Review_Computational_Foundation

COSC 3337 :Data Science I
Instructor: Nouhad Rizk

1 Data Preprocessing and Machine Learning with Scikit-Learn

[ ]:
```
#%load_ext watermark
#%watermark -v -a 'Nouhad Rizk' -p numpy,scipy,matplotlib,sklearn,mlxtend
```

1.1 Overview

In this lecture, we are closing the "Computational Foundation" section by introducing yet another Python library, pandas, which is extremely handy for data (pre)processing. The second focus of this lecture is on the Scikit-learn machine learning library, which is widely considered as the most mature and most well-designed general machine learning library.

1.2 Pandas – A Python Library for Working with Data Frames

- Pandas is probably the most popular and convenient data wrangling library for Python (official website: https://pandas.pydata.org)
- Pandas stands for PANel-DAta-S.
- Relativ similar to data frames in R.
- How is it different from NumPy arrays?
  - Allows for heterogenous data (columns can have different data types)
  - Adds some more convenient functions on top that are handy for data processing

1.2.1 Loading Tabular Datasets from Text Files

- Here, we are working with structured data, data which is organized similar to a "design matrix" (see lecture 1) – that is, examples as rows and features as columns (in contrast: unstructured data such as text or images, etc.).
- CSV stands for "comma separated values" (also common: TSV, tab seperated values).
- The `head` command is a Linux/Unix command that shows the first 10 rows by default; the `!` denotes that Jupyter/the IPython kernel should execute it as a shell command (`!`-commands may not work if you are on Windows, but it is not really important).

[ ]:
```
!head iris.csv
```

- We use the `read_csv` command to load the CSV file into a pandas data frame object f of the class `DataFrame`.
- Data frames also have a `head` command; here it shows the first 5 rows.

[ ]:
```
import pandas as pd


df = pd.read_csv('iris.csv')
df.head()
```

[ ]:
```
type(df)
```

- It is always good to double check the dimensions and see if they are what we expect.
- The `DataFrame` `shape` attribute works the same way as the NumPy array `shape` attribute (Lecture 04).

[ ]:
```
df.shape
```

1.2.2 Basic Data Handling

- The `apply` method offers a convenient way to manipulate pandas `DataFrame` entries along the column axis.
- We can use a regular Python or lambda function as input to the apply method.
- In this context, assume that our goal is to transform class labels from a string representation (e.g., "Iris-Setosa") to an integer representation (e.g., 0), which is a historical convention and a recommendation for compatibility with various machine learning tools.

[ ]:
```
df['Species'] = df['Species'].apply(lambda x: 0 if x=='Iris-setosa' else x)
df.head()
```

Digression: Lambda Functions

- If you are not familiar with "lambda functions," they are basically the same as "regular function but can be written more compactly as a one-liner.

[ ]:
```
def some_func(x):
    return 'Hello World ' + str(x)

some_func(123)
```

[ ]:
```
f = lambda x: 'Hello World ' + str(x)
f(123)
```

.map vs. .apply

- If we want to map column values from one value to another, it is often more convenient to use the `map` method instead of apply.
- The achieve the following with the `apply` method, we would have to call `apply` three times.

[ ]:
```
d = {'Iris-setosa': 0,
     'Iris-versicolor': 1,
     'Iris-virginica': 2}

df = pd.read_csv('iris.csv')
df['Species'] = df['Species'].map(d)
df.head()
```

- The `tail` method is similar to `head` but shows the last five rows by default; we use it to double check that the last class label (Iris-Virginica) was also successfully transformed

[ ]:
```
df.tail()
```

- It's actually not a bad idea to check if all row entries of the `Species` column got transformed correctly.

[ ]:
```
import numpy as np


np.unique(df['Species'])
```

NumPy Arrays

- Pandas' data frames are built on top of NumPy arrays.
- While many machine learning-related tools also support pandas `DataFrame` objects as inputs now, by convention, we usually use NumPy arrays most tasks.
- We can access the NumPy array that is underlying a `DataFrame` via the `values` attribute.

[ ]:
```
y = df['Species'].values
y
```

- There are many different ways to access columns and rows in a pandas `DataFrame`, which we won't discuss here; a good reference documentation can be found at https://pandas.pydata.org/pandas-docs/stable/indexing.html
- The `iloc` attribute allows for integer-based indexing and slicing, which is similar to how we use indexing on NumPy arrays (Lecture 04). The following expression will select column 1, 2, 3, and 4 (sepal length, sepal width, petal length, petal width) from the `DataFrame` and then assign the underlying NumPy array to `X`.

[ ]:
```
X = df.iloc[:, 1:5].values
```

- Just as a quick check, we show the first 5 rows in the NumPy array:

[ ]:
```
X[:5]
```

1.2.3 Exploratory Data Analysis

- Occasionally, we will use the MLxtend library (http://rasbt.github.io/mlxtend/) – MLxtend stands for "machine learning extensions" and contains some convenience functions for machine

learning and data science tasks.

