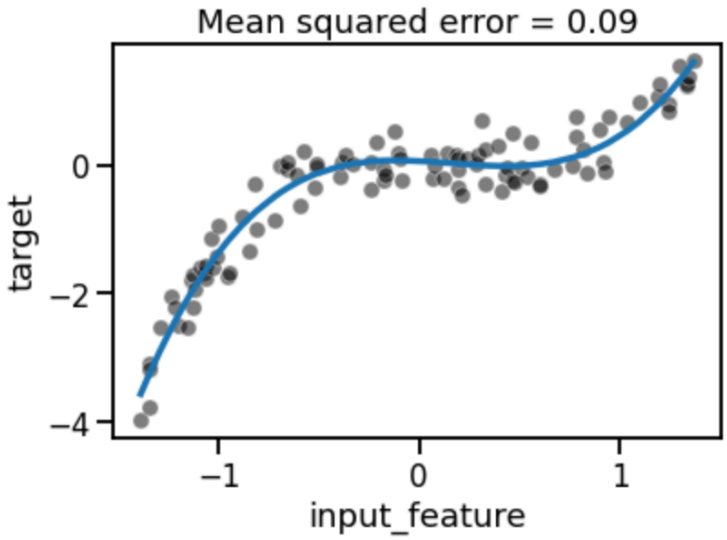
¶ Create new features using expert knowledge

- In this example, we have a cubic and squared relationship
  - ¬ we engineered the data in the first place to have this...
- Polynomial feature expansion
  - ¬ Create two new features

```
data ** 2
data ** 3
```

- By adding non-linear features can overcome linearity limitations

```
data_expanded = np.concatenate([data, data ** 2, data ** 3],
axis=1)
data_expanded.shape
linear_regression.fit(data_expanded, target)
target_predicted = linear_regression.predict(data_expanded)
mse = mean_squared_error(target, target_predicted)
```



- Polynomial Features
  - ¬ instead of generating them manually, like above, scikit-learn has a module

```
polynomial_regression = make_pipeline(
    PolynomialFeatures(degree=3),
    LinearRegression(),
)
```

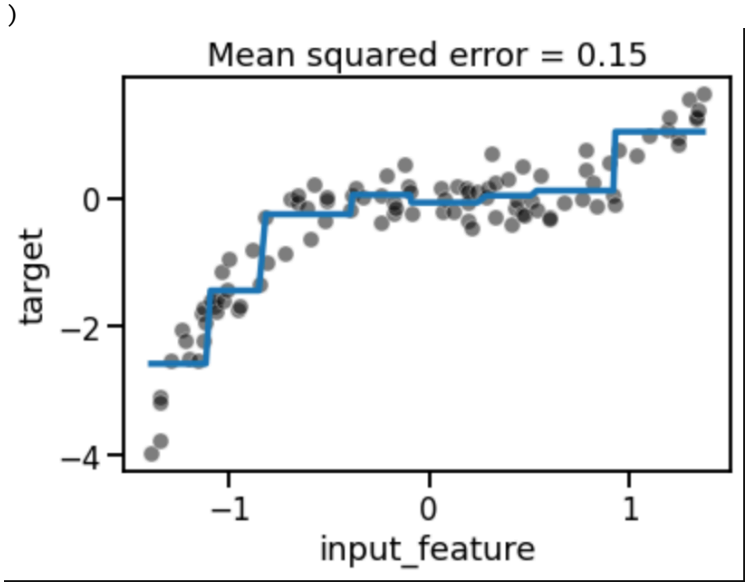
¶ Use a "kernel"

- Weight is assigned to each sample
  - ¬ Instead of learning a weight per feature
  - ¬ Not all samples will be used
  - ¬ Support vector machine algorithm
- Beyond the scope of this course
  - ¬ see SVMs

```
from sklearn.svm import SVR
svr = SVR(kernel="poly", degree=3)
svr.fit(data, target)
target_predicted = svr.predict(data)
mse = mean_squared_error(target, target_predicted)
```

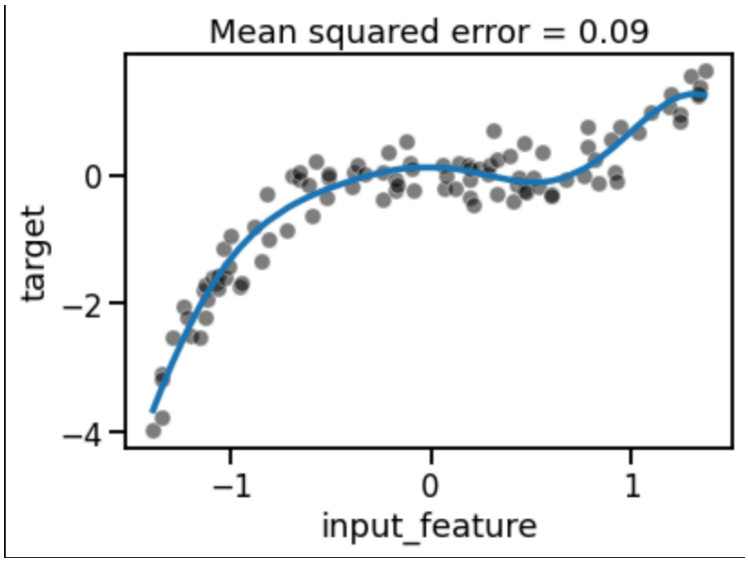
  - ¬ Produces similar graph to the above
- For datasets larger than 10,000
  - ¬ computationally more efficient to use feature expansion
  - ¬ eg, PolynomialFeatures
  - ¬ or, non-linear transformers:
    - ◊ KBinsDiscretizer

```
from sklearn.preprocessing import KBinsDiscretizer
binned_regression = make_pipeline(
    KBinsDiscretizer(n_bins=8), LinearRegression(),
)
```



    - ◊ Nystroem

```
from sklearn.kernel_approximation import Nystroem

nystroem_regression = make_pipeline(
    Nystroem(n_components=5), LinearRegression(),
)
```



§ 4.2.3 Exercise M4.03

- linear_models_ex_03.ipynb
- Training linear regression on datasets with more than one feature
- Cross-validation list of valid `scoring` arguments:

```
from sklearn.metrics import SCORERS
sorted(SCORERS.keys())
```

- Cross-validate with MAE as the metric

```
from sklearn.model_selection import cross_validate
cv_results = cross_validate(model, data, target, cv=10,
scoring="neg_mean_absolute_error", return_estimator=True)
```

  - ◊ Don't forget to change 'test_score' to positive to get MAE
- Create a box plot with column names as plot labels

```
weights = pd.DataFrame([lin_reg.coef_ for lin_reg in
cv_results['estimator']], columns=data.columns)

import matplotlib.pyplot as plt
color = {"whiskers": "black", "medians": "black", "caps":
"black"}
weights.plot.box(color=color, vert=False)
_ = plt.title("Value of linear regression coefficients")
```

  - ◊ NOTE
    - † make sure `columns=data.columns` when you create the dataframe
    - † keep default `use_index=True` for column names to display in plot

§ 4.3.1 Intuitions on regularized linear models

- How to avoid overfitting
  - Linear models are simpler
    - So expect them to be less overfitting
  - Often underfit
    - Esp when # of input variables is small vs # of data points eg less than 10 features
    - the problem is then not linearly separable
- But they can overfit...
  - Possible causes
    - n_samples << n_features, eg, genomics
    - many uninformative features

Example for linear regression

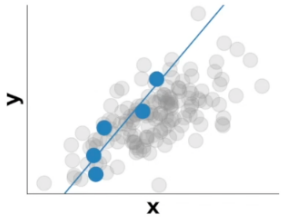
- Housing
  - eg, birthday of the first owner of the house
  - the linear model could find spurious relationships, which then causes overfitting
- Feature selection is critical

Regularization

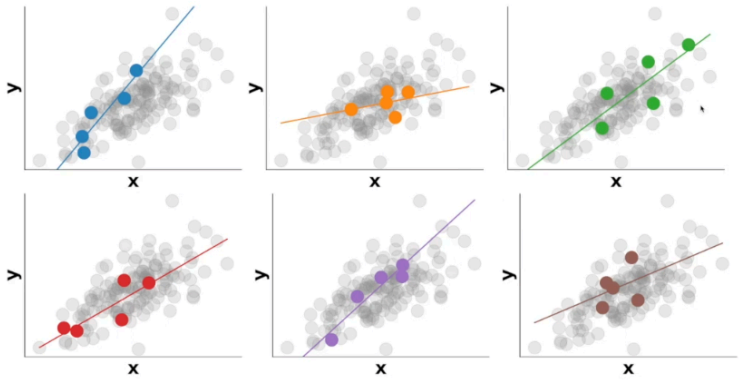
- Reduces overfitting
- Eg: Linear Regression
  - one alternative is Ridge regression
- Ridge Regression
  - pass a parameter alpha to tune it
  - control the strengths of the regularization
  - tries to pull the coefficients toward zero
    - if a large value will reduce error, it is left alone
    - but if it does not reduce error, it will pull it toward zero more relative to other features
  - Alpha
    - no good default value for alpha
    - alpha set to 0 is just normal linear regression
- Recommendation is to always use Ridge with tuned alpha

Regularization Example

- Predict y given x
- Very small training set
  - 5 points
- Linear regression gives a straight line
- However, if sample is selected at random, it might not be representative



- So if data set is small and noisy, random choice of sample can result in overfitting
  - see how general shape of data trends along a different slope
- Instead, choose another 5 points
  - creates diferent solutions significantly
  - very sensitive to selection

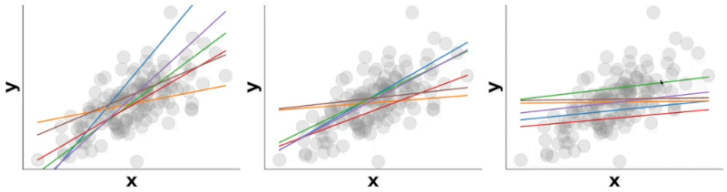


- Bias-variance tradeoff
  - Linear Regression
    - High variance problem
      - slope is very sensitive to the particular selection of the training set

No bias
model is free to find the best fit for a given set

Ridge Regression
Bias
restraints prevent the model from moving too much

Alpha too small, no regularization going to increased alpha
Good tradeoff should be found
Too much alpha puts all slopes close to zero
Intercept is not altered



Too much variance | Best tradeoff | Too much bias

alpha too small | alpha too large

Alpha trade off
GridSearchCV
use a range of possible values
only uses positive quantities
if it's zero, it's regular regression
will internally find the best alpha
RidgeCV
much faster than GridSearch
almost as fast as fitting a single Ridge model
100x faster than GridSearch
Regularization in logistic regression
LogisticRegression(C=1) is regularized by default
in scikit-learn
C is like alpha, but opposite
large C=1000, weaker regularization
C=0.01 is strongly regularized
Two effects of regularization
small C, strong reg, the region where the model has low confidence is larger
large C, weak reg, region of low confidence is much smaller
orientation with respect to the groups can change significantly
small reg, the points close to the decision boundary stongly affect the slope
large reg, many more points are involved in affecting the position of the line
to make it move, you have to consider many more data points

Take Home
Can overfit when:
n_samples is too small and n_features too large
esp, with non-informative features

Regularization for regression
    linear -> ridge
    large alpha: strong reg
Regularization classification
    logistic regression regularized by default in scikit-learn
    small C: strong reg

§ 4.3.2 Regularization of linear regression model

• linear_models_regularization.ipynb

¶ Effect of regularization

• Again, using PolynomialFeatures to augment the feature space and check with cross-validation

```
from sklearn.model_selection import cross_validate
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

linear_regression =
make_pipeline(PolynomialFeatures(degree=2),
                                        LinearRegression())
cv_results = cross_validate(linear_regression, data,
target,
                            cv=10,
scoring="neg_mean_squared_error",
                            return_train_score=True,
                            return_estimator=True)
```

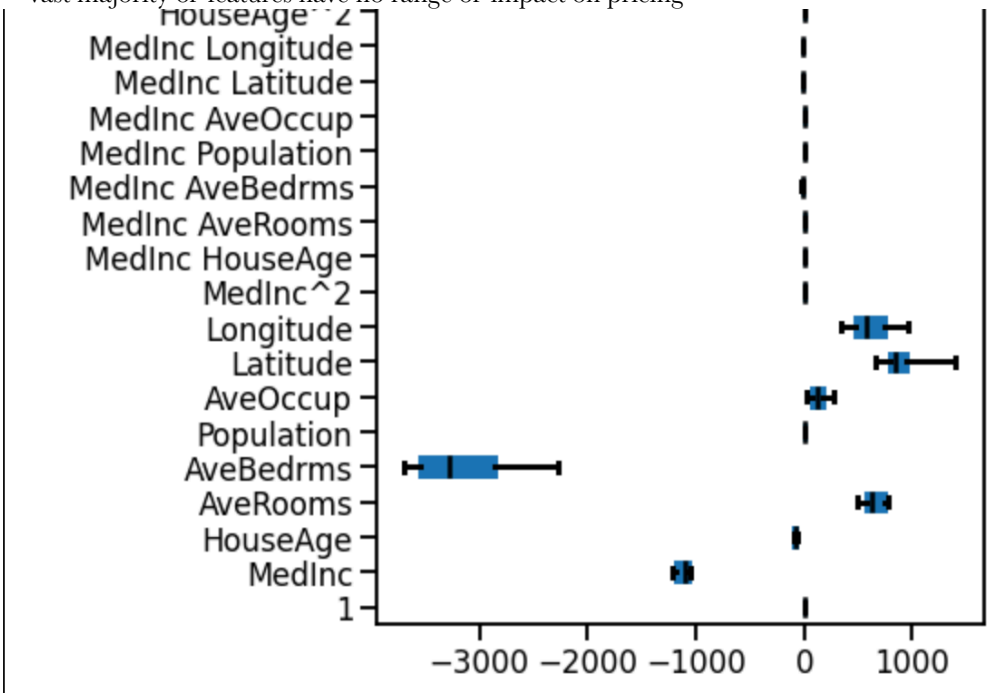
◊ Compare 'train_score' with 'test_score' to see performance gap
† gap shows overfitting happening

