

End to end Machine Learning Project

Machine Learning Practice Course

Outline

1. Steps in ML projects
2. Illustration through practical set up

ML Project

- Excellent wine company wants to develop ML model for ***predicting wine quality*** on certain ***physiochemical characteristics*** in order to replace expensive quality sensor.
- Let's understand steps involved in addressing this problem.

Steps in ML projects

1. Look at the big picture.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor and maintain your system.



A few words of wisdom



- ML is usually a small piece in a big project. e.g. wine quality prediction is a small piece in setting up the manufacturing process.
- Typically 10-15% of time is spent on ML.
- A lot more time is spent on capturing and processing data needed for ML and taking decisions based on output of ML module.
- Needs strong collaboration with domain experts, product managers and eng-teams for successful execution.

Step 1: Look at the big picture

1. Frame the problem
2. Select a performance measure
3. List and check the assumptions

1.1 Frame the problem

- What is input and output?
- What is the business objective? How does company expects to use and benefit from the model?
 - Useful in problem framing
 - Algorithm and performance measure selection
 - Overall effort estimation
- What is the current solution (if any)?
 - Provides a useful baseline

Design consideration in problem framing

- Is this a **supervised, unsupervised or a RL** problem?
- Is this a **classification, regression** or some other task?
- What is the nature of the output: **single** or **multiple** outputs?
- Does system need **continuous learning** or **periodic updates**?
- What would be the learning style: **batch** or **online**?

1.2 Selection of performance measure

- Regression
 - Mean Squared Error (MSE) or
 - Mean Absolute Error (MAE)
- Classification
 - Precision
 - Recall
 - F1-score
 - Accuracy

1.3 Check the assumptions

- List down various assumptions about the task.
- Review with domain experts and other teams that plan to consume ML output.
- Make sure all assumptions are reviewed and approved before coding!

Step 2: Get the data

- Data spread across multiple tables, files or documents with access control.
- Obtain appropriate access controls and authorizations.
- Get familiarized with data by looking at schema and a few rows. (Familiarity with SQL would be useful here.)

Load basic libraries



```
1 import pandas as pd  
2 import matplotlib.pyplot as plt  
3 import seaborn as sns  
4 import numpy as np
```

- Let's first access our data - in this case, we need to download it from the web.
- It's a good practice to create a function for downloading and extracting the data.

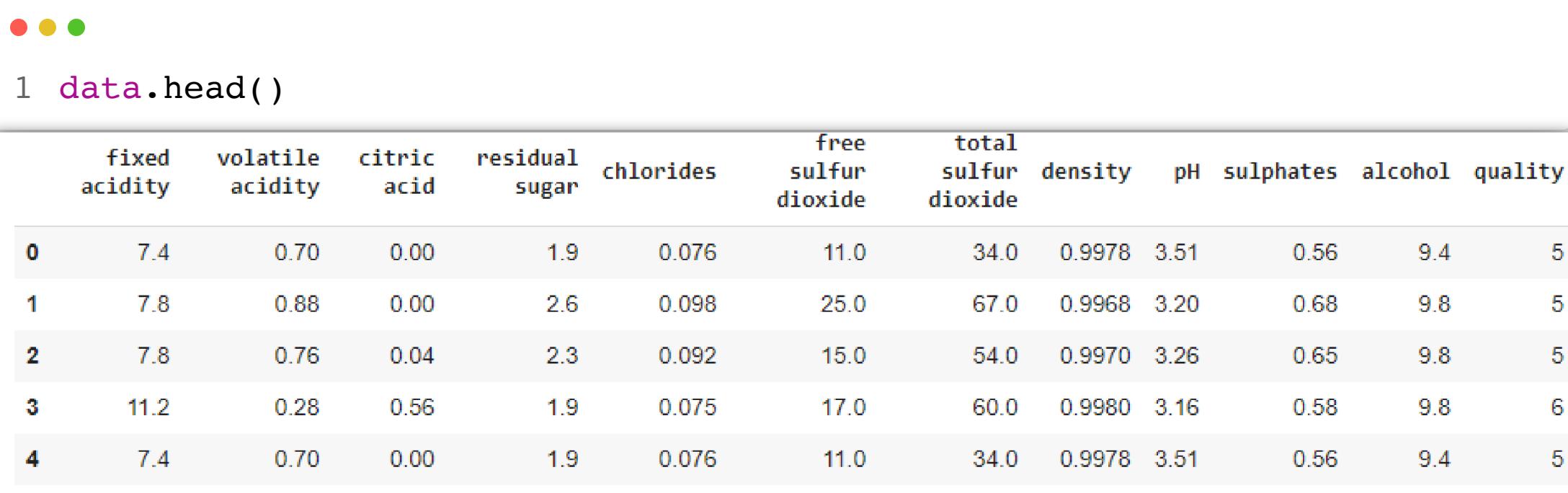


```
1 data_url = 'https://archive.ics.uci.edu/ml/machine-learning-  
databases/wine-quality/winequality-red.csv'  
2 data = pd.read_csv(data_url, sep=";")
```

Now that the data is loaded, let's examine it.

2.1 Check data samples

Let's look at a few data samples with head() method.