- In particular, we will use the `scatterplotmatrix` function to display a scatter plot matrix of the dataset, which is useful to get a quick overview of the dataset (to inspect the relationship between features, look for outliers, etc.).
- I just added the `scatterplotmatrix` functions a few days ago, hence it is not in mlxtend's latest release version; however, you can install the development version (currently 0.14dev) directly from GitHub by un-commenting and executing the next cell.

[ ]:
```
#!pip install mlxtend
```

[ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from mlxtend.data import iris_data
from mlxtend.plotting import scatterplotmatrix


names = df.columns[1:5]

fig, axes = scatterplotmatrix(X[y==0], figsize=(10, 8), alpha=0.5)
fig, axes = scatterplotmatrix(X[y==1], fig_axes=(fig, axes), alpha=0.5)
fig, axes = scatterplotmatrix(X[y==2], fig_axes=(fig, axes), alpha=0.5,␣
 ↪names=names)

plt.tight_layout()
plt.legend(labels=['Setosa', 'Versicolor', 'Virginica'])
plt.savefig('images/eda.pdf')
plt.show()
```

1.3 Splitting a Dataset into Train, Validation, and Test Subsets

- The following code cells in this section illustrate the process of splitting a dataset into several subsets.
- One important step, prior to splitting a dataset, is shuffling it, otherwise, we may end up with unrepresentative class distributions if the dataset was sorted prior to splitting.

[ ]:
```
import numpy as np


indices = np.arange(X.shape[0])
rng = np.random.RandomState(123)
permuted_indices = rng.permutation(indices)
permuted_indices
```

[ ]:
```
train_size, valid_size = int(0.65*X.shape[0]), int(0.15*X.shape[0])
test_size = X.shape[0] - (train_size + valid_size)
print(train_size, valid_size, test_size)
```

[ ]:
```
train_ind = permuted_indices[:train_size]
valid_ind = permuted_indices[train_size:(train_size + valid_size)]
test_ind = permuted_indices[(train_size + valid_size):]
```

[ ]:
```
X_train, y_train = X[train_ind], y[train_ind]
X_valid, y_valid = X[valid_ind], y[valid_ind]
X_test, y_test = X[test_ind], y[test_ind]

X_train.shape
```

1.4 Python Classes

- This section illustrates the concept of "classes" in Python, which is relevant for understanding how the scikit-learn API works on a fundamental level later in this lecture.
- Note that Python is an object oriented language, and everything in Python is an object.
- Classes are "templates" for creating objects (this is called "instantiating" objects).
- An object is a collection of special "functions" (a "function" of an object or class is called "method") and attributes.
- Note that the `self` attribute is a special keyword for referring to a class or an instantiated object of a class, "itself."

[ ]:
```
class VehicleClass():

    def __init__(self, horsepower):
        "This is the 'init' method"
        # this is a class attribute:
        self.horsepower = horsepower

    def horsepower_to_torque(self, rpm):
        "This is a regular method"
        numerator = self.horsepower * 33000
        denominator = 2* np.pi * 5000
        return numerator/denominator

    def tune_motor(self):
        self.horsepower *= 2

    def _private_method(self):
        print('this is private')

    def __very_private_method(self):
        print('this is very private')
```

[ ]:
```
# instantiate an object:
car1 = VehicleClass(horsepower=123)
print(car1.horsepower)
```

[ ]:
```
car1.horsepower_to_torque(rpm=5000)
```

[ ]:
```
car1.tune_motor()
car1.horsepower_to_torque(rpm=5000)
```

[ ]:
```
car1._private_method()
```

- Python has the motto "we are all adults here," which means that a user can do the same things as a developer (in contrast to other programming languages, e.g., Java).
- A preceding underscore is an indicator that a method is considered "private" – this means, this method is meant to be used internally but not by the user directly (also, it does not show up in the "help" documentation)
- a preceding double-underscore is a "stronger" indicator for methods that are supposed to be private, and while users can access these (adhering to the "we are all adults here" moto), we have to refer to "name mangling."

[ ]:
```
# Excecuting the following would raise an error:
# car1.__very_private_method()
```

[ ]:
```
# If we use "name mangling" we can access this private method:
car1._VehicleClass__very_private_method()
```

- Another useful aspect of using classes is the concept of "inheritance."
- Using inheritance, we can "inherit" methods and attributes from a parent class for re-use.
- For instance, consider the `VehicleClass` as a more general class than the `CarClass` – i.e., a car, truck, or motorbike are specific cases of a vehicle.
- Below is an example of a `CarClass` that inherits the methods from the `VehicleClass` and adds a specific `self.num_wheels=4` attribute – if we were to create a `BikeClass`, we could set this to `self.num_wheels=2`, for example.
- All-in-all, this is a very simple demonstration of class inheritance, however, it's a concept that is very useful for writing "clean code" and structuring projects – the scikit-learn machine learning library makes heavy use of this concept internally (we, as users, don't have to worry about it too much though, it is useful to know though in case you would like to modify or contribute to the library).

[ ]:
```
class CarClass(VehicleClass):

    def __init__(self, horsepower):
        super(CarClass, self).__init__(horsepower)
        self.num_wheels = 4