¬ Common risk of PolynomialFeatures is overfitting

• Examine weights of the model to confirm overfitting

¬ `get_feature_names_out` will return column names

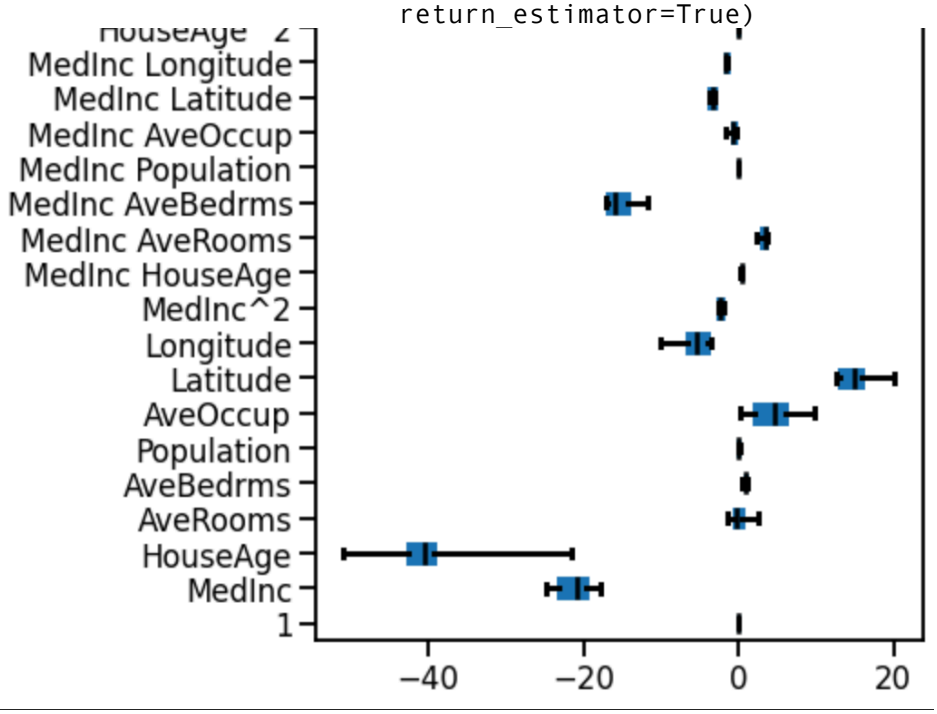
¬ vast majority of features have no range of impact on pricing



• Using Ridge instead gives a much smaller differnce in 'train_score' and 'test_score'

¬ Meaning less overfitting

```
from sklearn.linear_model import Ridge

ridge = make_pipeline(PolynomialFeatures(degree=2),
                      Ridge(alpha=100))
cv_results = cross_validate(ridge, data, target,
                            cv=10,
scoring="neg_mean_squared_error",
                            return_train_score=True,
                            return_estimator=True)
```



¬ Ridge enforces all weights to have similar magnitude

◊ overall scale of weights has moved toward 0 compared to linear reg above

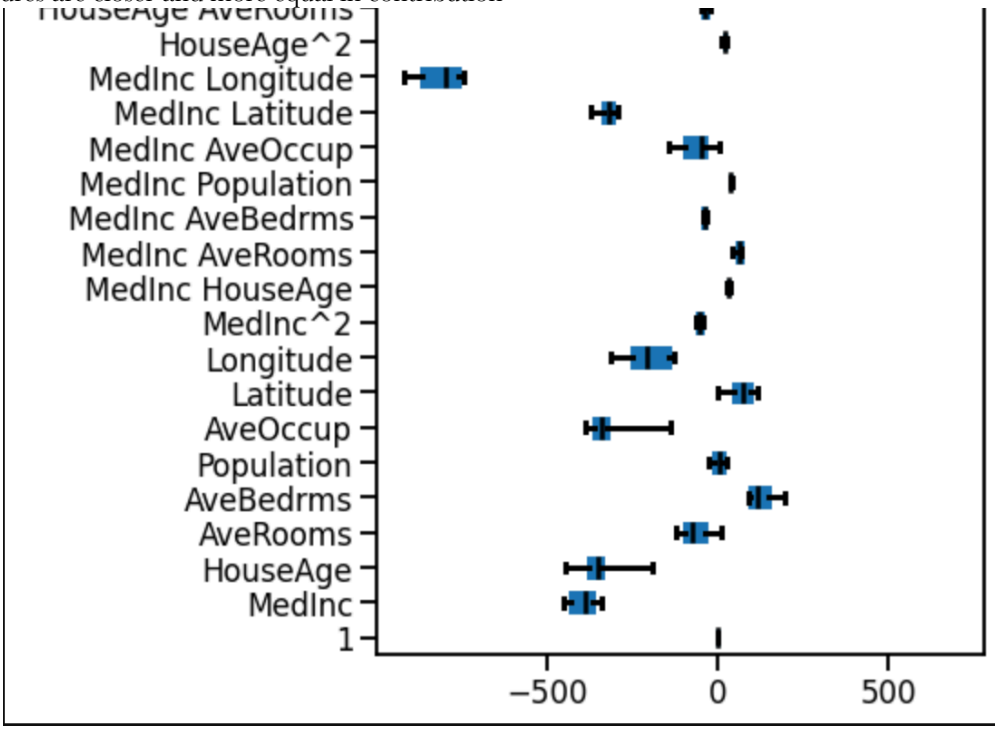
¶ Feature scaling and regularization

- Interaction of elements
  - ¬ Weights define link between:
    - ◊ feature values
    - ◊ predicted target
  - ¬ Regularization constrains:
    - ◊ weights
    - ◊ using alpha
  - ¬ Feature scaling effects:
    - ◊ weights
    - ◊ regularization
- Scaling
  - ¬ two equally important features on same scale
    - ◊ will be regularized similarly
  - ¬ different data scale example: age in years, annual revenue in dollars
    - ◊ boost features with small scale
    - ◊ reduce weights of high scale features
  - ¬ regularization forces weights closer together
    - ◊ rescaling forces data values closer together

- Note:
    - ¬ some solvers using gradients expect rescaled data
- Using scaling brings 'test_score' and 'train_score' closer together

```
ridge = make_pipeline(PolynomialFeatures(degree=2),
StandardScaler(),Ridge(alpha=0.5))
```

    - ◊ less overfitted
    - ◊ also better 'test_score'
- Features are closer and more equal in contribution



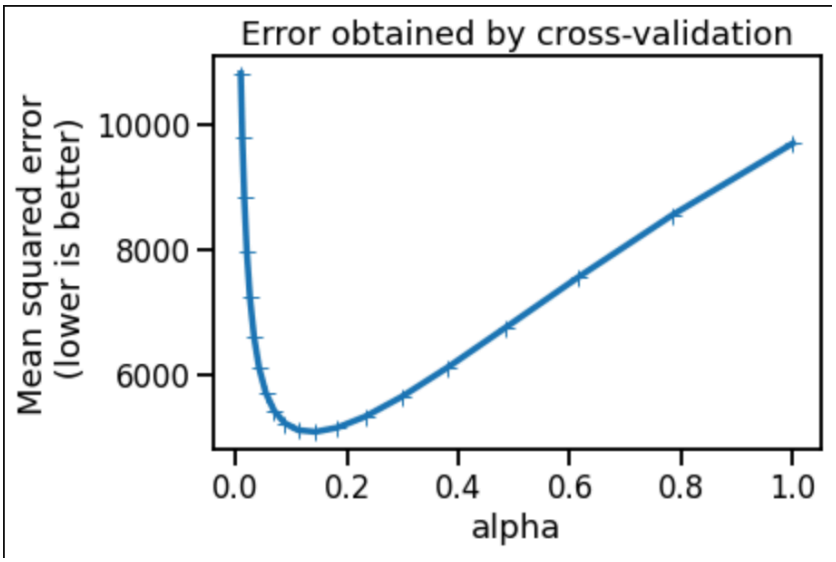
- Increasing alpha decreases weight values
    - ¬ Negative alpha enhances large weights and overfitting
- Scaling categorical features
    - ¬ common to omit them from scaling
    - ¬ but, when imbalanced (more samples in one category than another)
        - ◊ scaling can even out impact of regularization
    - ¬ rare categories are problematic

¶ Fine tuning the regularization parameter

- Out-of-sample rule
    - ¬ Care must be taken with training/testing, so that a chosen alpha is tested independently
    - ¬ cross-validation functions
- Nested cross validation
    - ¬ inner CV will search for the best alpha
    - ¬ outer CV will provide estimate of the testing score

```
import numpy as np
from sklearn.linear_model import RidgeCV
from sklearn.model_selection import ShuffleSplit

alphas = np.logspace(-2, 0, num=20)

ridge = make_pipeline(PolynomialFeatures(degree=2),
StandardScaler(),RidgeCV(alphas=alphas, store_cv_values=True))

cv = ShuffleSplit(n_splits=5, random_state=1)

cv_results = cross_validate(ridge, data, target,cv=cv,
scoring="neg_mean_squared_error", return_train_score=True,
return_estimator=True, n_jobs=2)
```

- ¬ With optimal alpha, training and testing are very close together
  - ◊ very litle overfitting



- ¬ Store the error found in cross-validation
  `store_cv_values=True`
- ¬ The graph shows the trade-off of too high vs too low alpha

• Now check the best alpha for cross-validation over each of the 5 shuffle splits
- ¬ The best alpha does wander somewhat between iterations
- ¬ Common practice is to use the average value alpha

§ 4.3.3 Exercise M4.04

• linear_models_ex_04.ipynb
• Create a regression dataset, where 2 out of 3 features are informative

```
from sklearn.datasets import make_regression

data, target, coef = make_regression(
    n_samples=2_000,
    n_features=5,
    n_informative=2,
    shuffle=False,
    coef=True,
    random_state=0,
    noise=30,
)
```

- Build a data set with two informative columns and three noise columns
  - ¬ Original coefficients:

```
Relevant feature #0     9.566665
Relevant feature #1    40.192077
Noisy feature #0        0.000000
Noisy feature #1        0.000000
Noisy feature #2        0.000000
```

  - ¬ then add two duplicates of the two columns
  - ¬ running linear regression on this produces massive coefficients for the informative columns

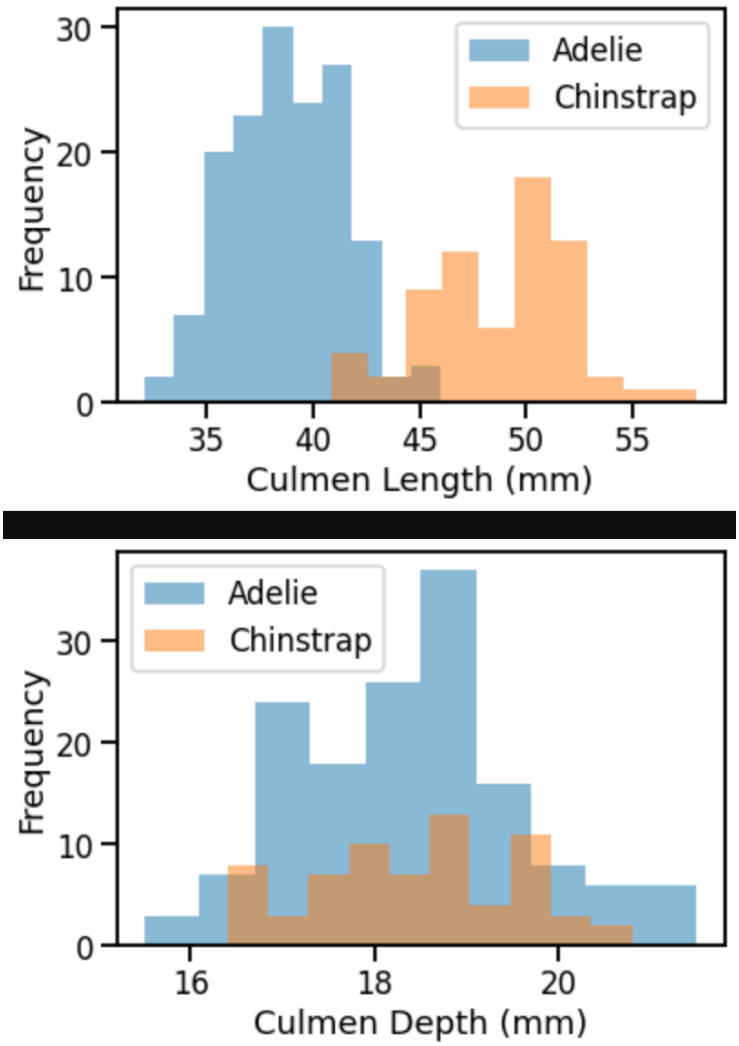
```
[ 1.31388260e+12 -1.59473665e+14 -1.99218750e-01 -1.68457031e-01
9.52148438e-02 -6.56941302e+11  4.12806902e+13 -6.56941302e+11
1.18192975e+14]
```

    - ◊ finding coefficients involves inverting the matrix `np.dot(data.T, data)`
      - † which is not possible (or creates massive errors)
  - ¬ running ridge on it applies penalty to the weights
    - ◊ matrix inverted is `np.dot(data.T, data) + alpha * I`
    - ◊ which is possible
  - ¬ coefficients end up 3 times smaller than original, because of info columns multiplied 3 times

```
[ 3.6313933  13.46802113 -0.20549345 -0.18929961  0.11117205
3.6313933 13.46802113  3.6313933  13.46802113]
```

§ 4.4.1 Linear model for classificaion

- linear_models_ex_04.ipynb
- Penguin dataset



◊ Classification problem
◊ Culmen Depth is not helpful

• Logistic function used to model the probability
¬ Logistic regression

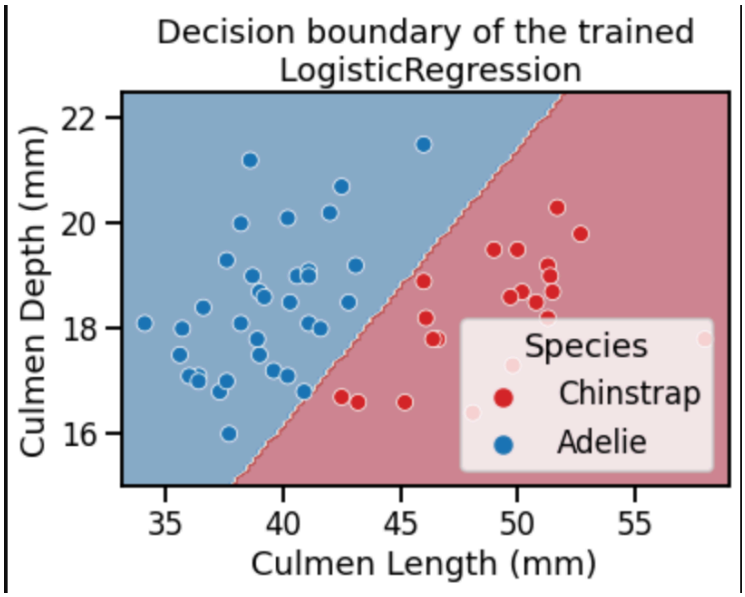
```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

logistic_regression = make_pipeline(
    StandardScaler(), LogisticRegression(penalty="none")
)
logistic_regression.fit(data_train, target_train)
accuracy = logistic_regression.score(data_test, target_test)
print(f"Accuracy on test set: {accuracy:.3f}")
```

◊ No regularization used:

```
penalty='none'
```

• With only two features, we can display a decision boundary
¬ DecisionBoundaryDisplay is currently a proposed pull-request

```
DecisionBoundaryDisplay.from_estimator(
    logistic_regression, data_test, response_method="predict",
cmap="RdBu_r", alpha=0.5
)
sns.scatterplot(
    data=penguins_test, x=culmen_columns[0],
y=culmen_columns[1],
    hue=target_column, palette=["tab:red", "tab:blue"])
_ = plt.title("Decision boundary of the trained\n
LogisticRegression")
```



◊ since the line is oblique (slanted), it means we used a combination of features
◊ Note that no regularization happened
`penalty='none'`

§ 4.4.2 Exercise M4.05

- linear_models_ex_05.ipynb
- Arguments for settinng regularization
  - ¬ Type of regularization
    `penalty`
    † default: `l2`
  - ¬ Strength of regularization
    `C`
    † penalty='none' is equivalent to setting C to infinity
    † in example, low C, or high regularization, makes the culmin width weight 0
    » this shows in a graph as a ine parallel to width axis