The screenshot shows a Jupyter Notebook cell with three colored dots (red, yellow, green) indicating the status of the cell. The code `1 data.head()` is run, and the output displays the first five rows of a dataset. The columns represent wine quality features: fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, alcohol, and quality.

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

2.2 Features

It's a good idea to understand significance of each feature by consulting the experts.

Feature	Significance
Fixed acidity	Most acids involved with wine are fixed or nonvolatile (do not evaporate readily)
Volatile acidity	The amount of acetic acid in wine, which at too high of levels can lead to an unpleasant, vinegar taste
Citric acid	Found in small quantities, citric acid can add 'freshness' and flavor to wines
Residual sugar	It's rare to find wines with less than 1 gram/liter and wines with greater than 45 grams/liter are considered sweet.
Chlorides	The amount of salt in the wine.
•	•
Alcohol	The percentage of alcohol contents in the wine.



```
1 feature_list = data.columns[:-1].values  
2 label = [data.columns[-1]]  
3  
4 print ("Feature list:", feature_list)  
5 print ("Label:", label)
```

Feature list: ['fixed acidity' 'volatile acidity' 'citric acid'
'residual sugar' 'chlorides' 'free sulfur dioxide' 'total
sulfur dioxide' 'density' 'pH' 'sulphates' 'alcohol'] **Label:**
['quality']

2.3 Data statistics

Let's use *info()* method to get quick description of data.

```
1 data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   fixed acidity    1599 non-null    float64 
 1   volatile acidity 1599 non-null    float64 
 2   citric acid      1599 non-null    float64 
 3   residual sugar   1599 non-null    float64 
 4   chlorides        1599 non-null    float64 
 5   free sulfur dioxide 1599 non-null    float64 
 6   total sulfur dioxide 1599 non-null    float64 
 7   density          1599 non-null    float64 
 8   pH               1599 non-null    float64 
 9   sulphates        1599 non-null    float64 
 10  alcohol          1599 non-null    float64 
 11  quality          1599 non-null    int64  
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

2.3 Data statistics

- Total entries: 1599 (Tiny dataset by ML standard)
- There are total 12 columns: 11 features + 1 label
 - Label column: **quality**
 - Features: **[fixed acidity, volatile acidity, citric acid, residual sugar, cholrides, free sulphur dioxide, total sulphur dioxide, density, pH, sulphates, alcohol]**
- All columns are numeric (float64) and label is an integer.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column           Non-Null Count Dtype  
 ---  -----          -----          Dtype   
 0   fixed acidity    1599 non-null   float64 
 1   volatile acidity 1599 non-null   float64 
 2   citric acid      1599 non-null   float64 
 3   residual sugar   1599 non-null   float64 
 4   chlorides        1599 non-null   float64 
 5   free sulfur dioxide 1599 non-null   float64 
 6   total sulfur dioxide 1599 non-null   float64 
 7   density          1599 non-null   float64 
 8   pH               1599 non-null   float64 
 9   sulphates        1599 non-null   float64 
 10  alcohol          1599 non-null   float64 
 11  quality          1599 non-null   int64  
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

In order to understand nature of numeric attributes, we use **describe()** method.



1 `data.describe()`

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000
mean	8.319637	0.527821	0.270976	2.538806	0.087467	15.874922	46.467792	0.996747	3.311113
std	1.741096	0.179060	0.194801	1.409928	0.047065	10.460157	32.895324	0.001887	0.154386
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.000000	0.990070	2.740000
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22.000000	0.995600	3.210000
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38.000000	0.996750	3.310000
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	62.000000	0.997835	3.400000
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289.000000	1.003690	4.010000

This one prints count and statistical properties - mean, standard deviations and quartiles.

- The wine quality can be between **0** and **10**, but in this dataset, the quality values are between 3 and 8. Let's look at the distribution of examples by the wine quality.



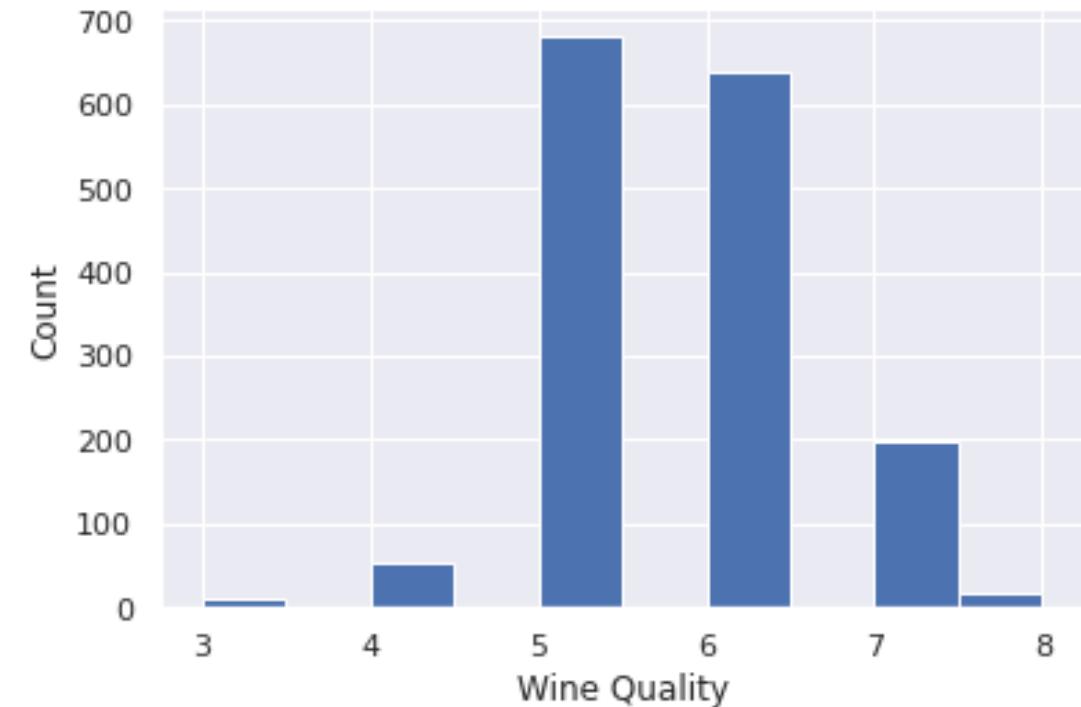
```
1 data[ 'quality' ].value_counts()  
  