new_car = CarClass(horsepower=123)
print('Number of wheels:', new_car.num_wheels)
print('Horsepower:', new_car.horsepower)
new_car.tune_motor()
print('Horsepower:', new_car.horsepower)
```

1.5 K-Nearest Neighbors Implementation

- Below is a very simple implementation of a K-nearest Neighbor classifier.
- This is a very slow and inefficient implementation, and in real-world problems, it is always recommended to use established libraries (like scikit-learn) instead of implementing algorithms from scratch.
- The scikit-learn library, for example, implements kNN much more efficiently and robustly – using advanced data structures (KD-Tree and Ball-Tree, which we briefly discussed in Lecture 02).
- A scenario where it is useful to implement algorithms from scratch is for learning and teaching purposes, or if we want to try out new algorithms, hence, the implementation below, which gently introduces how things are implemented in scikit-learn.

[ ]:
```
class KNNClassifier(object):
    def __init__(self, k, dist_fn=None):
        self.k = k
        if dist_fn is None:
            self.dist_fn = self._euclidean_dist

    def _euclidean_dist(self, a, b):
        dist = 0.
        for ele_i, ele_j in zip(a, b):
            dist += ((ele_i - ele_j)**2)
        dist = dist**0.5
        return dist

    def _find_nearest(self, x):
        dist_idx_pairs = []
        for j in range(self.dataset_.shape[0]):
            d = self.dist_fn(x, self.dataset_[j])
            dist_idx_pairs.append((d, j))

        sorted_dist_idx_pairs = sorted(dist_idx_pairs)

        return sorted_dist_idx_pairs

    def fit(self, X, y):
        self.dataset_ = X.copy()
        self.labels_ = y.copy()
        self.possible_labels_ = np.unique(y)

    def predict(self, X):
        predictions = np.zeros(X.shape[0], dtype=int)
        for i in range(X.shape[0]):
            k_nearest = self._find_nearest(X[i])[:self.k]
            indices = [entry[1] for entry in k_nearest]
            k_labels = self.labels_[indices]
            counts = np.bincount(k_labels,
```

```
                                    minlength=self.possible_labels_.shape[0])
            pred_label = np.argmax(counts)
            predictions[i] = pred_label
        return predictions



knn_model = KNNClassifier(k=3)
knn_model.fit(X_train, y_train)
```

[ ]:
```
print(knn_model.predict(X_valid))
```

Note that there are class attributes with a `_` suffix in the implementation above – this is not a typo. - The trailing `_` (e.g., here: `self.dataset_`) is a scikit-learn convention and indicates that these are "fit" attributes – that is, attributes that are available only *after* calling the `fit` method.

1.6 The Scikit-Learn Estimator API

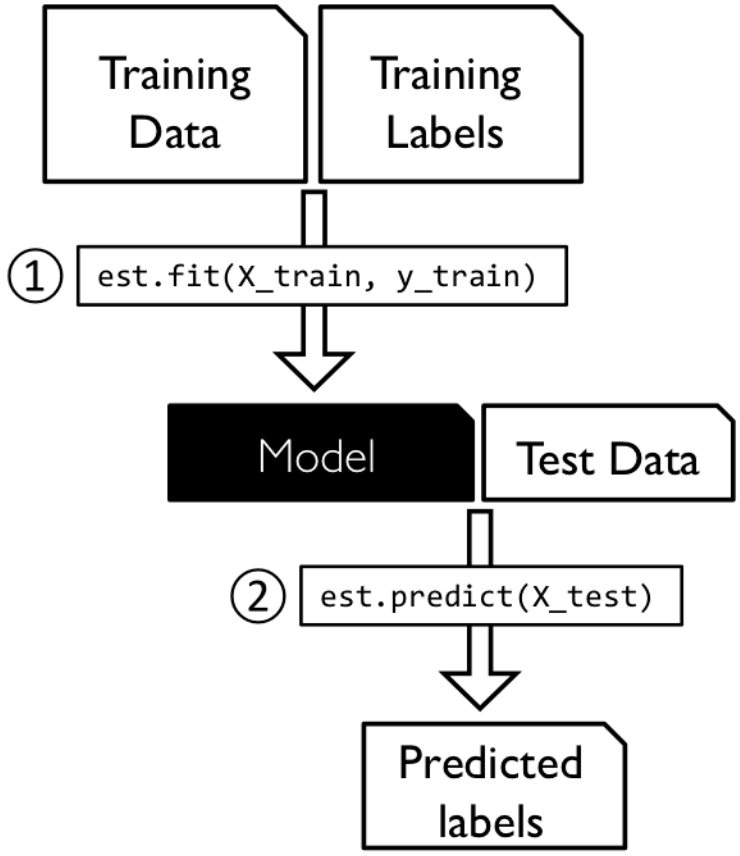
- Below is an overview of the scikit-learn estimator API, which is used for implementing classification and regression models/algorithms.
- We have seen the methods in the context of the kNN implementation earlier; however, one interesting, additional method we have not covered yet is `score`.
- The `score` method simply runs `predict` on the features (`X`) internally and then computes the performance by comparing the predicted targets to the true targets `y`.
- In the case of classification models, the `score` method computes the classification accuracy (in the range [0, 1]) – i.e., the proportion of correctly predicted labels. In the case of regression models, the `score` method computes the coefficient of determination (R^2).