§ 4.4.3 Beyond linear separation in classificaiton

- logistic_regression_non_linear.ipynb
- NOTE: logistic regression is not a regression problem, it is a classification problem
- Similar methods available to address non-linear classification boundaries
  - ¬ feature augmentation with expert knowledge
  - ¬ kernel-based method
- Two different methods of generating generic example data sets

¬ Make moons
◊ two crescent-shaped classes hooked together

```
from sklearn.datasets import make_moons

feature_names = ["Feature #0", "Features #1"]
target_name = "class"

X, y = make_moons(n_samples=100, noise=0.13, random_state=42)
```

¬ Make Gaussian quantiles
◊ center class 1 ringed by class 2

```
from sklearn.datasets import make_gaussian_quantiles

feature_names = ["Feature #0", "Features #1"]
target_name = "class"

X, y = make_gaussian_quantiles(
    n_samples=100, n_features=2, n_classes=2, random_state=42)
```

• Radial basis function (RBF) kernel
¬ along with support vector machine classifier

```
kernel_model = make_pipeline(StandardScaler(), SVC(kernel="rbf",
                                    gamma=5))
```

• Overfitting risk
¬ adding flexibility to models to fit non-linear boundaries can increase overfitting
¬ makes decision function more sensitive to noisy data points

§ 4.4.1 Module 4 Quiz

• Select only numerical columns from dataset

```
data = adult_census.select_dtypes(["integer", "floating"])
```

• Import DummyClassifier
¬ set as constant, with constant argument
¬ set as most_frequent

```
from sklearn.dummy import DummyClassifier

# dumdum1 = DummyClassifier(strategy="constant", constant="
>50K")

dumdum2 = DummyClassifier(strategy="most_frequent")
cv_results6 = cross_validate(dumdum2, data, target, cv=10)
cv_results7 = cross_validate(dumdum2, data, target, cv=10,
scoring="balanced_accuracy")
```

§ 4.5.1 Wrap Up

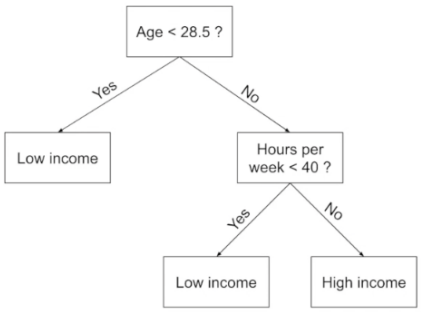
• Predictions of linear model depend on:
¬ weighted sum of values of the input features
¬ added to an intercept parameter
• Fitting a linear model consists of:
¬ adjusting weight coefficients
¬ adjusting intercept
◊ minimize predicion errors
• Scaling needed to bring features into same dynamic range
• Regularization can reduce overfitting
¬ weights are constrained to stay small
• Regularization hyperparameter
¬ tuned by CV for each new:

◊ problem
◊ dataset
• Feature engineering required for using linear models on non-linear relations
¬ otherwise underfitting occurs
◊ training score and test score are farther apart

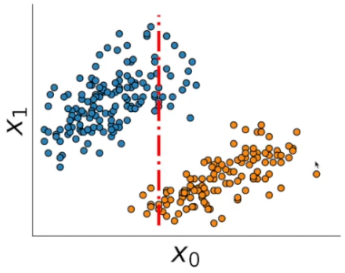
— MODULE 5: DECISION TREE MODELS

§ Intuitions on tree-based models

• https://www.youtube.com/watch?v=1kIHC1O_drM
• Oliver Grisel
• Decision Trees
Often the best performing with tabular data
What is a decision tree
Income classification in the US

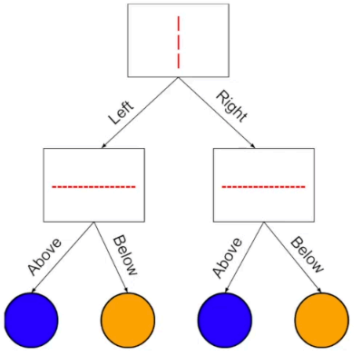


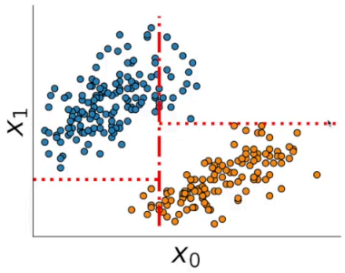
hierarchical decisions based on one variable at a time
reaching the leaf gives the answer
Growing a classification tree
Two groups of data points
predict their color



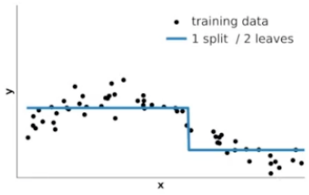
Root node of the decision tree
The red line in the graph
partition data into everything on the left and everything on the right
left, predict bluc, right predict orange
Or, predict class probabilities
count numbers of blue and orange on left, divide the total number on the left
find probability of being blue or orange on the left
After the first split, new decision node
Horisontal split along x1 axis
Now 100% accurate for this split

Another node, with another horizontal split
Again, 100% accurate

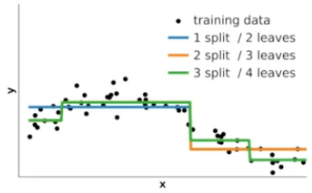




Regression with a decision tree
Root node, decision node, split
Split on specific value of x
average value of y on the left
average value of x on the right
Constant piecewise is only loosly approximate
(trying a linear model would have underfit as well_

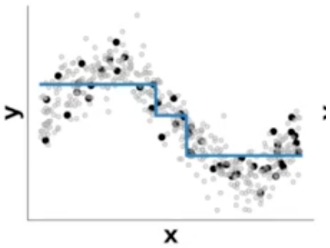


Second split, two additional subgroups

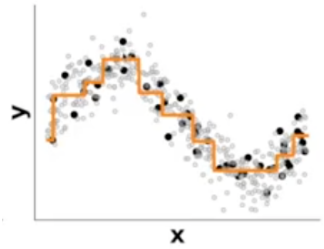


Each new split, gives a new node
Green line starts to much more closely fitting the actual data

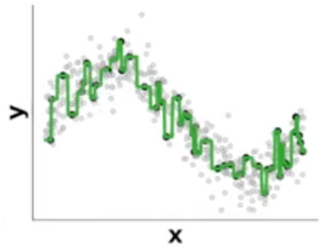
Tree Depth and Overfitting
Too deep will hurt generalization
Serious underfit on the left, both below and above
`max_depth/max_leaf_nodes` is too small



Test and train error is approximately the same
no significant underfitting



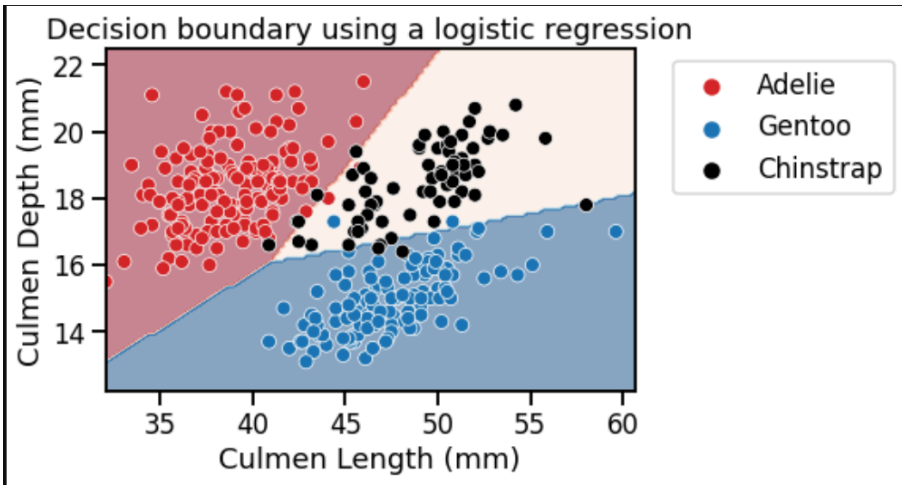
Eventually all the training points could be fit to their own leaf
- but this causes overfit
- Exactly zero training error
- But the error for the grey data point prediction is worse than the example before
  - `max_depth/max_leaf_nodes` is too large



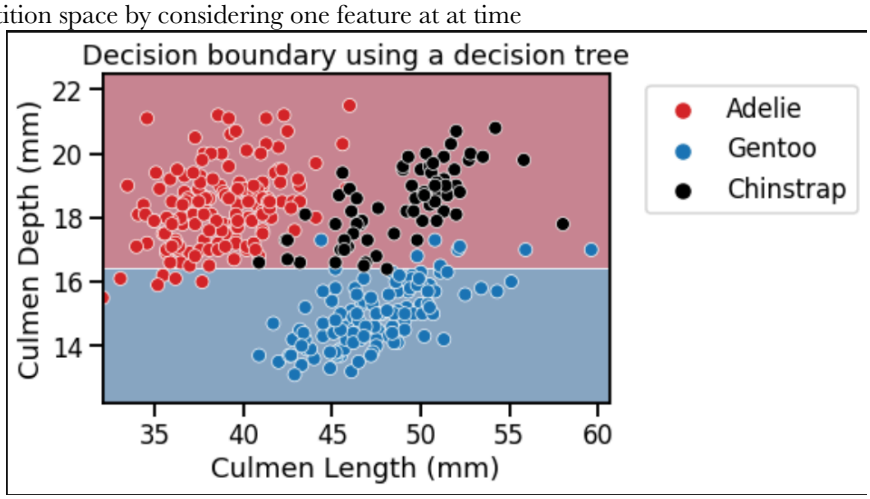
Take home:
- Sequence of simple decision rules
  - one feature and one threshold at a time
- No scaling required for numerical features
  - good for tabular data
  - Decision exactly the same if scaled or not
- `max_depth` controls trade-off between underfitting and overfitting
- Mostly useful as a building block for ensemble model
  - Random Forests
  - Gradient Boosting Decision Trees
    - Not very good by themselves, though

§ 5.1.1 **Build a classification decision tree**

- trees_classification.ipynb
- Linear classifier
  - ¬ defines linear separation to split classes
  - ¬ uses linear combination of input features
    - ◊ displays as non-perpendicular to a specific axis



• Decision trees are non-parametric models
  ¬ no mathematical decision function
  ¬ no weights or intercept
• Partition space by considering one feature at at time



  ◊ discards the feature culmin length
    † decision trees do not use a combination of features to make a split

```
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier(max_depth=1)
tree.fit(data_train, target_train)
```

• Tree structure

```
from sklearn.tree import plot_tree

_, ax = plt.subplots(figsize=(8, 6))
_ = plot_tree(tree, feature_names=culmen_columns,
              class_names=tree.classes_, impurity=False, ax=ax)
```

  ¬ `plt.subplots` creates a figure and axis
    ◊ then pass the axis to the `plot_tree` function for drawing

• **criterion**
  ¬ Partitions minimize the class diversity in each sub-partition
  ¬ settable parameter
  ¬ classifier will predict the most represented class within a partition
    ◊ eg, Adelie

- Counts in the partition are also taken during training
    - ◊ Data extraction and graph

```
y_pred_proba = tree.predict_proba(sample_2)
y_proba_class_0 = pd.Series(y_pred_proba[0],
index=tree.classes_)

y_proba_class_0.plot.bar()
plt.ylabel("Probability")
_ = plt.title("Probability to belong to a penguin class")
```

    - ◊ Calculations can also be done manually with the data counts from the table
- Note: culmen length has been disregarded for the moment
    - ¬ not used in prediction
- Scoring

```
test_score = tree.score(data_test, target_test)
print(f"Accuracy of the DecisionTreeClassifier: {test_score:.2f}")
```

§ 5.2.1 Decision tree for regression

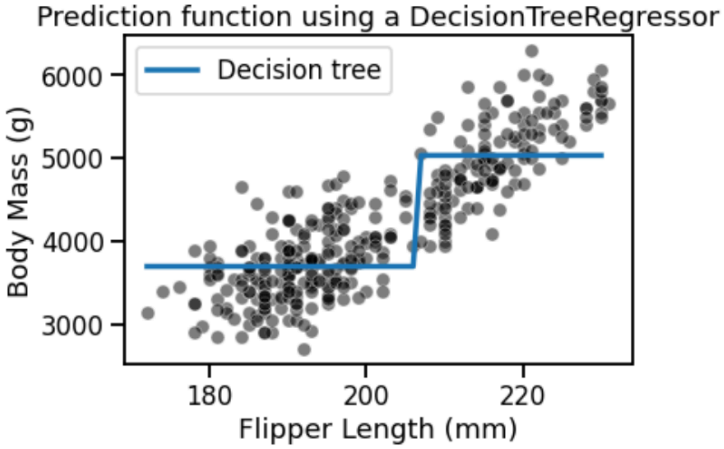
- trees_regression.ipynb
- Synthetic dataset
    - ¬ min and max are the same as the actual flipper length column
    - ¬ numpy arange then fills in evenly spaced data, defaulted to increments of 1

```
import numpy as np
data_test = pd.DataFrame(np.arange(
                         data_train[feature_name].min(),
                         data_train[feature_name].max()),
                         columns=[feature_name])
```

- Dashed line for `plt.plot`

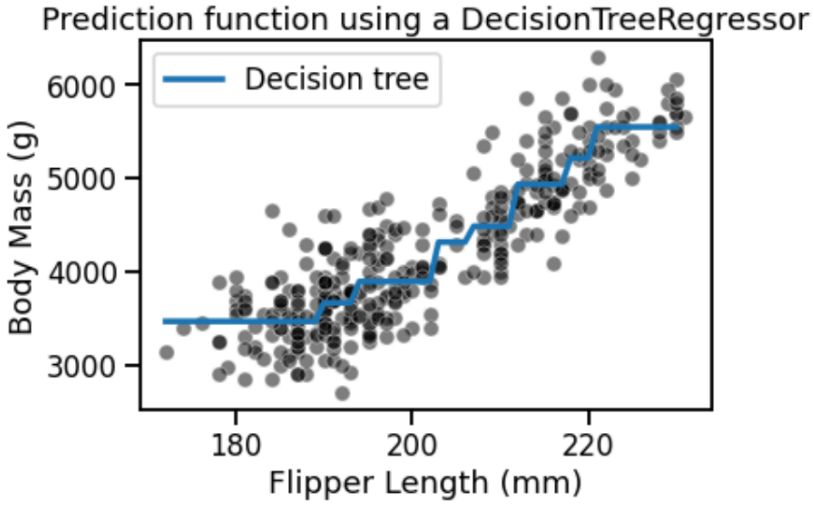
```
linestyle="--"
```

- Non-parametric model
    - ¬ Do not make assumptions about how the data is distributed
    - ¬ Predictions are piecewise constant
    - ¬ Feature space divided into two partitions



    - ¬ Threshold for flipper length is 206.5mm
        - ◊ vertical line
    - ¬ Predicted values on each side of the split are two constants
        - ◊ 3683.50 g and 5023.62 g
        - ◊ values which correspond to mean values of training samples in each partition
- Increased depth