5    681  
6    638  
7    199  
4     53  
8     18  
3     10  
Name: quality, dtype: int64
```

- High quality value → better quality of wine
- You can see that there are lots of samples of average wines than good or the poor quality ones.
 - Many examples with quality = 5 or 6

The information can be viewed through histogram plot.

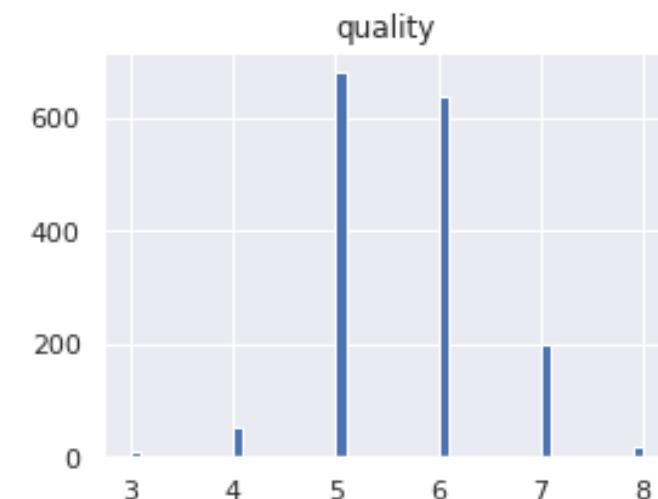
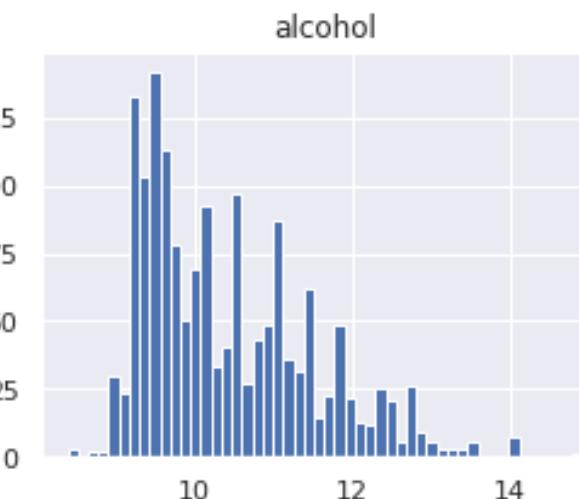
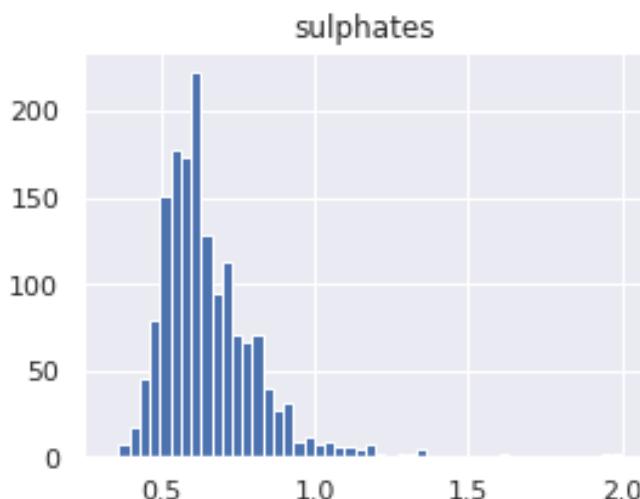
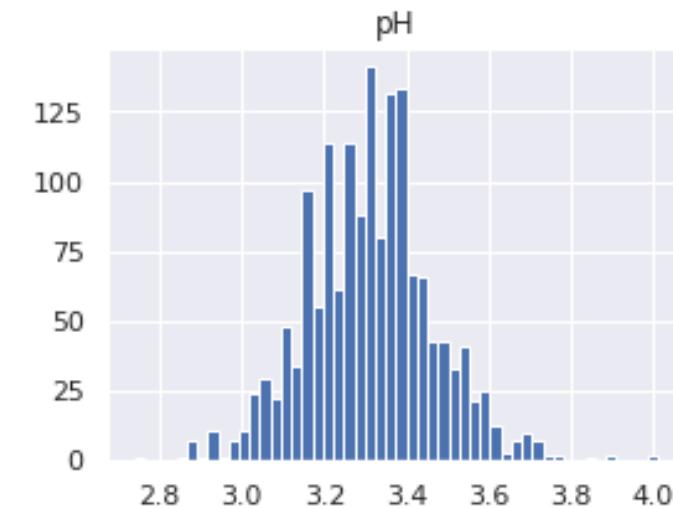
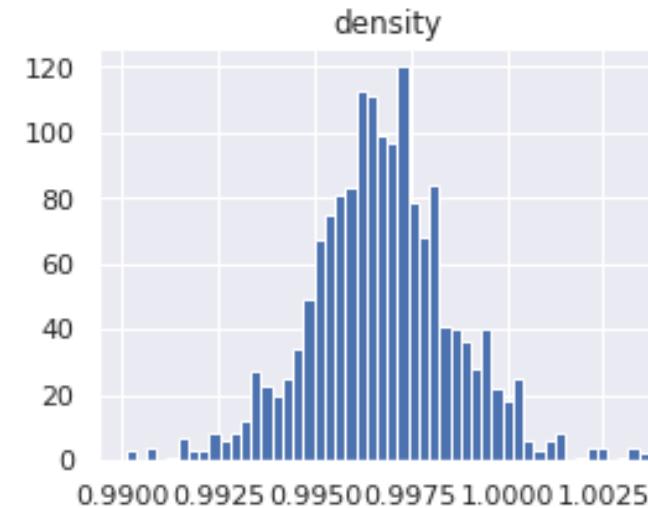
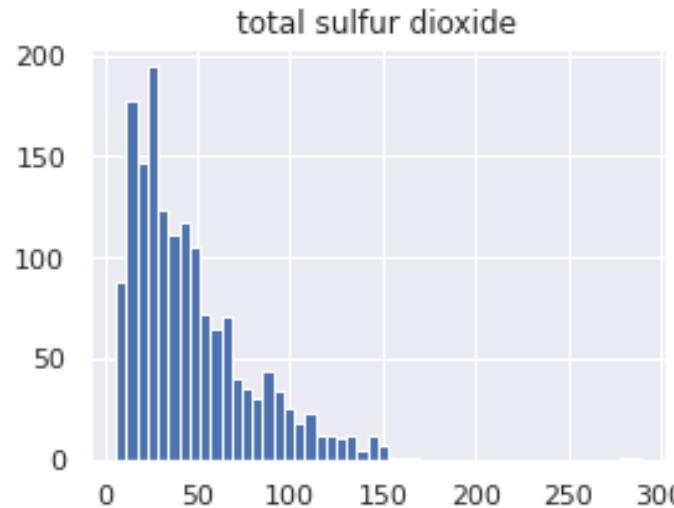
- A Histogram gives the count of how many samples occurs within a specific range (bins).
- The x-axis denotes the range of values in a feature and
- The y-axis denotes the frequency of samples with those specific values.