```
class SupervisedEstimator(...):

    def __init__(self, hyperparam_1, ...):
        self.hyperparm_1
        ...

    def fit(self, X, y):
        ...
        self.fit_attribute_
        return self

    def predict(self, X):
        ...
        return y_pred

    def score(self, X, y):
        ...
        return score

    def _private_method(self):
```

...

...

- The graphic below summarizes the useage of the `SupervisedEstimator` API that scikit-learn uses for implementing classification and regression algorithms/models.

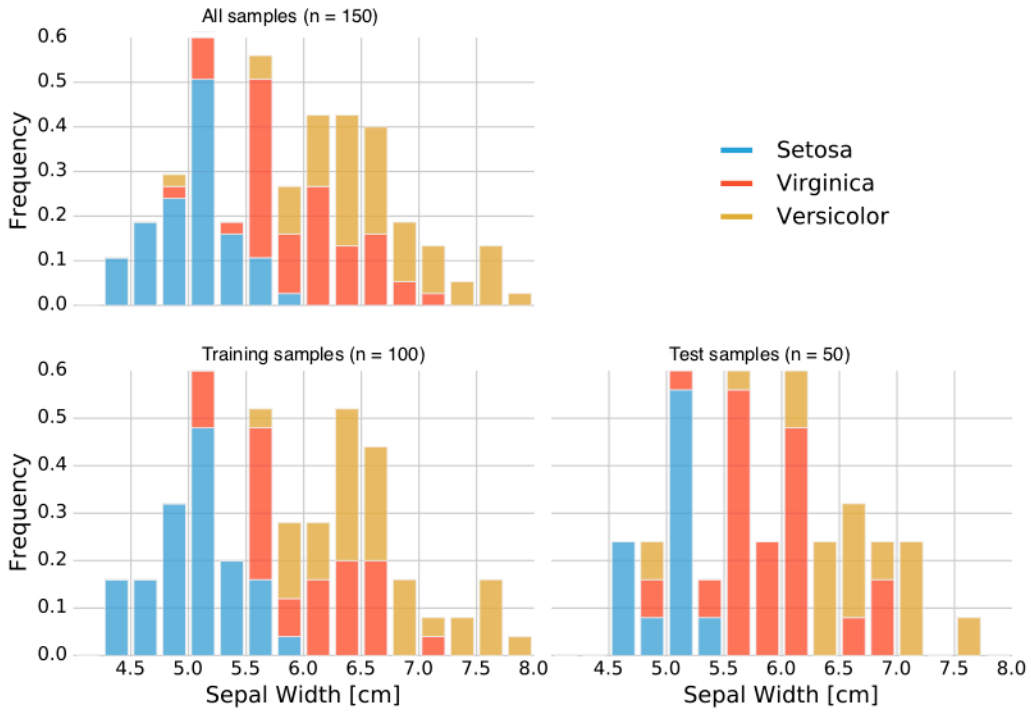


- For 2D datasets (which we usually only have in teaching/learning contexts), we can plot the decision regions using a convenient wrapper function in mlxtend as shown below.

[ ]:
```
from sklearn.neighbors import KNeighborsClassifier
from mlxtend.plotting import plot_decision_regions


knn_model = KNeighborsClassifier(n_neighbors=3)
knn_model.fit(X_train[:, 2:], y_train)
plot_decision_regions(X_train[:, 2:], y_train, knn_model)
plt.xlabel('petal length[cm]')
plt.ylabel('petal width[cm]')
plt.savefig('images/decisionreg.pdf')
plt.show()
```

1.7 Stratification

- Previously, we wrote our own code to shuffle and split a dataset into training, validation, and test subsets, which had one considerable downside.
- If we are working with small datasets and split it randomly into subsets, it will affect the class distribution in the samples – this is problematic since machine learning algorithms/models assume that training, validation, and test samples have been drawn from the same distributions to produce reliable models and estimates of the generalization performance.



- The method of ensuring that the class label proportions are the same in each subset after splitting, we use an approach that is usually referred to as "stratification."
- Stratification is supported in scikit-learn's `train_test_split` method if we pass the class label array to the `stratify` parameter as shown below.

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


X_temp, X_test, y_temp, y_test = \
        train_test_split(X, y, test_size=0.2,
                         shuffle=True, random_state=123, stratify=y)
np.bincount(y_temp)
```

[ ]:
```
X_train, X_valid, y_train, y_valid = \
        train_test_split(X_temp, y_temp, test_size=0.2,
                         shuffle=True, random_state=123, stratify=y_temp)

print('Train size', X_train.shape, 'class proportions', np.bincount(y_train))
```

```
print('Valid size', X_valid.shape, 'class proportions', np.bincount(y_valid))
print('Test size', X_test.shape, 'class proportions', np.bincount(y_test))
```

1.8 Data Scaling

- In the case of the Iris dataset, all dimensions were measured in centimeters, hence "scaling" features would not be necessary in the context of kNN – unless we want to weight features differently.
- Whether or not to scale features depends on the problem at hand and requires your judgement.
- However, there are several algorithms (especially gradient-descent, etc., which we will cover later in this course), which work much better (are more robust, numerically stable, and converge faster) if the data is centered and has a smaller range.
- There are many different ways for scaling features; here, we only cover to of the most common "normalization" schemes: min-max scaling and z-score standardization.