¬ Increases the number of partitions
¬ And so increases the number of constant values that the tree is capable of predicting`

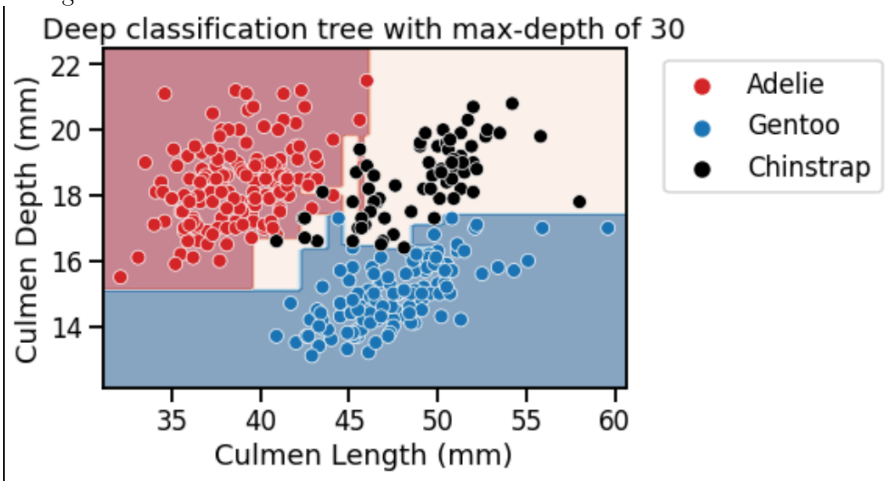


§ 5.3.1 Importance of decision tree hyperparameters on generalization

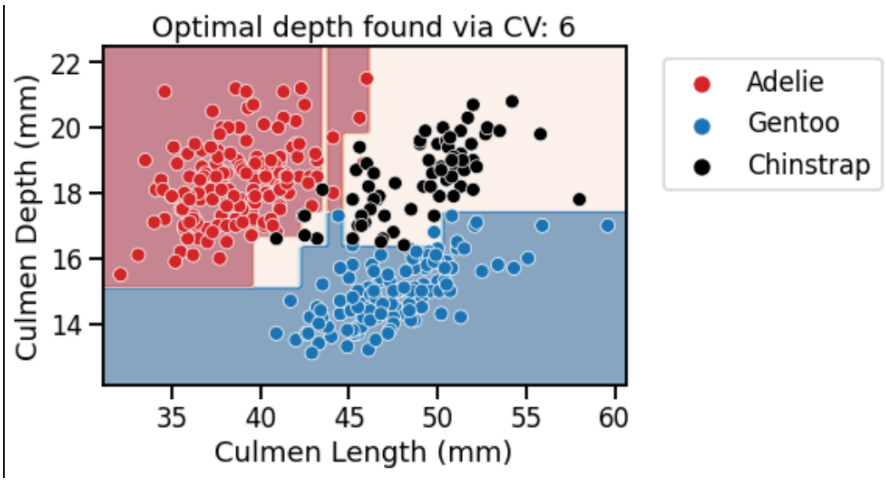
• trees_hyperparameters.ipynb

¶ Effect of max_depth parameter

• Overfitting



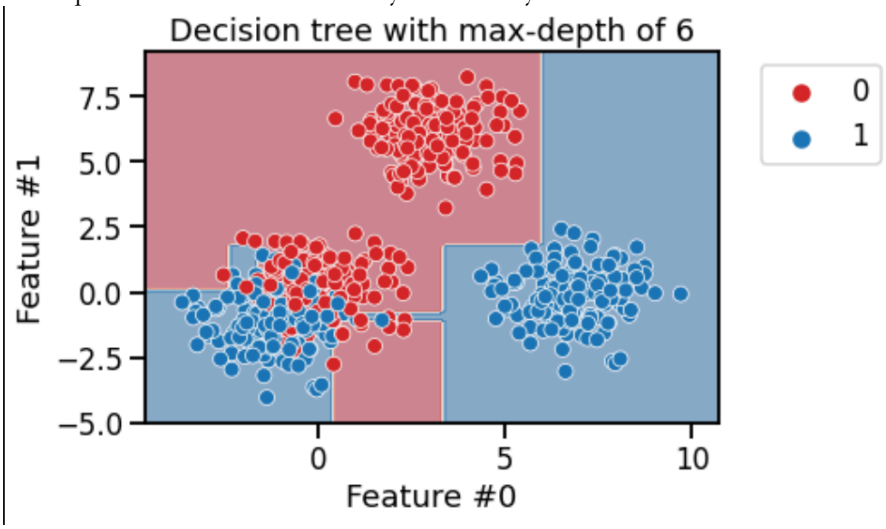
• Optimal??

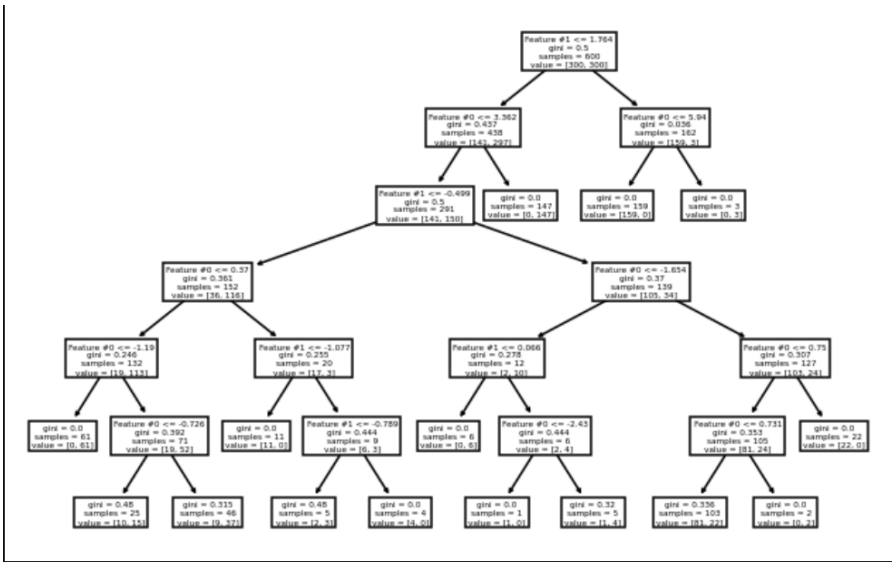


- Hyperparameter must be optimed for each application
  - ¬ no single value optimal for any dataset

¶ Other hyperparemters in decision trees

- `max_depth` controls overall complexity of the tree
  - ¬ However, asymmetric development could also factor into overfitting
- Example creates a dataset where:
  - ¬ one portion is clearly segregated
  - ¬ one portion is mixed and need asymetric analysis



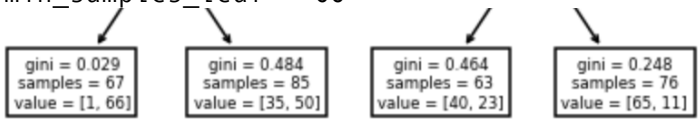


¬ Fixing the `max_depth` parameter would cut the tree horizontally somewhere
◊ even if more growth would be beneficial

• Asymetric hyperparameters

`min_samples_leaf` Will stop if next leaf split covers less than min
`min_samples_split`
`max_leaf_nodes`
`min_impurity_decrease`

¬ `min_samples_leaf = 60`



§ Wrap up

- are suited for both regression and classification problems;
- are non-parametric models;
- are not able to extrapolate;
- are sensitive to hyperparameter tuning.

— MODULE 6: ENSEMBLE OF MODELS

§ Contents

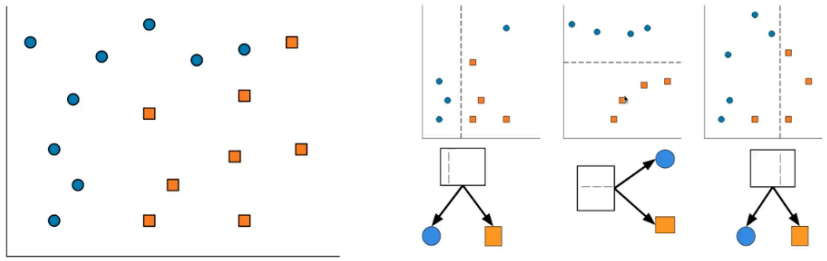
¬ Gradient Boosting

§ Intuitions on ensemble models

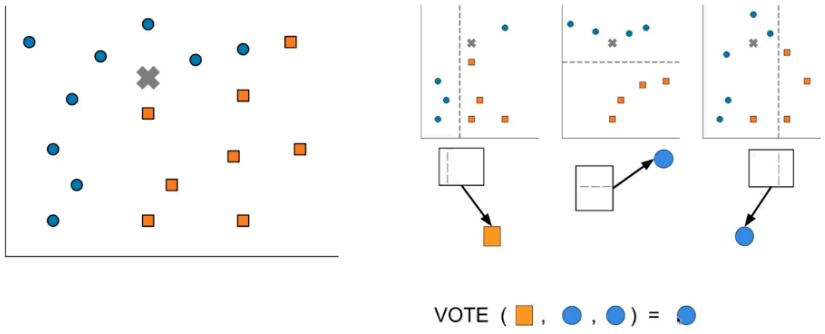
- https://www.youtube.com/watch?v=SnvdnIlOsHQ
- Ensemble of tree-based models
  - Often over or underfit
- Bagging
  - Random Forests
- Boosting
  - Gradient Boosting
- Bagging (Bootstrap Aggregating)
  - Two steps
    - Boosting
    - Aggregation

Take a random subset of the data points
- The bootstrap samples overlap
- each data in the training set will appear many times

1. Bootstrap
   - fit an independent model on each sample
   - we will have three different model for three samples
   - just a depth of one (in practice we use very deep trees)



2. Aggregation
   - first decison tree, predict on the majority side
   - eg: one orange and two blue votes
   - the resulting vote is of the ensesmble is blue



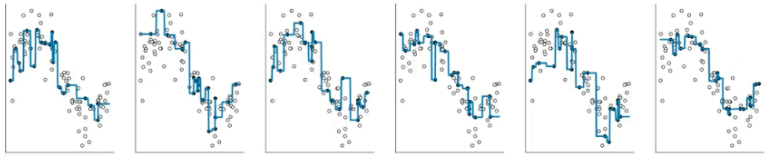

```
BaggingClassifier
RandomForestClassifier
```

Typical decision is higher quality than the individual trees
- even if individual trees overfit, average will overfit less

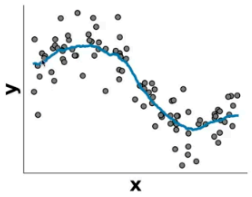
Eg 2: Regression
- Non-linear relationship

Good amount of noise

1. Take multiple random subsets
   enough that all the points will eventually be represented
2. Fit model perfectly
   models will all overfit
   but each one overfits differently
3. Average prediction does not overfit



- Select multiple random subsets of the data
- Fit one model on each
- Average predictions



• Bagging vs Random Forests
   Bagging is a general strategy for any base model
      linear, trees, etc...
   Random forests are bagged randomized decision trees
      want to decorrelate the prediction errors of the individual trees
      additional randomization step
      when considering a split
         instead of considering all the features to select the best feature to split on
         we randomly take a subsample of the columns (features)
         calculate best split for this random subsample
         when decision trees are trained this way
            we inject noisy constraint in the training procedure
            extra randomization decorrelates the prediction error
            make the averaging step of the bagging work better
      uncorrelated errors make bagging work better

Let each deep tree overfit
Bagging ensemble averaging will not overfit

§ 6.1.1 Introductory example to ensemble models

• ensemble_introduction.ipynb
• Bagging ensemble method
   ¬ 20 decision trees

```
%%time
from sklearn.ensemble import BaggingRegressor

base_estimator = DecisionTreeRegressor(random_state=0)
bagging_regressor = BaggingRegressor(
    base_estimator=base_estimator, n_estimators=20,
random_state=0)

cv_results = cross_validate(bagging_regressor, data, target,
n_jobs=2)
scores = cv_results["test_score"]

print(f"R2 score obtained by cross-validation: "
      f"{scores.mean():.3f} +/- {scores.std():.3f}")
```

◊ Better generalization performance than single, tuned tree and faster

§ 6.1.2 Bagging

- ensemble_bagging.ipynb
- Bootstrap Aggregating
  - ¬ **bootstrap resampling**
    - ◊ random sampling with replacement
      - † meaning points can be sampled multiple times (vs. without replacement)

¶ Bootstrap resampling

- Random sampling with replacement
  - ◊ will have some data points multiple times, and some not at all
  - ◊ resample will be the same size as the original

```
rng = np.random.RandomState(1)

def bootstrap_sample(data, target):
    # Indices corresponding to a sampling with replacement of
the same sample
    # size than the original data
    bootstrap_indices = rng.choice(
        np.arange(target.shape[0]), size=target.shape[0],
replace=True,
    )
    # In pandas, we need to use `.iloc` to extract rows using an
integer
    # position index:
    data_bootstrap = data.iloc[bootstrap_indices]
    target_bootstrap = target.iloc[bootstrap_indices]
    return data_bootstrap, target_bootstrap
```

¬ For example, from a dataset with 30 rows, the following indices are sampled:

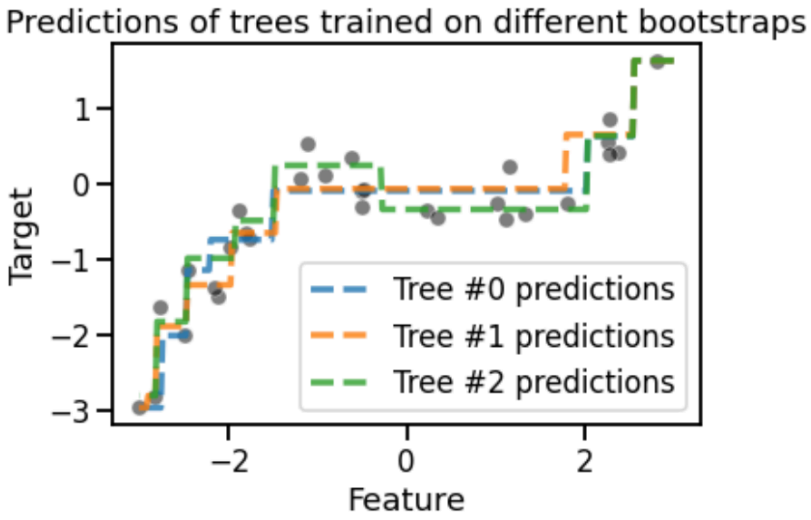
```
Bootstrap indices: [20 21  6  2 12 27 21 11  7 13  8 11
                    12 11 20  4  7  7 13  4 25 16 28 18
```

¬ Number of unique samples:

~63.2% of the original data is present

~36.8% are repeated samples

¬ Generating many datasets, each slightly different



¶ Aggregating

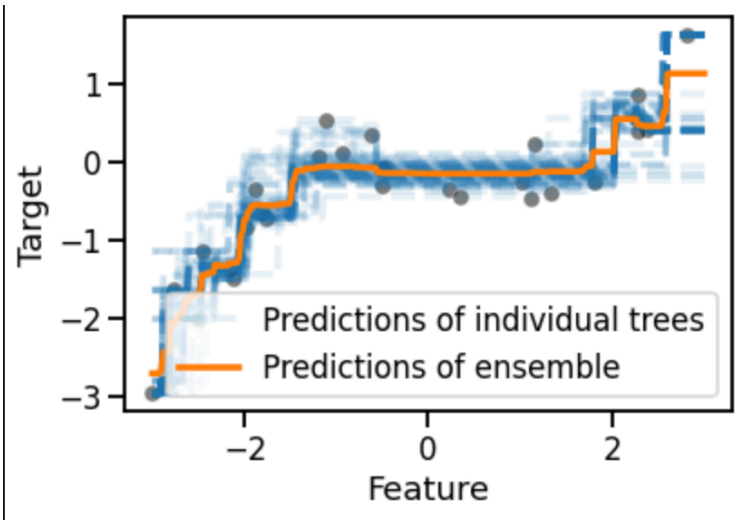
- Average the predictions
  - ¬ for a given test point:
    - ◊ feed input feature to each model
    - ◊ take all prediction per test point and average
  - ¬ less likely to overfit

¶ Bagging in scikit-learn

- **meta-estimator**
  - ¬ estimator that wraps another estimator
  - ¬ takes a base model, clones several times, trains on different resamples

```
from sklearn.ensemble import BaggingRegressor

bagged_trees = BaggingRegressor(
    base_estimator=DecisionTreeRegressor(max_depth=3),
    n_estimators=100,
)
_ = bagged_trees.fit(data_train, target_train)
```

  - ¬ Internal models can be accessed in list after fitting
    `bagged_trees.estimators_`
  - ¬ low `alpha` opacity in the scatterplot shows the iterations




```
for tree_idx, tree in enumerate(bagged_trees.estimators_):
    label = "Predictions of individual trees" if tree_idx == 0
else None
    # we convert `data_test` into a NumPy array to avoid a
warning raised in scikit-learn
    tree_predictions = tree.predict(data_test.to_numpy())
    plt.plot(data_test["Feature"], tree_predictions,
linestyle="--", alpha=0.1,
             color="tab:blue", label=label)

sns.scatterplot(x=data_train["Feature"], y=target_train,
color="black",
                alpha=0.5)

bagged_trees_predictions = bagged_trees.predict(data_test)
plt.plot(data_test["Feature"], bagged_trees_predictions,
         color="tab:orange", label="Predictions of ensemble")
_ = plt.legend()
```

¶ Bagging complex pipelines

- Bagged polynomal regression pipeline

```
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import make_pipeline


polynomial_regressor = make_pipeline(
    MinMaxScaler(),
    PolynomialFeatures(degree=4),
    Ridge(alpha=1e-10),
)
```

  - ¬ Pipline processing:
    - ◊ Scales the data to the 0-1 range
      `MinMaxScaler`
    - ◊ Extracts degree-4 polynomial features
      - † features will all be in the 0-1 range
        - » if x is 0-1, then x^n must als be in 0-1 for any n value
    - ◊ `alpha` is made small, because bagging is expected to work well

```
bagging = BaggingRegressor(
    base_estimator=polynomial_regressor,
    n_estimators=100,
    random_state=0,
)
_ = bagging.fit(data_train, target_train)
```

¬ Pipeline is then passed to `BaggingRegressor`

¶ Bagged polynomial regression looks better than the bagged trees

¬ but we know that the example data was generated with a polynomial

§ 6.2.3 Exercise M6.01

- ensemble_ex_01.ipynb
- Why this range of features??

```
from sklearn.model_selection import RandomizedSearchCV
param_dist = {
    'max_features': [.5, .8, 1.0],
    'n_estimators': randint(10, 30),
    'max_samples': [.5, .8, 1.0],
    'base_estimator__max_depth': randint(3, 10)}
model_randomCV = RandomizedSearchCV(bagged_trees,
param_distributions=param_dist,  n_iter=20,
scoring="neg_mean_absolute_error")
```

- Produces a random number somehow...

```
from scipy.stats import randint
randint(10, 30)
```

§ 6.2.3 Random forests

- ensemble_random_forest.ipynb
- Modification of Bagging
- Only decision trees for random forests
  - ¬ whereas bagging is for any classifier or regressor
- The search for the best split is done only on a subset of the original features taken at random
  - ¬ different random subsets for each split node
  - ¬ injects additional randomization into learning to decorrelate prediction errors of individual trees
    - ◊ Randomization occurs on both axes of the data matrix
      - † bootstrapping samples at each tree in the forest
      - † randomly selecting subset of features at each node

¶ A look at random forests

- `OrdinalEncoder` will work fine with trees, even with no true categorical rankings
- Rare categories
  - ¬ specifically encode unknown categories at prediction time
  - ¬ otherwise, rare categories might only be present on the validation side of CV split
    - ◊ `OrdinalEncoder` will raise an error
- Alternate column transformer

```
from sklearn.preprocessing import OrdinalEncoder
from sklearn.compose import make_column_transformer,
make_column_selector

categorical_encoder = OrdinalEncoder(
    handle_unknown="use_encoded_value", unknown_value=-1
)
preprocessor = make_column_transformer(
    (categorical_encoder,
make_column_selector(dtype_include=object)),
    remainder="passthrough"
)
```

- Bagging example

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bagged_trees = make_pipeline(
    preprocessor,
    BaggingClassifier(

base_estimator=DecisionTreeClassifier(random_state=0),
        n_estimators=50, n_jobs=2, random_state=0,
    )
)
```

- Improvement of random forest over bagged trees
  - ¬ Possibly due to decorrelated prediction errors of individual trees
  - ¬ Makes averaging step more efficient at reducing overfitting

```
from sklearn.ensemble import RandomForestClassifier

random_forest = make_pipeline(
    preprocessor,
    RandomForestClassifier(n_estimators=50, n_jobs=2,
random_state=0)
)
```

- Then cross-validate to check it

```
from sklearn.model_selection import cross_val_score
scores_random_forest = cross_val_score(random_forest, data,
target)
```

¶ Details about default hyperparameters

- Amount of randomness in each split can be controlled

`max_features`

  - ◊ 0.5 means 50% of the features are considered at each split
  - ◊ 1.0 would disable feature subsampling

`RandomForestRegressor`

  - ◊ disables feature subsampling by default

`RandomForestClassifier`

`max_features=np.sqrt(n_features)`

  - ◊ these reflect good practice from scientific literature

- Tuning max_features
  - ¬ too much randomness can underfit base models
  - ¬ too few randomness gives more correlated prediction errors, and lessens averaging benefit
- Bagging also has `max_features` parameter
  - ¬ but because they're base model isn't always a tree

¬ they can only randomly subsample features once before fitting each base model
◊ rather than each time when adding tree splits

Ensemble model class	Base model class	Default value for `max_features`	Features subsampling strategy
`BaggingClassifier`	User specified (flexible)	`n_features` (no subsampling)	Model level
`RandomForestClassifier`	`DecisionTreeClassifier`	`sqrt(n_features)`	Tree node level
`BaggingRegressor`	User specified (flexible)	`n_features` (no subsampling)	Model level
`RandomForestRegressor`	`DecisionTreeRegressor`	`n_features` (no subsampling)	Tree node level

§ 6.2.4 Exercise M6.02

- ensemble_ex_02.ipynb
- Extracting individual estimators from a forest
  - ¬ and plotting them on regularly spaced intervals

```
import numpy as np

data_range = pd.DataFrame(np.linspace(170, 235, num=300),
                          columns=data.columns)
tree_predictions = []
for tree in random_forest['randomforestregressor'].estimators_:
    # we convert `data_range` into a NumPy array to avoid a
warning raised in scikit-learn
    tree_predictions.append(tree.predict(data_range.to_numpy()))

forest_predictions = random_forest.predict(data_range)



import matplotlib.pyplot as plt
import seaborn as sns

sns.scatterplot(data=penguins, x=feature_name, y=target_name,
                color="black", alpha=0.5)

# plot tree predictions
for tree_idx, predictions in enumerate(tree_predictions):
    plt.plot(data_range[feature_name], predictions, label=f"Tree
#{tree_idx}",
             linestyle="--", alpha=0.8)

plt.plot(data_range[feature_name], forest_predictions,
label=f"Random forest")
_ = plt.legend(bbox_to_anchor=(1.05, 0.8), loc="upper left")
```

§ 6.3.1 Intuitions on ensemble models

- https://www.youtube.com/watch?v=k2ZCG1OLjVM&feature=youtu.be

• Boosting and Gradient Boosting
• Boosting for classification

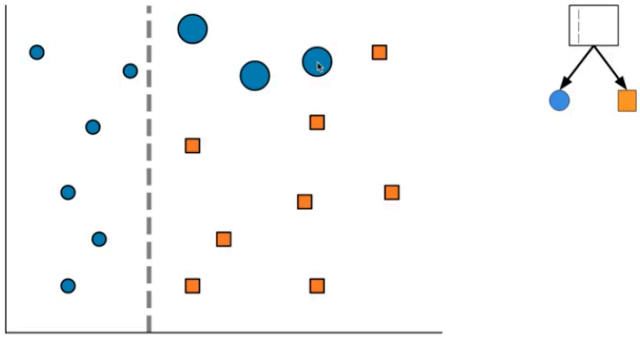
Eg

two features, classification task

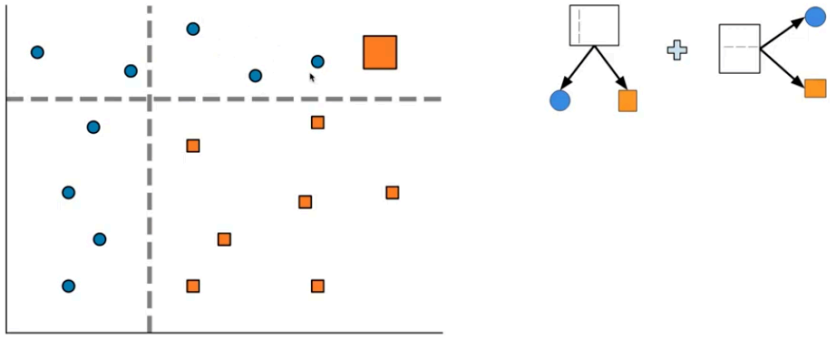
instead of sampling randomly, consider full training set

set a shallow model over the whole set

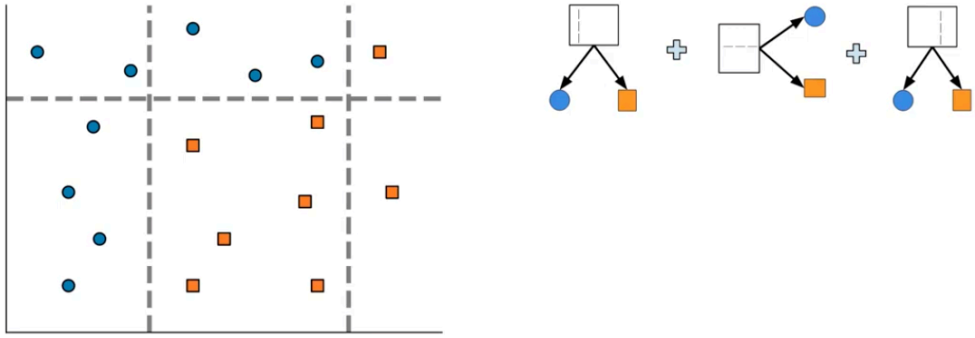
prediction errors will guide us to reweight to correct for those samples in error



Again, another error appears, put more weight on it



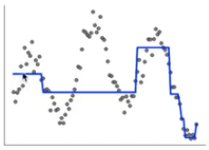
Next one ›



Trained iteratively, sequentially

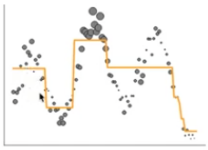
• Boosting for Regression

Underfitting regression model over the whole set

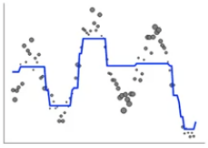


Then try to predict better

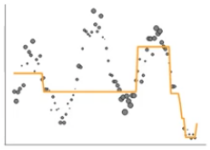
More weight put on the samples farthest away from the prediction function



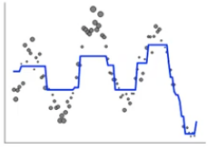
Then combine prediction functions together



Fit a new prediction with the altered weights



And then combine in the ensemble



- Boosting vs Gradient Boosting
  - AdaBoost
    - using reweighted learning procedure at each step
    - use any base model that accepts `sample_weight`
    - not recommended in practice anymore
  - Gradient Boosting
    - for focusing on decsion trees
    - more computationally efficient, but specialized
    - Histogram Gradient Boosting
      - more complex to explain
- Gradient Boosting and binned features
  - GradientBoostingClassifier
    - implementation of traditional (exact) method
    - fine for small data sets
    - too slow for samples > 10,000
  - HistGradientBoostingClassifier
    - preprocesses numerical features (256 levels, or binning)
    - efficient multi core implementation
    - computing histograms to approximate best split decision trees

much, much faster

- Take away

Bagging	Boosting
fit trees independently	fit trees sequentially
each deep tree overfits	each shallow tree underfits
averaging the tree predictions reduces overfitting	sequentially adding trees reduces underfitting

Gradient boosting tends to perform slightly better than bagging and random forest. Furthermore, shallow trees predict faster.

¬ In practice, Gradient Boosting works better than bagging or forests
because the trees are shallow, they tend to run faster
whereas the individual trees of bagging are more expensive
prediction accuracy is also slightly better

§ 6.3.2 Adaptive Boosting (AdaBoost)

- ensemble_adaboost.ipynb
- What does this do?????????