```
● ● ●  
1 sns.set()  
2 data.quality.hist()  
3 plt.xlabel('Wine Quality')  
4 plt.ylabel('Count')
```



Note taller bars for quality 5 and 6 compared to the other qualities.

In a similar manner, we can plot all numerical attributes with histogram plot for quick examination.



A few observations based on these plots:

1. Features are at different scales.
2. Features have different distributions -
 - A few are tail heavy. e.g. *residual sugar, free so2*
 - A few have multiple modes. e.g. *volatile acidity, citric acid*

Before any further exploration, it's a good idea to separate test set and do not look at it in order to have a clean evaluation set.

2.4 Create test set

- When we look at the test set, we are likely to notice patterns in that and based on that we may select certain models.
- This leads to biased estimation on test set, which may not generalize well in practice. This is called **data snooping bias**.

Let's write a function to split the data into training and test. Make sure to set the seed so that we get the same test set in the next run.



```
1 def split_train_test(data, test_ratio):
2     # set the random seed.
3     np.random.seed(42)
4
5     # shuffle the dataset.
6     shuffled_indices = np.random.permutation(len(data))
7
8     # calculate the size of the test set.
9     test_set_size = int(len(data) * test_ratio)
10
11    # split dataset to get training and test sets.
12    test_indices = shuffled_indices[:test_set_size]
13    train_indices = shuffled_indices[test_set_size:]
14    return data.iloc[train_indices], data.iloc[test_indices]
```



```
1 train_set, test_set = split_train_test(data, 0.2)
```

Scikit-Learn provides a few functions for creating test sets based on

1. **Random sampling**, which randomly selects $k\%$ points in the test set.
2. **Stratified sampling**, which samples test examples such that they are representative of overall distribution.

Random sampling

- `train_test_split()` function performs random sampling with
 - **random_state** parameter to set the random seed, which ensures that the same examples are selected for test sets across runs.
 - **test_size** parameter for specifying size of the test set.
 - **shuffle flag** to specify if the data needs to be shuffled before splitting.
- Provision for processing multiple datasets with an identical number of rows and selecting the same indices from these datasets.
 - Useful when labels are in different dataframe.