1.8.1 Normalization – Min-max scaling

- Min-max scaling squashes the features into a [0, 1] range, which can be achieved via the following equation for a single input i:

$$x^{[i]}_{\text{norm}} = \frac{x^{[i]} - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

- Below is an example of how we can implement and apply min-max scaling on 6 data instances given a 1D input vector (1 feature) via NumPy.

[ ]:
```
x = np.arange(6).astype(float)
x
```

[ ]:
```
x_norm = (x - x.min()) / (x.max() - x.min())
x_norm
```

1.8.2 Standardization

- Z-score standardization is a useful standardization scheme if we are working with certain optimization methods (e.g., gradient descent, later in this course).
- After standardizing a feature, it will have the properties of a standard normal distribution, that is, unit variance and zero mean ($N(\mu = 0, \sigma^2 = 1)$); however, this does not transform a feature from not following a normal distribution to a normal distributed one.
- The formula for standardizing a feature is shown below, for a single data point $x^{[i]}$.

$$x^{[i]}_{\text{std}} = \frac{x^{[i]} - \mu_x}{\sigma_x}$$

[ ]:
```
x = np.arange(6).astype(float)
x
```

[ ]:
```
x_std = (x - x.mean()) / x.std()
x_std
```

- Conveniently, NumPy and Pandas both implement a `std` method, which computes the standard devation.
- Note the different results shown below.

[ ]:
```
df = pd.DataFrame([1, 2, 1, 2, 3, 4])
df[0].std()
```

[ ]:
```
df[0].values.std()
```

- The results differ because Pandas computes the "sample" standard deviation (s_x), whereas NumPy computes the "population" standard deviation (σ_x).

$$s_x = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x^{[i]} - \bar{x})^2}$$

$$\sigma_x = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x^{[i]} - \mu_x)^2}$$

- In the context of machine learning, since we are typically working with large datasets, we typically don't care about Bessel's correction (subtracting one degree of freedom in the denominator).
- Further, the goal here is not to model a particular distribution or estimate distribution parameters accurately; however, if you like, you can remove the extra degree of freedom via NumPy's `ddof` parameters – it's not necessary in practice though.

[ ]:
```
df[0].values.std(ddof=1)
```

- A concept that is very important though is how we use the estimated normalization parameters (e.g., mean and standard deviation in z-score standardization).
- In particular, it is important that we re-use the parameters estimated from the training set to transfrom validation and test sets – re-estimating the parameters is a common "beginner-mistake" which is why we discuss it in more detail.

[ ]:
```
mu, sigma = X_train.mean(axis=0), X_train.std(axis=0)

X_train_std = (X_train - mu) / sigma
X_valid_std = (X_valid - mu) / sigma
X_test_std = (X_test - mu) / sigma
```

- Again, if we standardize the training dataset, we need to keep the parameters (mean and standard deviation for each feature). Then, we'd use these parameters to transform our test data and any future data later on

- Let's assume we have a simple training set consisting of 3 samples with 1 feature column (let's call the feature column "length in cm"):
- example1: 10 cm -> class 2
- example2: 20 cm -> class 2
- example3: 30 cm -> class 1

Given the data above, we estimate the following parameters from this training set:

- mean: 20
- standard deviation: 8.2

If we use these parameters to standardize the same dataset, we get the following z-score values:

- example1: -1.21 -> class 2
- example2: 0 -> class 2
- example3: 1.21 -> class 1

Now, let's say our model has learned the following hypotheses: It classifies samples with a standardized length value < 0.6 as class 2 (and class 1 otherwise). So far so good. Now, let's imagine we have 3 new unlabeled data points that you want to classify.

- example4: 5 cm -> class ?
- example5: 6 cm -> class ?
- example6: 7 cm -> class ?

If we look at the non-standardized "length in cm" values in the training datast, it is intuitive to say that all of these examples (5, 6, and 7) are likely belonging to class 2 because they are smaller than anything in the training set. However, if we standardize these by re-computing the standard deviation and and mean from the new data, we will get similar values as before (i.e., properties of a standard normal distribtion) in the training set and our classifier would (probably incorrectly) assign the "class 2" label to the samples 4 and 5.