```
np.flatnonzero(
```

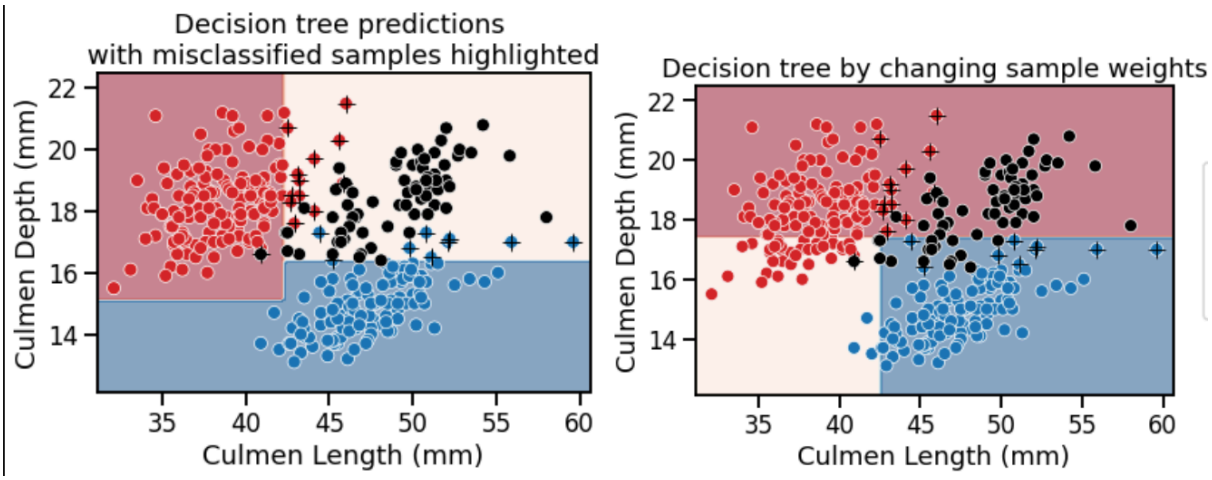
- Returning a dataframe with only misclassified data

```
misclassified_samples_idx = np.flatnonzero(
                                        target != target_predicted)
data_misclassified = data.iloc[misclassified_samples_idx]
```

- Create a new classifier
  - ¬ discard all correctly classified samples
  - ¬ only consider misclassified
    - ◊ misclassified assigned weight of 1, well classified a weight of 0

```
sample_weight = np.zeros_like(target, dtype=int)
sample_weight[misclassified_samples_idx] = 1

tree = DecisionTreeClassifier(max_depth=2, random_state=0)
tree.fit(data, target, sample_weight=sample_weight)
```

◊ the new classification function now correctly classifies previous errors



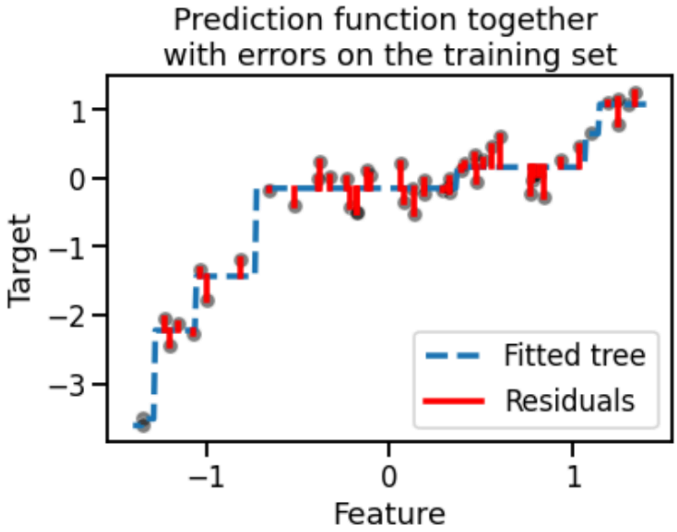
- Check overlap in numpy arrays:

```
remaining_misclassified_samples_idx =
    np.intersect1d(misclassified_samples_idx,
                   newly_misclassified_samples_idx)
```

- Weight predictions based on number of mistakes that each model is making
  - ¬ mistakes now being made on samples that were correct before
  - ¬ use classification error
  - ¬ Eg: classifier accuracy
    - ◊ first one: 94%
    - ◊ second one: 68%
  - ¬ use these percentages to weight the predictions of each model
- Boosting stragegy
  - ¬ method to compute weights to be assigned to samples
  - ¬ method for assigning weight to each learner for predictions
- AdaBoost
  - ¬ good demonstration of boosting algorithms
  - ¬ but not as efficient as gradient-boosting

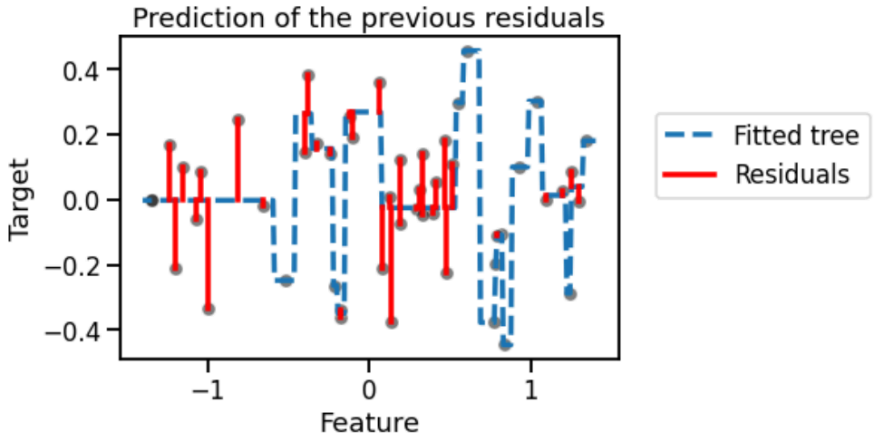
§ 6.3.3 Gradient-boosting decision tree (GBDT)

- ensemble_gradient_boosting.ipynb
- Instead of assigning weights to samples
  - ¬ fits on the residuals error
    - ◊ hence, "gradient"
- Each new tree predicts the error from the previous model
  - ¬ rather than predicting the target
- **Residuals**
  - ¬ error between predictions and the data



¬ Now create a tree that uses `data` to predict residuals instead of `target`

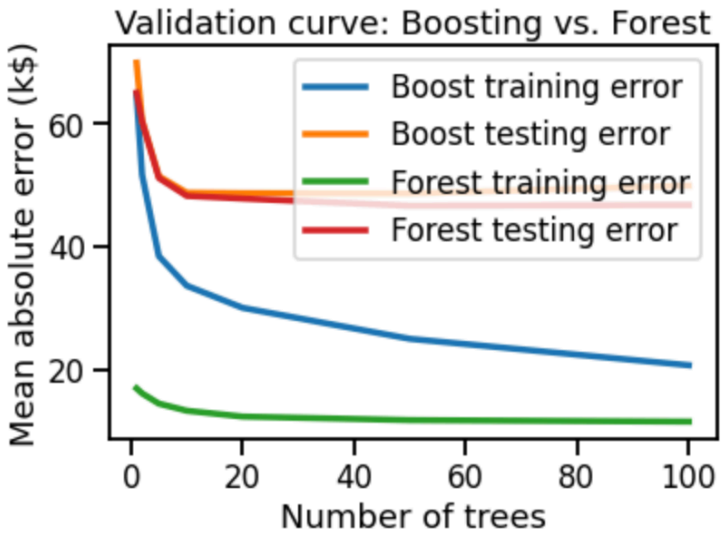
```
residuals = target_train - target_train_predicted
tree_residuals = DecisionTreeRegressor(
                                    max_depth=5, random_state=0)
tree_residuals.fit(data_train, residuals)
```



• Once the exact residual for a point can be accurately predicted
  ¬ point can be predicted by adding the predictions of all the trees
    ◊ the second tree corrects the first, the third corrects the second

§ 6.3.4 <u>Exercise M6.03</u>

• ensemble_ex_03.ipynb
• Both models always improve with more trees
  ¬ but plateau while training time keeps increasing



- Gradient-boosting offers early-stopping option
  - ¬ algorithm uses out-of-sample set to compute generalization performance for each new tree
  - ¬ if performance stops improving, it stops adding trees

```
gradient_boosting2 = GradientBoostingRegressor(
                        n_estimators=1000, n_iter_no_change=5)
```

- Mean absolute error for the whole train/test split is lower than the CV error
  - ¬ The train/test has more overall training data than the CV

§ 6.3.5 Speeding-up gradient-boosting

- ensemble_hist_gradient_boosting.ipynb
- Histogram gradient boosting
  - ¬ uses reduced number of splits
- Random forest efficiency
  - ¬ each tree can be fitted independently
  - ¬ algorithm scales efficiently with number of cores and number of samples
- Gradient boosting is sequential
  - ¬ computationally expensive
  - ¬ most expensive part is search for best split
    - ◊ brute-force approach
    - ◊ all possible splits evaluated, best is chosen
    - ◊ see "tree in depth" notebook
- Accelerating gradient-boosting
  - ¬ reduce number of splits
  - ¬ reduces performance
  - ¬ but add more estimators to compensate
- Discretizer
  - ¬ Reduce number of splits by binning data before gradient boosting

```
import numpy as np
from sklearn.preprocessing import KBinsDiscretizer

discretizer = KBinsDiscretizer(
    n_bins=256, encode="ordinal", strategy="quantile")
data_trans = discretizer.fit_transform(data)
data_trans
```

    - ◊ Transforms data into integral values
    - ◊ value represents the bin index

`n_bins=256`
» at most 256 bins
◊ warnings appear if too many bins make bins which are too small
† too small bins are removed

• Fit time is reduced, though performance is the same

```
print(f"Mean absolute error via cross-validation: "
      f"{-cv_results_gbdt['test_score'].mean():.3f} +/- "
      f"{cv_results_gbdt['test_score'].std():.3f} k$")
print(f"Average fit time: "
      f"{cv_results_gbdt['fit_time'].mean():.3f} seconds")
print(f"Average score time: "
      f"{cv_results_gbdt['score_time'].mean():.3f} seconds")
```

• HistGradientBoosting is even more optimized for large datasets
¬ Each feature is binned
¬ especially advantageous over 10,000 samples

```
from sklearn.ensemble import HistGradientBoostingRegressor
histogram_gradient_boosting = HistGradientBoostingRegressor(
    max_iter=200, random_state=0, early_stopping=True)
```

§ 6.4.1 Hyperparameter tuning

• ensemble_hyperparameters.ipynb

¶ Random forest

• `n_estimators` is the main parameter to tune
¬ More trees, better generalization performance
¬ but will slow down fitting and prediction time
◊ goal is balance between performance and time

• Depth of each tree in forest also an important hyperparameter
`max_depth`
◊ enforces more symmetric tree
`max_leaf_nodes`
◊ asymmetry ok

• Deep trees are typical, since overfit on individuals is desired

• Use a randomized search to check the relationship between n_estimators and max_leaf_nodes

```
import pandas as pd
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestRegressor

param_distributions = {
    "n_estimators": [1, 2, 5, 10, 20, 50, 100, 200, 500],
    "max_leaf_nodes": [2, 5, 10, 20, 50, 100],
}
search_cv = RandomizedSearchCV(
    RandomForestRegressor(n_jobs=2),
param_distributions=param_distributions,
    scoring="neg_mean_absolute_error", n_iter=10,
random_state=0, n_jobs=2,
)

cv_results = pd.DataFrame(search_cv.cv_results_)
```

¬ Results show better improvement from larger number of leaves
◊ especially once number of trees gets to at least 50 trees

• Scoring the best model

```
search_cv.fit(data_train, target_train)
error = -search_cv.score(data_test, target_test)
```

¶ Gradient-boosting decision trees

- Parameters are coupled in gradient-boosting
  - ¬ cannot set parameters one after another
- Most important are
  - `n_estimators`
  - `learning_rate`
  - `max_depth` or `max_leaf_nodes`
- For max_depth, fitting full grown trees would be detrimental
  - ¬ low depth, 3-8 typically
  - ¬ or few leaves, 2^3 to 2^8 typically
  - ¬ weak learners at each step helps reduce overfitting
- In turn, n_estimators should be increased if max_depth is low
  - ¬ as before, using early-stopping to prevent unnecessary iterations is best
- learning_rate changes how many errors are corrected per each tree iteration
  - ¬ small rate only corrects few samples
    - ◊ need more estimators overall
    - ◊ takes more time
  - ¬ large learning rate can possibly create overfitted ensemble
- Use Randomized Search to check a range of parameter options

```
from scipy.stats import loguniform
from sklearn.ensemble import GradientBoostingRegressor

param_distributions = {
    "n_estimators": [1, 2, 5, 10, 20, 50, 100, 200, 500],
    "max_leaf_nodes": [2, 5, 10, 20, 50, 100],
    "learning_rate": loguniform(0.01, 1),
}
search_cv = RandomizedSearchCV(
    GradientBoostingRegressor(),
param_distributions=param_distributions,
    scoring="neg_mean_absolute_error", n_iter=20,
random_state=0, n_jobs=2
)
cv_results = pd.DataFrame(search_cv.cv_results_)
```

    - ◊ learning rate needs to be at least > 0.1
    - ◊ best ranked models need more trees, or leaves, for a smaller learning rate
    - ◊ However, firm conclusions are difficult
      - † Since the best value of a hyperparameter so heavily depends on the others
- Note: the best model fit with training data and scored with the test data will perform better
  - ¬ than the CV results, which uses a subset of the training set to train
  - ¬ because it uses a larger set for training

§ 6.4.2 Exercise M6.04

- ensemble_ex_04.ipynb
- Don't forget about KFold:

```
from sklearn.model_selection import KFold
Kfold = KFold(n_splits=5, shuffle=True, random_state=0)
```

- Check the best resulting parameters and the number of trees from a GridSearch estimator

```
for each in cv_results['estimator']:
    print(each.best_params_)
    print(each.best_estimator_.n_iter_)
```

- I'm not sure what this means:

Inspect the results of the inner CV for each estimator of the outer CV. Aggregate the mean test score for each parameter combination and make a box plot of these scores.

— MODULE 7: EVALUATING MODEL PERFORMANCE

§ Contents

- When to use a specific cross-valdation or metric
  - ¬ eg, nested cv, optimization, regression, classification

§ 7.1.1 Comparing model performance with a simple baseline

- cross_validation_baseline.ipynb
- ShuffleSplit
  - ¬ with 20% of data held for validation
- Baseline
  - ¬ eg, Dummy Regressor
    - ◊ always predicts the mean of the target column
      `strategy="mean"`
    - ◊ does not reference the data
- Run both the DecisionTreeRegressor and the DummyRegressor through cross-validate
  - ¬ Store 'test_score' key in a pandas series
- Plot these values
  - ¬ eg: decision tree is lower than dummy, but still considerable error
- Analysis
  - ¬ we can show that a model accounting for features is more accurate
  - ¬ vs. just picking the average every time
    - ◊ median is also an option, which performs better for a set with extreme outliers

§ 7.1.2 Exercise M7.01

- cross_validation_ex_02.ipynb
- Good separation between regression and dummy scores shows that regression is effective
- Since classes are imbalanced, predicting most frequent will show better than realistic results
- Stratified
  - ¬ randomly generates predictions based on the training set's distribution
- Uniform
  - ¬ generates uniformly random predictions
  - ¬ for binary classification, CV will be 50% on average
- `permutation_test_score`
  - ¬ instead of using a dummy classifier
  - ¬ compares CV of a model against the CV of the same model
    - ◊ but trained on randomly permuted class labels
  - ¬ many ML estimators end up approximately behaving like most frequent Dummy Classifier
    - ◊ ie, always predicting the majority class
    - ◊ hence why this dummy is then substituted
  - ¬ **chance level**
    - ◊ for imbalanced classification problems
    - ◊ `most_frequent` is sometimes called this, even though no "chance" is really at play
  - ¬ actually using `permutation_test_score` is computationally intensive

§ 7.2.1 Stratification

- cross_validation_stratification.ipynb
- Other options besides KFold and ShuffleSplit
- Create a small random test set and run KFold as the splitter
  - ¬ Then use LogisticRegresion

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

model = make_pipeline(StandardScaler(), LogisticRegression())

import numpy as np
from sklearn.model_selection import KFold

data_random = np.random.randn(9, 1)
cv = KFold(n_splits=3)
for train_index, test_index in cv.split(data_random):
    print("TRAIN:", train_index, "TEST:", test_index)

from sklearn.model_selection import cross_validate

cv = KFold(n_splits=3)
results = cross_validate(model, data, target, cv=cv)
test_score = results["test_score"]
print(f"The average accuracy is "
      f"{test_score.mean():.3f} +/- {test_score.std():.3f}")
```

◊ Train and test sets look like the following:

```
TRAIN: [3 4 5 6 7 8] TEST: [0 1 2]
TRAIN: [0 1 2 6 7 8] TEST: [3 4 5]
TRAIN: [0 1 2 3 4 5] TEST: [6 7 8]
```

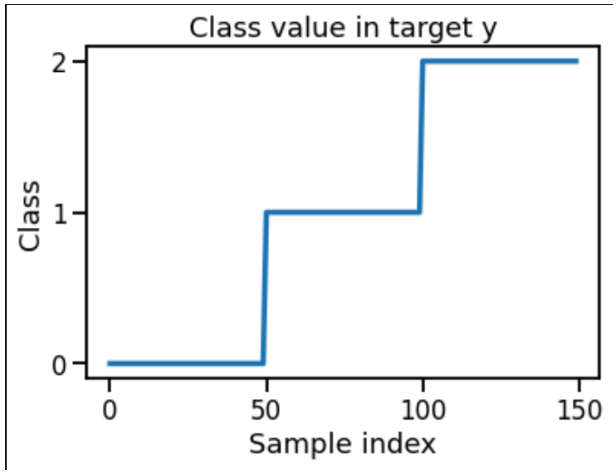
◊ And average accuracy ends up at 0

◊ Why?