```
1 from sklearn.model_selection import train_test_split
```



```
1 from sklearn.model_selection import train_test_split
```

We can read the documentation for this function by using the following line of code:



```
1 ?train_test_split
```

Help X

Signature: `train_test_split(*arrays, **options)`

Docstring:

Split arrays or matrices into random train and test subsets

Quick utility that wraps input validation and
``next(ShuffleSplit().split(X, y))`` and application to
input data
into a single call for splitting (and optionally
subsampling) data in a
oneliner.

Read more in the :ref:`User Guide <cross_validation>`.

Parameters

`*arrays` : sequence of indexables with same length /
`shape[0]`

 Allowed inputs are lists, numpy arrays, scipy-
 sparse
 matrices or pandas dataframes.

Let's perform random sampling on our dataset:



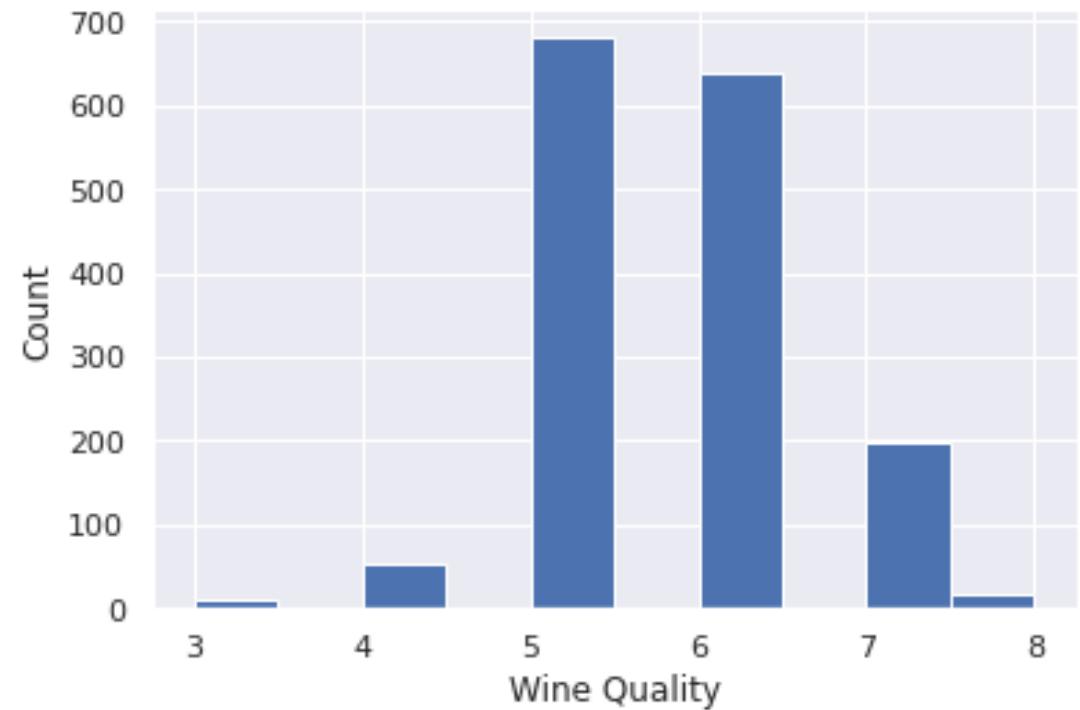
```
1 train_set, test_set = train_test_split(data, test_size=0.2, random_state=42)
```

Stratified sampling

- Data distribution may not be uniform in real world data.
- Random sampling - by its nature - introduces biases in such data sets.

Recall the label distribution in our dataset: It's not uniform!

```
● ● ●  
1 sns.set()  
2 data.quality.hist()  
3 plt.xlabel('Wine Quality')  
4 plt.ylabel('Count')
```



- Many examples of class 5 and 6 compared to the other classes.
- This causes a problem while random sampling. The test distribution may not match with the overall distribution.

How do we sample in such cases?

- We divide the population into homogenous groups called **strata**.
- Data is sampled from each stratum so as to match it with the overall data distribution.
- Scikit-Learn provides a class *StratifiedShuffleSplit* that helps us in stratified sampling.



```
1 from sklearn.model_selection import StratifiedShuffleSplit  
2 split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)  
3 for train_index, test_index in split.split(data, data["quality"]):  
4     strat_train_set = data.loc[train_index]  
5     strat_test_set = data.loc[test_index]
```

Let's examine the test set distribution by the wine quality that was used for stratified sampling.