- example5: -1.21 -> class 2
- example6: 0 -> class 2
- example7: 1.21 -> class 1

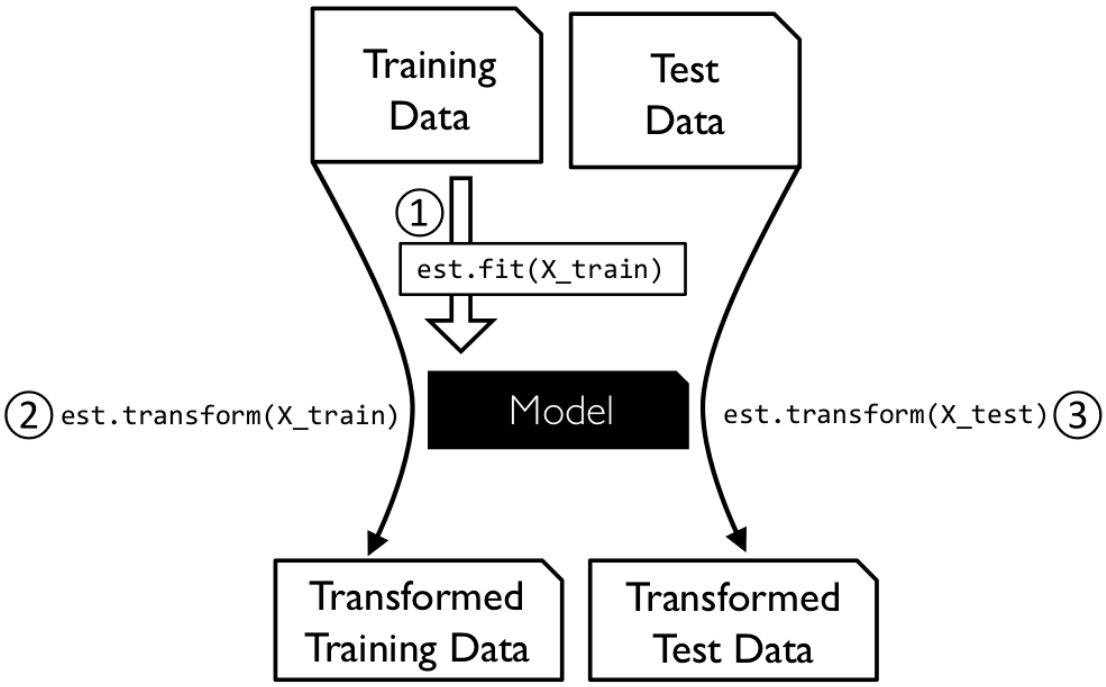
However, if we use the parameters from the "training set standardization," we will get the following standardized values

- example5: -18.37
- example6: -17.15
- example7: -15.92

Note that these values are more negative than the value of example1 in the original training set, which makes much more sense now!

1.8.3 Scikit-Learn Transformer API

- The transformer API in scikit-learn is very similar to the estimator API; the main difference is that transformers are typically "unsupervised," meaning, they don't make use of class labels or target values.



- Typical examples of transformers in scikit-learn are the `MinMaxScaler` and the `StandardScaler`, which can be used to perform min-max scaling and z-score standardization as discussed earlier.

[ ]:
```
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler


scaler = StandardScaler()
scaler.fit(X_train)
X_train_std = scaler.transform(X_train)
X_valid_std = scaler.transform(X_valid)
X_test_std = scaler.transform(X_test)
```

1.9 Categorical Data

- When we preprocess a dataset as input to a machine learning algorithm, we have to be careful how we treat categorical variables.
- There are two broad categories of categorical variables: nominal (no order implied) and ordinal (order implied).

[ ]:
```
df = pd.read_csv('categoricaldata.csv')
df
```

- In the example above, 'size' would be an example of an ordinal variable; i.e., if the letters refer to T-shirt sizes, it would make sense to come up with an ordering like M $<$ L $<$ XXL.

- Hence, we can assign increasing values to a ordinal values; however, the range and difference between categories depends on our domain knowledge and judgement.
- To convert ordinal variables into a proper representation for numerical computations via machine learning algorithms, we can use the now familiar `map` method in Pandas, as shown below.

[ ]:
```
mapping_dict = {'M': 2,
                'L': 3,
                'XXL': 5}

df['size'] = df['size'].map(mapping_dict)
df
```

- Machine learning algorithms do not assume an ordering in the case of class labels.
- Here, we can use the `LabelEncoder` from scikit-learn to convert class labels to integers as an alternative to using the `map` method

[ ]:
```
from sklearn.preprocessing import LabelEncoder


le = LabelEncoder()
df['classlabel'] = le.fit_transform(df['classlabel'])
df
```

- Representing nominal variables properly is a bit more tricky.
- Since machine learning algorithms usually assume an order if a variable takes on integer values, we need to apply a "trick" here such that the algorithm would not make this assumption.
- this "trick" is also called "one-hot" encoding – we binarize a nominal variable, as shown below for the color variable (again, we do this because some ordering like orange $<$ red $<$ blue would not make sense in many applications).

[ ]:
```
pd.get_dummies(df)
```

- Note that executing the code above produced 3 new variables for "color," each of which takes on binary values.
- However, there is some redundancy now (e.g., if we know the values for `color_green` and `color_red`, we automatically know the value for `color_blue`).
- While collinearity may cause problems (i.e., the matrix inverse doesn't exist in e.g., the context of the closed-form of linear regression), again, in machine learning we typically would not care about it too much, because most algorithms can deal with collinearity (e.g., adding constraints like regularization penalties to regression models, which we learn via gradient-based optimization).
- However, removing collinearity if possible is never a bad idea, and we can do this conveniently by dropping e.g., one of the columns of the one-hot encoded variable.