† The target vector is ordered

» Our target column happens to have exactly three values

† and KFold does not shuffle, it splits in order



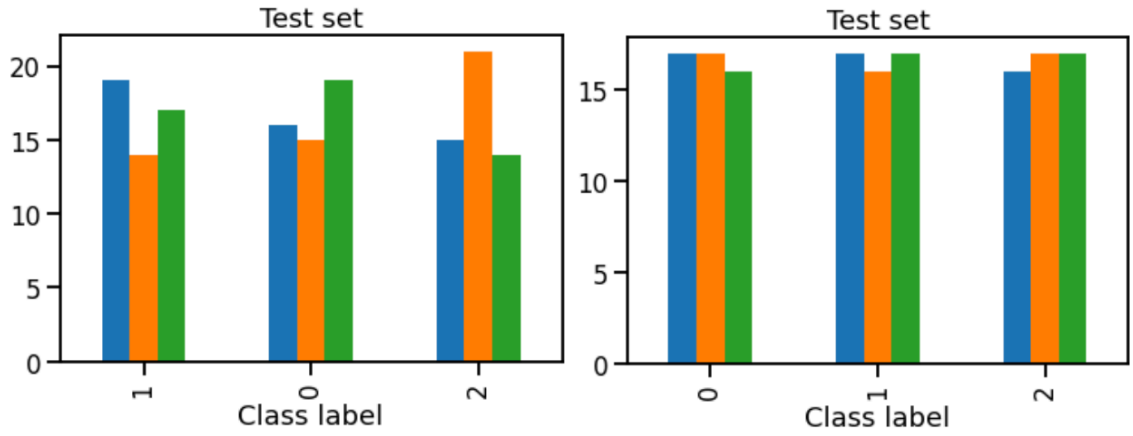
- Add `shuffle` in as a kwarg for KFold and it all works much better

```
cv = KFold(n_splits=3, shuffle=True, random_state=0)
```

    - ◊ eg: Accuracy is 95%
- Even better, use a `Stratified` strategy for CV

```
from sklearn.model_selection import StratifiedKFold
cv = StratifiedKFold(n_splits=3)
```

    - ◊ Rather than truly random shuffling
    - ◊ eg: 96% accuracy
    - ◊ **stratify**
        - † for unequal numbers of each class
        - † to preserve the original class frequencies
        - † applied to the data split, according to class
        - † ensures that each split gets an equal proportion of each class in the target column
    - ¬ KFold vs StratifiedKFold:



§ 7.2.2 Sample grouping

- cross_validation_grouping.ipynb
- Examining sample groups some more
    - ¬ First, run with plain KFold
    - ¬ Then run with shuffled KFold
        - ◊ this one has less variance (test scores are less spread out)
- But there are two folds in the no-shufflle scores that are conspicuously lower
- Looking at the data, there is a distinct pattern in the target column
    - ¬ looking at the documentation, it explains 13 individuals were responsible for writing test data
    - ¬ there are 14 very similar patterns of number sequences in the test data
- Use those apparent patterns to chop up the data into chunks
    - ¬ and label each for a different writer
- Supply this information to `GroupKFold` through the cross-validation function

```
from sklearn.model_selection import GroupKFold
cv = GroupKFold()
test_score = cross_val_score(model, data, target, groups=groups,
                             cv=cv, n_jobs=2)
```

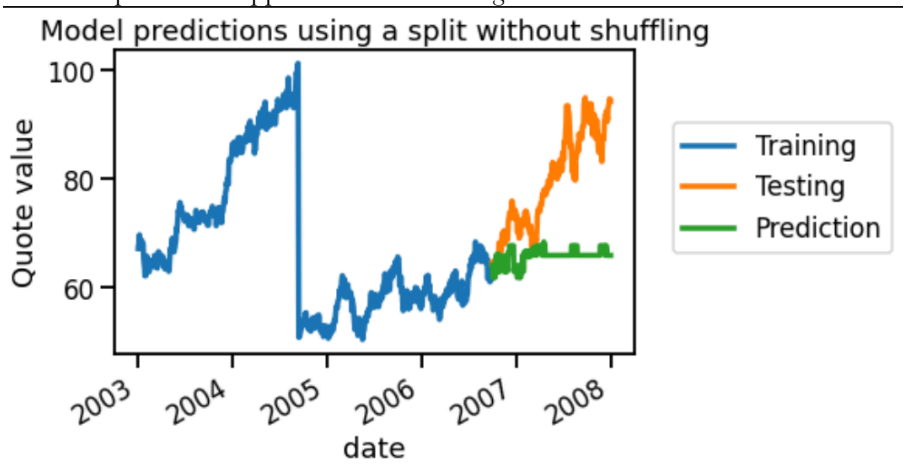
- However, the score ends up being lower than unshuffled KFold
    - ¬ 0.931 +/- 0.026
    - ¬ 0.920 +/- 0.021
        - ◊ But there was bias built into the test data that allowed "cheating"
        - ◊ standard deviation is lower, thoughs

§ 7.2.2 Non i.i.d. data

- cross_validation_time.ipynb
- **Independent and Identically Distributed Random Variables**
    - ¬ i.i.d. data
- Time series
    - ¬ samples depend on the past information
    - ¬ non-i.i.d.
- Predictive model could not work on random data
    - ¬ can it work on stock price data
    - ¬ try to predict Chevron, using data from Exxon, ConocoPhillips and Valero
- Decision Tree Regressor was used
    - ¬ expected to overfit
    - ¬ not expected to generalize then
- But regressor actually works very well for some reason
    - ¬ we try with CV
    - ¬ and also with R2, on a single test/train split, with `shuffle=True`

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import r2_score

data, target = quotes.drop(columns=["Chevron"]),
quotes["Chevron"]

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

regressor = DecisionTreeRegressor()
regressor.fit(data_train, target_train)
target_predicted = regressor.predict(data_test)

test_score = r2_score(target_test, target_predicted)
```

- Our testing set closely mirrors the training set
    - ¬ so the model really just memorized the training set
- Check if this is happening by not shuffling the data when doing the split
    - ¬ first 75% training, last 25% is testing
    - ¬ previously, not shuffling the data made it perform marginally worse
        - ◊ Now, not shuffling causes the score to fail miserably
        - ◊ the data to be predicted happens after the training



- Other solutions for trying to predict:

¬ Group samples into time blocks, eg by quarter

```
from sklearn.model_selection import LeaveOneGroupOut

groups = quotes.index.to_period("Q")
cv = LeaveOneGroupOut()
test_score = cross_val_score(regressor, data, target,
                             cv=cv, groups=groups, n_jobs=2)
```

◊ but this still fails

¬ For forecasting, we shouldn't use training data ulterior to the testing data

```
from sklearn.model_selection import TimeSeriesSplit

cv = TimeSeriesSplit(n_splits=groups.nunique())
test_score = cross_val_score(regressor, data, target,
                             cv=cv, groups=groups, n_jobs=2)
```

◊ but this also fails

§ 7.3.1 Nested cross-validation

- cross_validation_nested.ipynb
- Using a single cross-validation for both
  - ¬ hyperparameter tuning and
  - ¬ estimating generalization performance
- Problematic, because the evaluation will underestimate overfitting from the tuning process
- Hyperparameter tuning is a form of ML itself
  - ¬ requires nested CV
- GridSearchCV
  - ¬ has `.best_params_` attribute
  - ¬ and `.best_score_` attribute
    - ◊ however, we used knowledge from the test sets to select the hyperparameters
- Nested CV
  - ¬ Inner CV will only get the training set of the outer CV
  - ¬ keeps final testing scores independent
- While the nested score might be close to the GridSearch score, it can be trusted
  - ¬ by doing multiple runs of both nested and bare GridSearch
  - ¬ we see overall, the basic GridSearch is optimistic compared to the nested CV

§ 7.4.1 Classification

- metrics_classification.ipynb
- **Objective function**
  - ¬ ML operates by optimizing these functions
  - ¬ the objective function is typically decoupled from the evaluation metric
    - ◊ the objective function serves as a proxy for the evaluation metric
- Classification metrics
  - ¬ where target vector is categorical, rather than continuous
- Bar graph the number of samples for each of the classes

```
import matplotlib.pyplot as plt

target.value_counts().plot.barh()
plt.xlabel("Number of samples")
_ = plt.title("Number of samples per classes present\n in the
target")
```

    - ¬ Eg: shows we have two, imbalanced classes
- Eg: only fits on a single split, rather than a proper CV, to focus on the metrics presentation

```
data_train, data_test, target_train, target_test =
train_test_split(
    data, target, shuffle=True, random_state=0, test_size=0.5)

classifier = LogisticRegression()
classifier.fit(data_train, target_train)
```

¶ Classifier predictions

• Eg: then do a single prediction as well as predictions for the whole test set

```
new_donor = pd.DataFrame(
    {
        "Recency": [6],
        "Frequency": [2],
        "Monetary": [1000],
        "Time": [20],
    }
)
classifier.predict(new_donor)
target_predicted = classifier.predict(data_test)
target_predicted[:5]
```

¶ Accuracy as a baseline

• Compare predictions with the ground-truth

¬ **ground-truth**

◊ the actual target vector from the test data

◊ `target_test`

```
target_test == target_predicted
```

• Calculate the accuracy

```
import numpy as np
np.mean(target_test == target_predicted)
```

OR:

```
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(target_test, target_predicted)
```

OR:

```
classifier.score(data_test, target_test)
```

¶ Confusion matrix and derived metrics

• Finer granularity of the error in the example

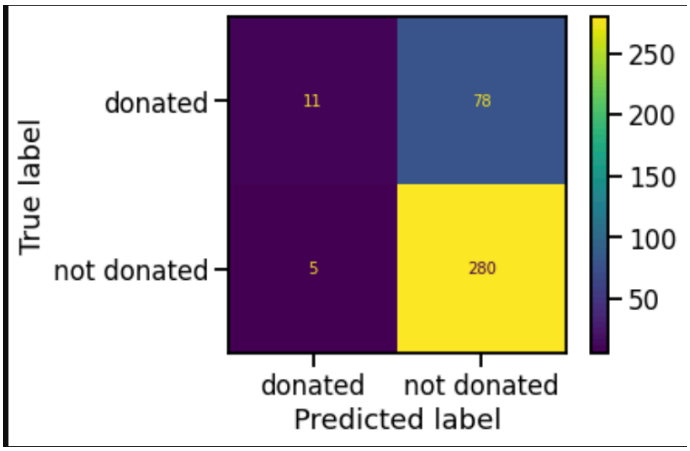
¬ either we predicted they would give blood when they didn't

¬ or predicted they would not when they did

```
from sklearn.metrics import ConfusionMatrixDisplay
_ = ConfusionMatrixDisplay.from_estimator(
                                classifier, data_test, target_test)
```

◊ Confusion Matrix

TP	True positives
TN	True negatives
FN	False negatives
FP	False positives



¬ With these splits, we can show generalization performance for certain settings
◊ eg, examine the set of people who gave when we predicted they would
◊ eg, examine everyone who gave, regardless of prediction

• **Precision**
TP / (TP + FP)
¬ the likelihood someone actually gave blood when they were predicted to do so

• **Recall**
TP / (TP + FN)
¬ how well the classifier was able to correctly identify those who gave blood

```
from sklearn.metrics import precision_score, recall_score

precision = precision_score(target_test,
            target_predicted, pos_label="donated")
recall = recall_score(target_test, target_predicted,
         pos_label="donated")
```

• Both Precision and Recall only focus on samples predicted to be positive
¬ Accuracy takes both predictions into account
• Compare with the confusion matrix
¬ Left column
◊ more than half of donated predictions were correct
◊ Precision = .688
¬ Right column
◊ a minority, but still a large number of donors mislabeled as not donated
◊ Recall = .124

¶ The issue of class imbalance

• When classes are far out of balance, the accuracy score should not be used
¬ a most frequent dummy classifier can actually beat the accuracy of our model

```
from sklearn.dummy import DummyClassifier
dummy_classifier = DummyClassifier(strategy="most_frequent")
dummy_classifier.fit(data_train, target_train)
```

• Either use Precision and Recall, or use the balanced accuracy score instead

```
from sklearn.metrics import balanced_accuracy_score
balanced_accuracy = balanced_accuracy_score(target_test,
target_predicted)
```

◊ defined as the average recall obtained on each class

¶ Evaluation and different probability thresholds

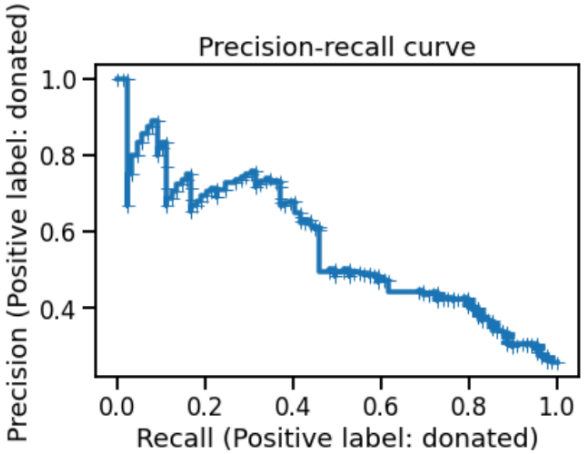
• Binary predictions of yes or no can be supplemented by probability ratios instead

```
pd.DataFrame(classifier.predict_proba(data_test),
                                columns=classifier.classes_)
```

• Get the labels out of a trained classifier for a pandas DataFrame