```
1 strat_dist = strat_test_set["quality"].value_counts() / len(strat_test_set)
```

Now compare this with the overall distribution:



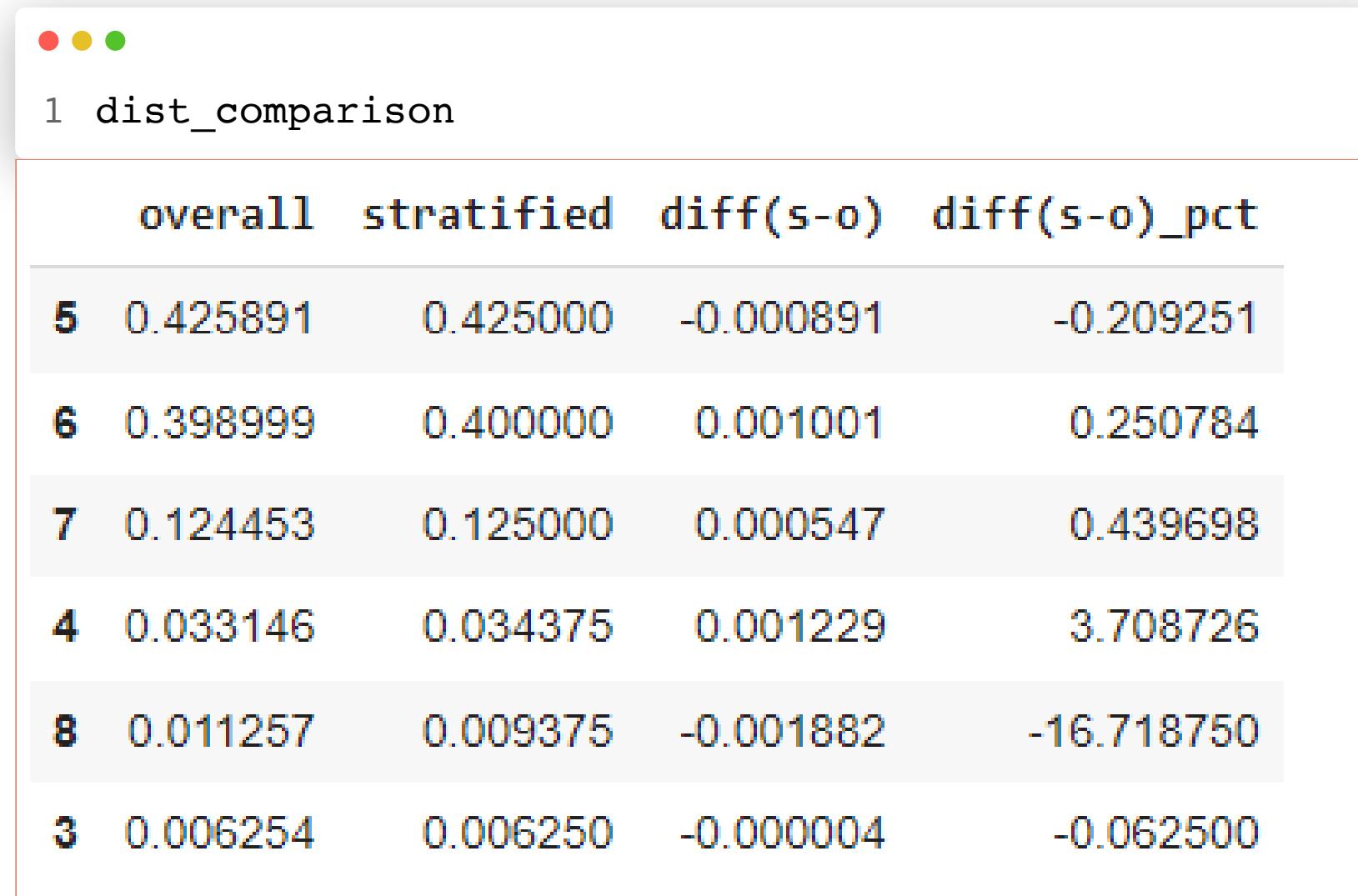
```
1 overall_dist = data["quality"].value_counts() / len(data)
```

Let's look at them side-by-side:



```
1 dist_comparison = pd.DataFrame({'overall': overall_dist, 'stratified': strat_dist})
2 dist_comparison['diff(s-o)'] = dist_comparison['stratified'] - dist_comparison['overall']
3 dist_comparison['diff(s-o)_pct'] = 100*(dist_comparison['diff(s-o)']/dist_comparison['overall'])
```

You can notice that there is a small difference in most strata.



The screenshot shows a Jupyter Notebook cell with a title "1 dist_comparison". The cell contains a Pandas DataFrame with six rows and five columns. The columns are labeled "overall", "stratified", "diff(s-o)", and "diff(s-o)_pct". The rows are numbered 5 through 3 from top to bottom. The "diff(s-o)" column shows small numerical values, while the "diff(s-o)_pct" column shows larger values ranging from -16.718750 to 3.708726. The rows are highlighted with alternating background colors.

	overall	stratified	diff(s-o)	diff(s-o)_pct
5	0.425891	0.425000	-0.000891	-0.209251
6	0.398999	0.400000	0.001001	0.250784
7	0.124453	0.125000	0.000547	0.439698
4	0.033146	0.034375	0.001229	3.708726
8	0.011257	0.009375	-0.001882	-16.718750
3	0.006254	0.006250	-0.000004	-0.062500

Let's contrast this with random sampling:



```
1 random_dist = test_set["quality"].value_counts() / len(test_set)
2 random_dist
```

```
6    0.412500
5    0.406250
7    0.131250
4    0.031250
8    0.015625
3    0.003125
Name: quality, dtype: float64
```

Sampling bias comparison

Compare the difference in distribution of stratified and uniform sampling:

- Stratified sampling gives us test distribution closer to the overall distribution than the random sampling.