[ ]:
```
pd.get_dummies(df, drop_first=True)
```

1.10 Missing Data

- There are many different ways for dealing with missing data.
- The simplest approaches are removing entire columns or rows.
- Another simple approach is to impute missing values via the feature means, medians, mode, etc.
- There is no rule or best practice, and the choice of the approprite missing data imputation method depends on your judgement and domain knowledge.
- Below are some examples for dealing with missing data.

[ ]:
```
df = pd.read_csv('missingdata.csv')
df
```

[ ]:
```
# missing values per column:

df.isnull().sum()
```

[ ]:
```
# drop rows with missing values:

df.dropna(axis=0)
```

[ ]:
```
# drop columns with missing values:

df.dropna(axis=1)
```

[ ]:
```
from sklearn.preprocessing import Imputer


imputer = Imputer(missing_values='NaN', strategy='mean', axis=0)
X = df.values
X = imputer.fit_transform(df.values)
X
```

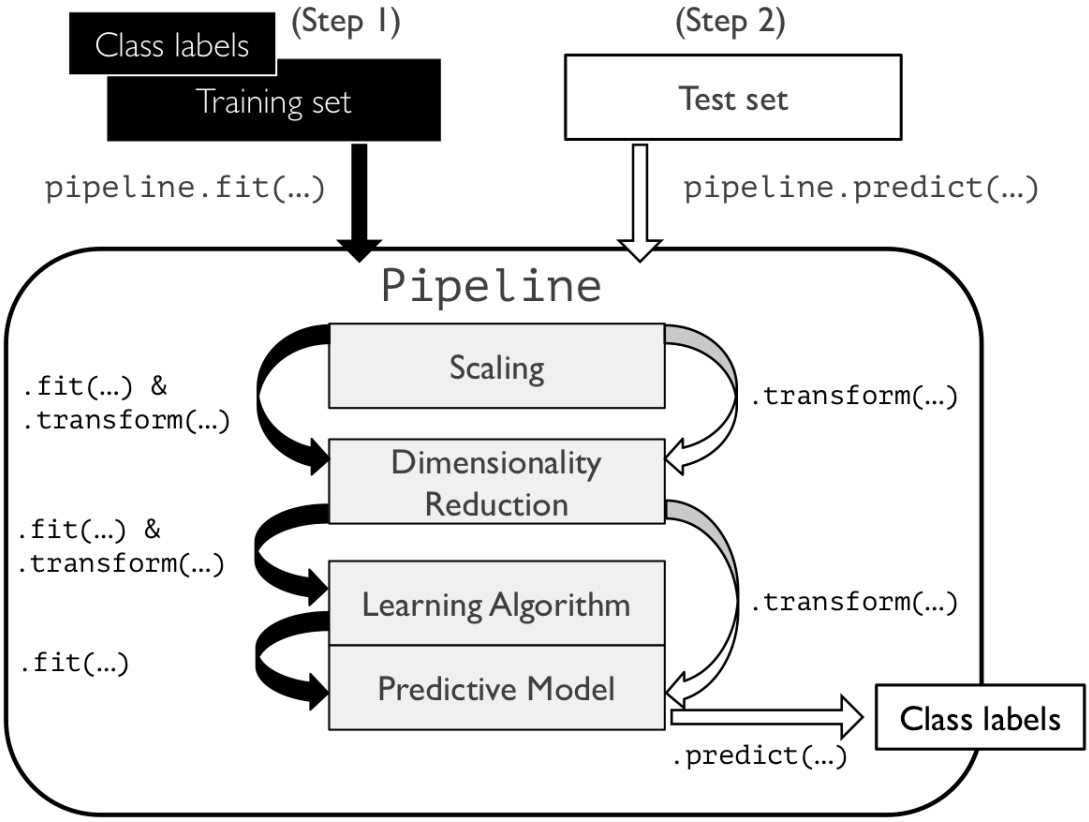
1.11 Feature Transformation, Extraction, and Selection

We have already covered very simple cases of feature transformation, i.e., normalization, that is, min-max scaling and standardization. There are many other cases, but an extensive coverage of feature preprocessing is beyond the scope of a machine learning class. However, we will will look at some popular feature selection (sequential feature selection) and feature extraction (e.g., principal component analysis) techniques later in this course.

1.12 Scikit-Learn Pipelines

- Scikit-learn pipelines are an extremely convenient and powerful concept – one of the things that sets scikit-learn apart from other machine learning libraries.
- Pipelines basically let us define a series of perprocessing steps together with fitting an estimator.

- Pipelines will automatically take care of pitfalls like estimating feature scaling parameters from the training set and applying those to scale new data (which we discussed earlier in the context of z-score standardization).
- Below is an visualization of how pipelines work.



- Below is an example pipeline that combines the feature scaling step with the kNN classifier.