```
columns=classifier.classes_
```

• Get the max value from each row of a DataFrame and its column name in a Series
```
target_proba_predicted.idxmax(axis=1)
```
• Turn a Series into a numpy array
```
.to_numpy()
```
• Check numpy array if all values are True
```
np.all(ndarray_name)
```
• Check numpy array if any values are True
```
np.any(ndarray_name)
```
• Binary predictions are based upon the default decision threshold of 0.5
    ¬ but this might not lead to the optimal generalization performance of our classifier
• Precision-Recall Curve
```
from sklearn.metrics import PrecisionRecallDisplay

disp = PrecisionRecallDisplay.from_estimator(
    classifier, data_test, target_test, pos_label='donated',
    marker="+"
)
plt.legend(bbox_to_anchor=(1.05, 0.8), loc="upper left")
_ = disp.ax_.set_title("Precision-recall curve")
```

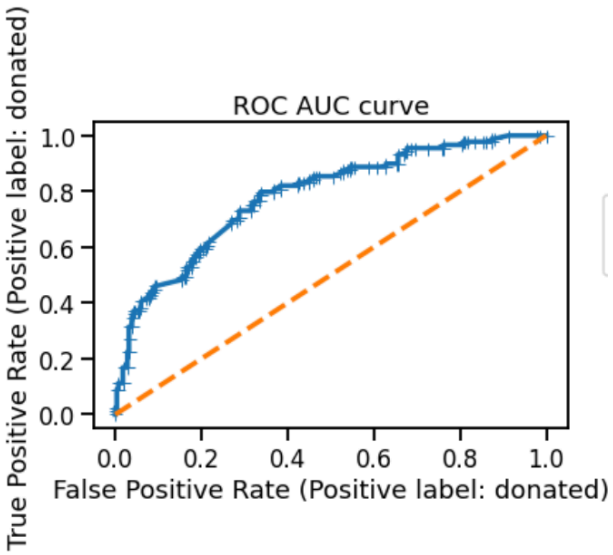


    † Scikit-learn returns a display with all plotting elements
    † displays expose a matplotlib axis to add new elements
        `ax_`
• On this curve, each blue cross corresponds to a decision threshold probability
    ¬ If we wanted to drive Recall to 1 (perfect) or the same for Precision
        ◊ then the other would go to 0
    ¬ Looks like the goal would be to get ourselves as far away from the origin as possible
        ◊ ie, minimizing False Negatives (maximize Recall)
        ◊ and minimizing False Positives (maximize Precision)
    ¬ But also depends of if FP or FN are more the more adverse outcome
• Average Precision
    ¬ AP
    ¬ Area under the curve (AUC)
    ¬ for ideal classifier,
• Alternate metric for determining gen. performance by varying probability threshold
    ¬ Focus on compromise of positive and negative class determination
    ¬ **Sensitivity**
        TP / (TP + FN)
        ◊ same as Recall
    ¬ **Specificity**
        TN / (TN + FP)

◊ changed from Precision by replacing with TN on top and bottom

- Receiver Operating Characteristic Curve

```
from sklearn.metrics import RocCurveDisplay

disp = RocCurveDisplay.from_estimator(
    classifier, data_test, target_test, pos_label='donated',
    marker="+")
disp = RocCurveDisplay.from_estimator(
    dummy_classifier, data_test, target_test,
pos_label='donated',
    color="tab:orange", linestyle="--", ax=disp.ax_)
plt.legend(bbox_to_anchor=(1.05, 0.8), loc="upper left")
_ = disp.ax_.set_title("ROC AUC curve")
```



- Overall generalization performance of the classifier
  - ¬ Area under the curve
    - ◊ Lower bound is the most-frequent dummy classifier line
    - ◊ Lower bound is 0.5

§ 7.4.2 Exercise M7.02

- metrics_ex_01.ipynb
- Running a cross-validation with
  - ¬ a custom scorer, to evaluate Precision, using 'donated' as the positive label
  - ¬ using a list of scoring methods, then plotting the results
- Another way to create a DataFrame

```
metrics = pd.DataFrame(
[scores["test_accuracy"], scores["test_balanced_accuracy"]],
index=["Accuracy", "Balanced accuracy"]).T
```

§ 7.5.1 Regression

- metrics_regression.ipynb
- Optimization problem
  - ¬ some ML models are designed to minimize an error using the training set
    - ◊ aka, **loss function**
- **Mean squared error**

```
mean_squared_error(target_test, target_predicted)
```

  - ◊ one example of a loss function

- **R^2**

```
regressor.score(data_test, target_test)
```

    - ◊ coefficient of determination
    - ◊ rescaled MSE
        - † proportion of variance of the target
            - † that is explained by the independent variables in the model
    - ◊ the default score in scikit-learn

    - ◊ Best possible score is 1, no lower bound
        - † However, model that predicts the expected value scores 0
    - ◊ Gives insight into quality of the model's fit
        - † But cannot be compared between datasets
        - † No meaning relative to the original units of the target

- **Mean absolute error**

```
mean_absolute_error(target_test, target_predicted)
```

    - ◊ eg: our model is predicting, on average, 22.6 k$ away from the true price
    - ◊ means can be overly impacted by large error

- **Median absolute error**

```
median_absolute_error(target_test, target_predicted)
```

    - ◊ Both Mean and Median have major limitation
        - † incurring an error of 50 k$ on a 50 k$ house
        - † has the same error impact as on a 500 k$ house

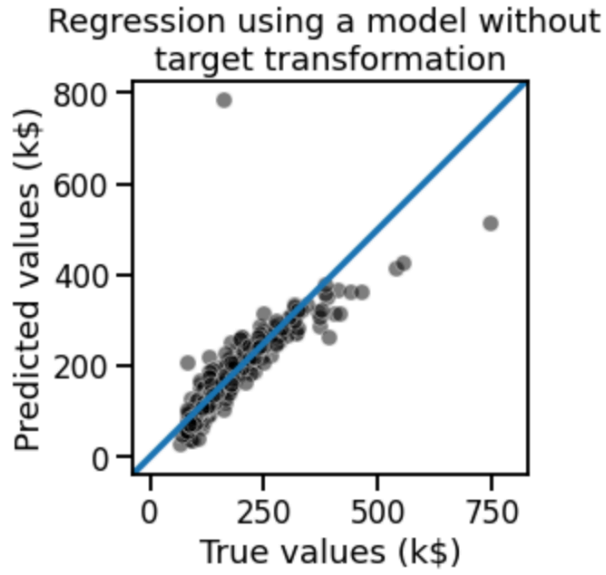
- **Mean absolute percentage error**

```
mean_absolute_percentage_error(target_test, target_predicted)
```

    - ◊ introduces relative scaling to address, eg, large error on a smaller house price

- Plotting a graph

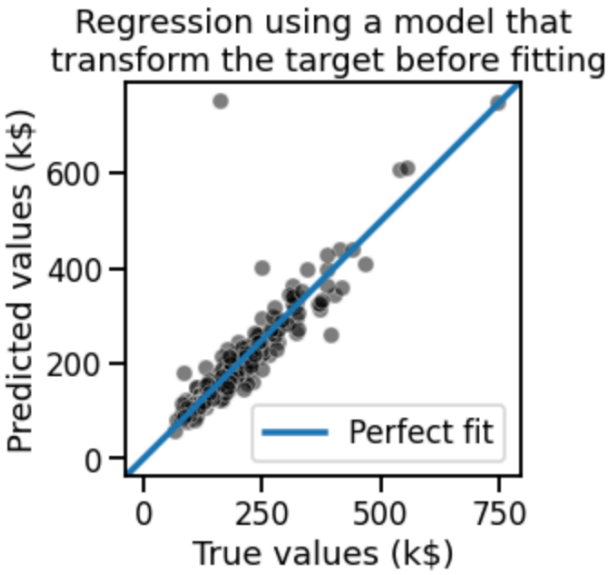
```
import matplotlib.pyplot as plt
import seaborn as sns

predicted_actual = {
    "True values (k$)": target_test, "Predicted values (k$)":
target_predicted}
predicted_actual = pd.DataFrame(predicted_actual)

sns.scatterplot(data=predicted_actual,
                x="True values (k$)", y="Predicted values (k$)",
                color="black", alpha=0.5)
plt.axline((0, 0), slope=1, label="Perfect fit")
plt.axis('square')
_ = plt.title("Regression using a model without \ntarget
transformation")
```



† The plot give a visual way to check if the model makes consistent errors
» eg, the model underestimates on large True Value prices
» typically happens when target does not follow a normal distribution
» try target transformation in these cases

- **Target transformation**

```
from sklearn.preprocessing import QuantileTransformer
from sklearn.compose import TransformedTargetRegressor

transformer = QuantileTransformer(
    n_quantiles=900, output_distribution="normal")
model_transformed_target = TransformedTargetRegressor(
    regressor=regressor, transformer=transformer)
model_transformed_target.fit(data_train, target_train)
target_predicted = model_transformed_target.predict(data_test)

predicted_actual = {
    "True values (k$)": target_test, "Predicted values (k$)":
target_predicted}
predicted_actual = pd.DataFrame(predicted_actual)
```



† transformed target model performs much better for high values

§ 7.5.2 Exercise M7.03

- metrics_ex_02.ipynb
- Regression metrics within a cross-validation framework

§ Module 7 Quiz

- Negative values in a series changed to 0s

```
accel = data['acceleration'].copy()
accel[accel < 0] = 0
```

Or:

```
accel.clip(lower=0)
```

- Pandas index operations
  - ¬ Return a dataframe index

    ```
    data.index
    ```

    ◊ object type: `pandas.core.indexes.datetimes.DatetimeIndex`
  - ¬ Return a numpy array of dates

    ```
    data.index.date
    ```
  - ¬ Return a numpy array of times

    ```
    data.index.time
    ```
  - ¬ Change an index object into a numpy array

    ```
    data.index.values
    ```
  - ¬ Return a numpy array of unique index values

    ```
    data.index.unique().values
    ```
- LeaveOneGroupOut generator

```
import numpy as np
from sklearn.model_selection import LeaveOneGroupOut

X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
y = np.array([1, 2, 1, 2])
groups = np.array([1, 1, 2, 2])
logo = LeaveOneGroupOut()

# Returns all possible permutations of training and testing
splits.
for train_index, test_index in logo.split(X, y, groups):
    print(train_index, test_index)
```

¬ For loop returns each possible permutations of testing and training sets

— **CONCLUSION**

§ Concluding remarks

- https://www.youtube.com/watch?v=dqnPOlPYA4s
- The big messages of the Mooc
  - ¬ The machine learning pipeline
    - ◊ the learners, the predictive models are trained on the train set
      - † different from the test set
    - ◊ built from a data matrix
      - † a give number of features for each observation
    - ◊ transformations of the data
      - † encoding categorical variables
      - † using only information available at train time
      - † scikit-learn pipeline object to facilitate this
  - ¬ Adapting model complexity to the data
    - ◊ minimize the error on the test set
      - † but train error can detect underfit
        - » ie, models too simple for the data
    - ◊ multiple hyper-parameters
      - † they control model complexity
      - † selecting them is important
      - † scikit-learn: GridSearchCV, RandomSearchCV
  - ¬ Specific models
    - ◊ understanding the models
      - † know when they are suited to the data
      - † debugging intuitions
    - ◊ linear models
      - † combining the values of features
      - † most useful for many features, or few observations
    - ◊ tree-based models
      - † series of binary choices (thresholds)
      - † useful for tabular data, columns of different nature, eg, age, weight, sex
      - † scikit-learn: HistGradientBoostingRegressor and Classifier
  - ¬ Learning more about scikit-learn
    - ◊ documentation
      - † rich, didactic, continuously improving
      - † user guide
    - ◊ stack overflow for questions

- ◊ we are an open-source community
  - † free, open, driven by community trying to be inclusive
  - † help by training others, communication, advocacy

• Impact on society

¬ Validation and evaluation matter

- ◊ a measure of prediction accuracy is always an imperfect estimate of how well it generalizes
  - † as you narrow down on a solution, spend increasingly more effort on validating it
  - † do many splits in your cross-validation
    - » despite cost in computation power

¬ Machine learning is a small part of the problem most of the time

- ◊ how to approach the whole problem
  - † what is the full value chain
- ◊ acquiring more/better data is often more important than using fancy models
- ◊ how are they being put into production, used routinely
  - † technical debt
    - » simpler models are easier to maintain, require less compute power
  - † drifts in the data distribution as more is accumulated over time
    - » continually checking model validation

¬ Technical craft is not all

- ◊ methodological elements are not enough to always have a solid conclusion, statistically
- ◊ after running software, biggest challenges:
  - † understanding the data
  - † understanding data's shortcomings
  - † what can and cannot be concluded
- ◊ automating machine learning does not solve data science
- ◊ domain knowledge and critical thinking about the data are crucial

¬ How the predicitons are used

- ◊ errors mean different things
  - † operational risk
    - » advertisement placement: errors are harmless
    - » medicine: errors can kill
  - † operational logic
    - » better a false detection or a miss?
    - » eg, detecting brain tumors
      - ‡ if sent to surgery: FP can be dangerous
      - ‡ however, sent to an MR scan: FN is the bigger threat, as scan is harmless
- ◊ predictions may modify how the system functions
  - † eg, predicting who will benefit from a hospital stay may overcrowd the hospital

¬ Choice of the output and the labeled dataset

- ◊ what we choose to predict is a very loaded choice
- ◊ interesting labels are often hard to get
  - † we then emphasize easy ways of accumulating labels
    - » but these come with biases
- ◊ our target value may just be a proxy of the real quantity of interest

¬ Biases in the data

- ◊ the data may not reflect the ground truth
  - † disease monitoring is a function of the testing policy
    - » which may change over time, or uneven across the population
    - » eg, wealthy people tend to have higher quality data
- ◊ the measured state of affaires may not be the desired one
  - † eg, women are payed less than men
  - † a learner will pick up and amplify inequalities

¬ Prediction models vs Causal models

- ◊ eg, people that go to the hospital die more than people who don't
  - † fallacy: comparing different populations
- ◊ eg, having blood pressure greater than a threshold triggers care which can be beneficial

† fallacy: having above-threshold blood pressure is beneficial
◊ Pure predictive settings, information is beneficial for their predictions
† but should not be trusted for designing interventions
† interpretation is subject in caution
¬ Societal impacts
◊ AI systems used for loans, jobs, medial treatment and law enforcement
† shortcomings in the data and models will harm people
`http://fairlearn.org`
» uses the same framework as scikit-learn
◊ ML can change:
† decision logic
† power structure
† operational costs
◊ No solution will be purely technical

§ <u>Topics we have not covered</u>

Topics we have not covered

- Unsupervised learning
  - ¬ Finding order and structure in the data, for instance to group samples, or to transform features
  - ¬ Particularly useful because it does not need labels
  - ¬ But given labels, supervised learning not unsupervised learning, is more likely to recover the link between data and labels
- Model inspection
  - ¬ Understanding what drives a prediction
  - ¬ Useful for debugging, for reasoning on the system at hand
  - ¬ Requires a lot of nuance
- Deep learning
  - ¬ Often not better than gradient boosting trees for classification or regression on tabular data
  - ¬ But more flexible: can work natively with tasks that involve variable length structures in the input and output of the model (e.g. speech to text)
  - ¬ For images, text, voice: use pretrained models
  - ¬ Comes with great computational and human costs, as well as large maintenance costs
  - ¬ Not in scikit-learn: have a look at resources on pytorch and tensorflow to get started!

Compare this to the quiz "plateau"

cross_validation_.ipynb

The average accuracy is 0.953 +/- 0.009
The average accuracy is 0.960 +/- 0.016

Don't understand how lower bound of ROC-AUC is .5, when area under dummy is .5?