```
1 dist_comparison.loc[:, ['diff(s-o)_pct', 'diff(r-o)_pct']]
```

	diff(s-o)_pct	diff(r-o)_pct
--	---------------	---------------

5	-0.209251	-4.611784
---	-----------	-----------

6	0.250784	3.383621
---	----------	----------

7	0.439698	5.461683
---	----------	----------

4	3.708726	-5.719340
---	----------	-----------

8	-16.718750	38.802083
---	------------	-----------

3	-0.062500	-50.031250
---	-----------	------------

Step 3: Data visualization

- Performed on training set.
- In case of large training set -
 - Sample examples to form **exploration set**.
- Enables to understand features and their relationship among themselves and with output label.

In our case, we have a small training data and we use it all for data exploration. There is no need to create a separate exploration set.

It's a good idea to create a copy of the training set so that we can freely manipulate it without worrying about any manipulation in the original set.



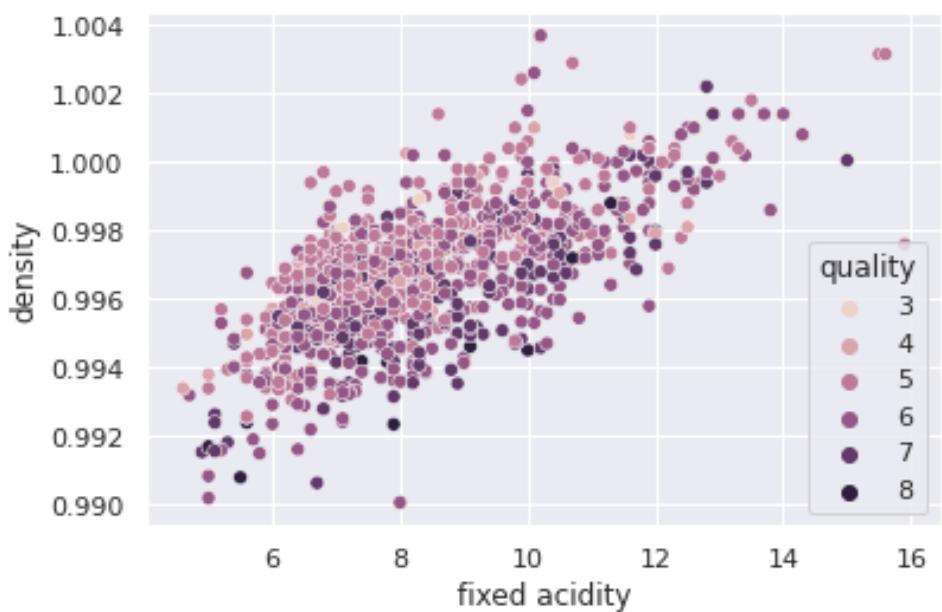
```
1 exploration_set = strat_train_set.copy()
```

Scatter Visualization

With seaborn library:

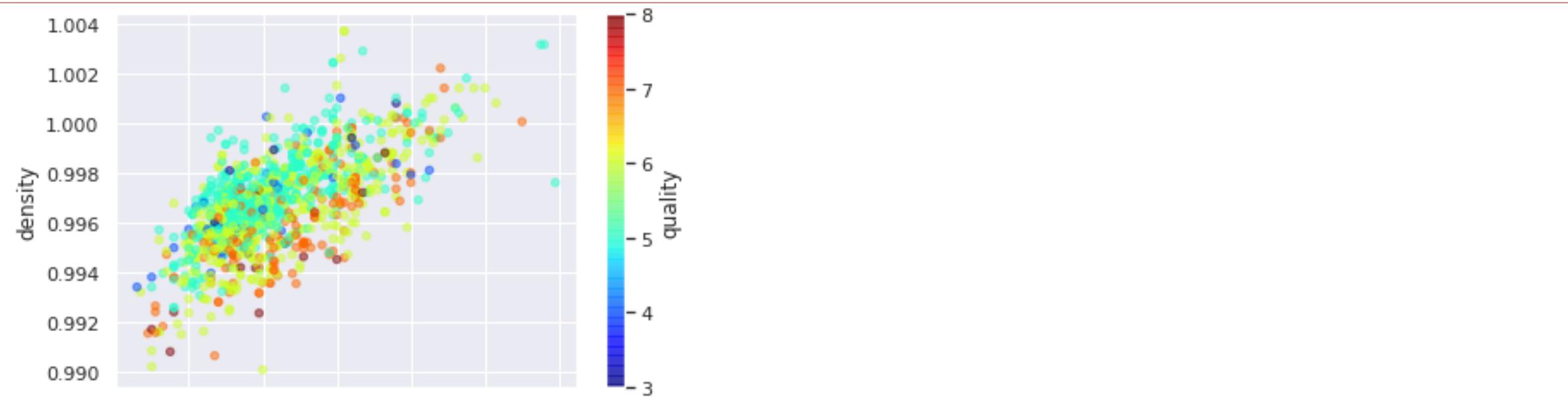


```
1 sns.scatterplot(x='fixed acidity', y='density', hue='quality',  
2                   data=exploration_set)
```



With matplotlib:

```
● ● ●  
1 exploration_set.plot(kind='scatter', x='fixed acidity', y='density', alpha=0.5,  
2 c="quality", cmap=plt.get_cmap("jet"))
```



Relationship between features

- Standard correlation coefficient between features.
 - Ranges between -1 to +1
 - Correlation = +1: Strong positive correlation between features
 - Correlation = -1: Strong negative correlation between features
 - Correlation = 0: No linear correlation between features
 - Visualization with heat map
- Only captures linear relationship between features.
 - For non-linear relationship, use rank correlation

Let's calculate correlations between our features.