[ ]:
```
from sklearn.pipeline import make_pipeline


pipe = make_pipeline(StandardScaler(),
                     KNeighborsClassifier(n_neighbors=3))
```

[ ]:
```
pipe
```

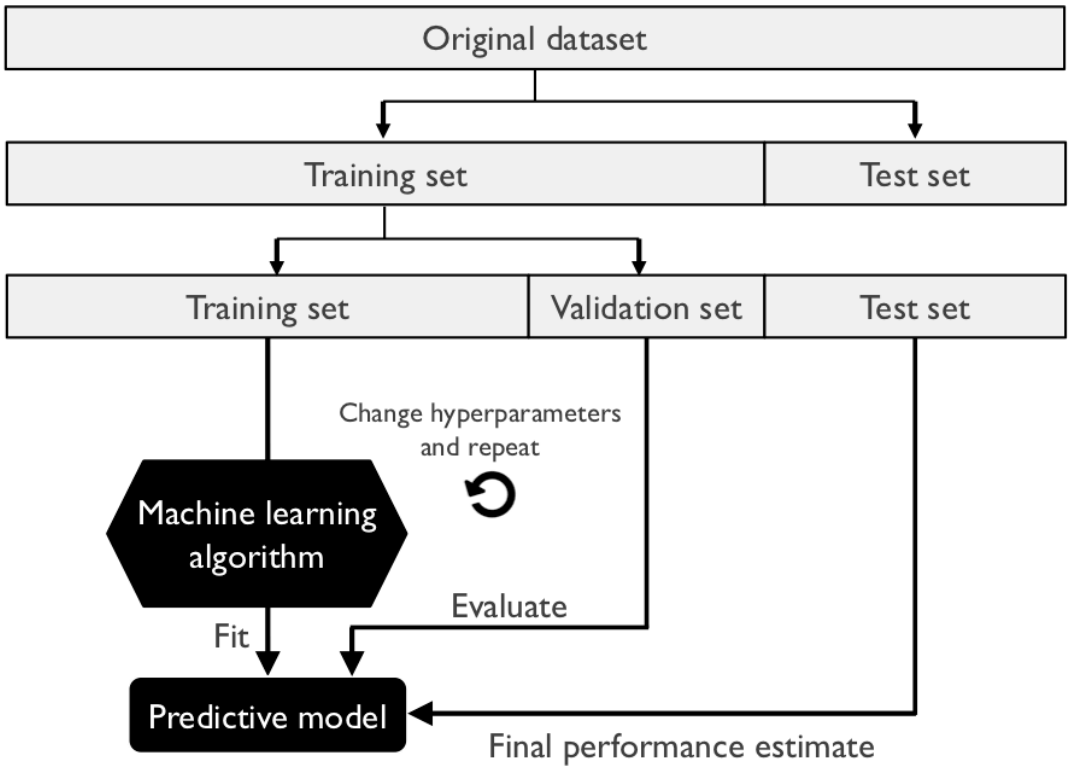
[ ]:
```
pipe.fit(X_train, y_train)
pipe.predict(X_test)
```

- As you can see above, the Pipeline itself follows the scikit-learn estimator API.

(Also see the FunctionTransformer in scikit-learn, which allows creating a transformer class from an arbitrary callable or function.)

1.13 Intro Model Selection – Pipelines and Grid Search

- In machine learning practice, we often need to experiment with an machine learning algorithm's hyperparameters to find a good setting.
- The process of tuning hyperparameters and comparing and selecting the resulting models is also called "model selection" (in contrast to "algorithm selection").
- We will cover topics such as "model selection" and "algorithm selection" in more detail later in this course.
- For now, we are introducing the simplest way of performing model selection: using the "holdout method."
- In the holdout method, we split a dataset into 3 subsets: a training, a validation, and a test datatset.
- To avoid biasing the estimate of the generalization performance, we only want to use the test dataset once, which is why we use the validation dataset for hyperparameter tuning (model selection).
- Here, the validation dataset serves as an estimate of the generalization performance, too, but it becomes more biased than the final estimate on the test data because of its repeated re-use during model selection (think of "multiple hypothesis testing").



[ ]:
```
from sklearn.model_selection import GridSearchCV
from mlxtend.evaluate import PredefinedHoldoutSplit
from sklearn.pipeline import make_pipeline
from sklearn.datasets import load_iris

iris = load_iris()
```

```
X, y = iris.data, iris.target

train_ind, valid_ind = train_test_split(np.arange(X.shape[0]),
                                        test_size=0.2, shuffle=True,
                                        random_state=123, stratify=y)

pipe = make_pipeline(StandardScaler(),
                     KNeighborsClassifier())

params = {'kneighborsclassifier__n_neighbors': [1, 3, 5],
          'kneighborsclassifier__p': [1, 2]}

split = PredefinedHoldoutSplit(valid_indices=valid_ind)

grid = GridSearchCV(pipe,
                    param_grid=params,
                    cv=split)

grid.fit(X, y)
grid.grid_scores_
```

1.14 Reading Assignment

- Python Machine Learning 2nd ed.: Ch04 up to "Selecting Meaningful Features" (pg 107-123)
- Python Machine Learning 2nd ed.: Ch06 up to "Debugging algorithms with learning and validation curves" (pg 185-194)

1.15 Further Resources

- Scikit-learn documentation: http://scikit-learn.org/stable/documentation.html