```
1 corr_matrix = exploration_set.corr()
```

Let's check features that are correlated with the label, which is quality in our case.

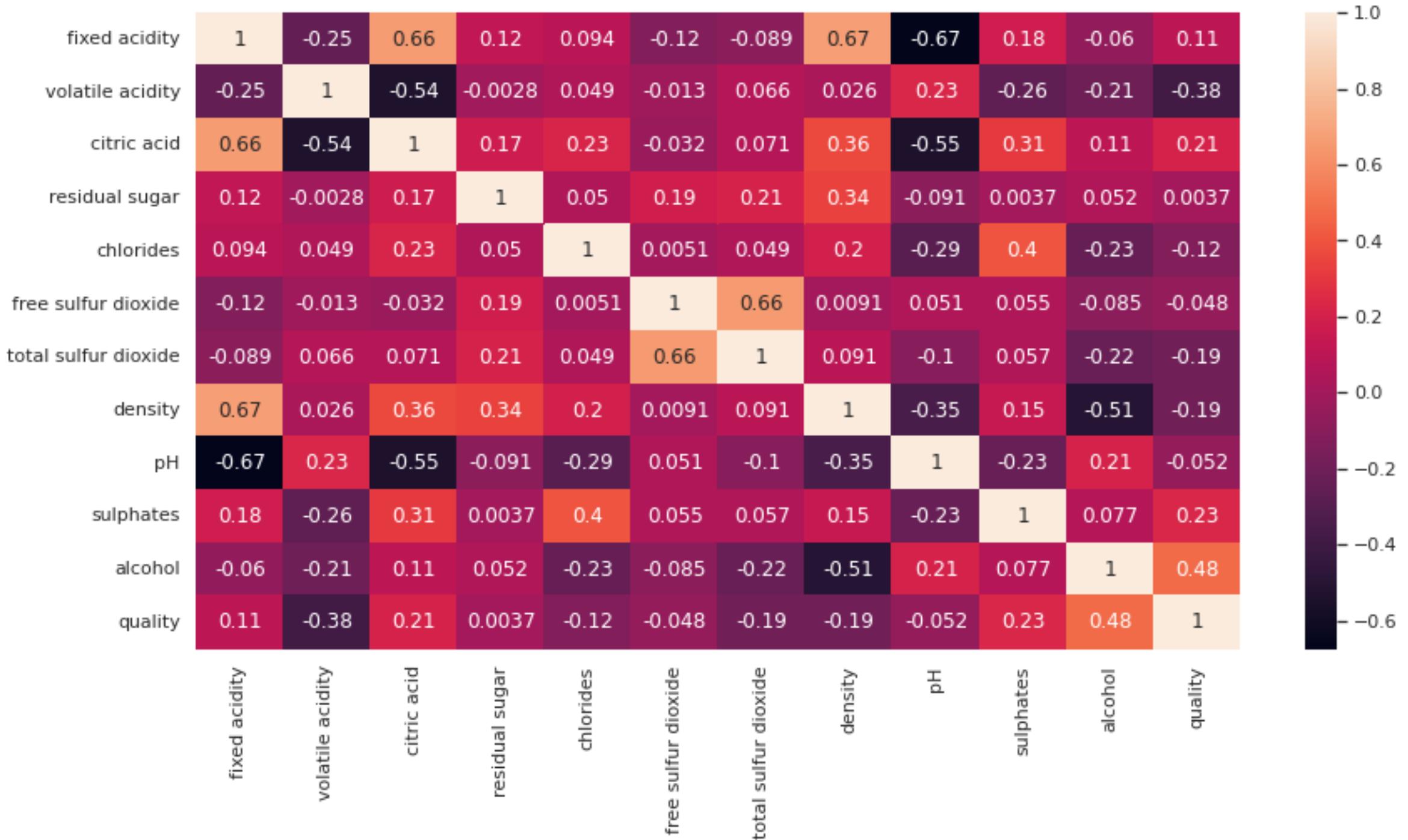
```
● ● ●  
1 corr_matrix['quality']  
  
fixed acidity          0.107940  
volatile acidity       -0.383249  
citric acid            0.210802  
residual sugar         0.003710  
chlorides               -0.120231  
free sulfur dioxide    -0.048291  
total sulfur dioxide   -0.194511  
density                 -0.193009  
pH                      -0.052063  
sulphates                0.228050  
alcohol                  0.481197  
quality                  1.000000  
Name: quality, dtype: float64
```

Notice that **quality** has strong positive correlation with **alcohol** content [0.48] and strong negative correlation with **volatile acidity** [-0.38].

Let's visualize correlation matrix with heatmap:



```
1 plt.figure(figsize=(14,7))  
2 sns.heatmap(corr_matrix, annot=True)
```



You can notice:

- The correlation coefficient on diagonal is +1.
- Darker colors represent negative correlations, while fainter colors denote positive correlations. For example
 - citric acid and fixed acidity have strong positive correlation.
 - pH and fixed acidity have strong negative correlation.

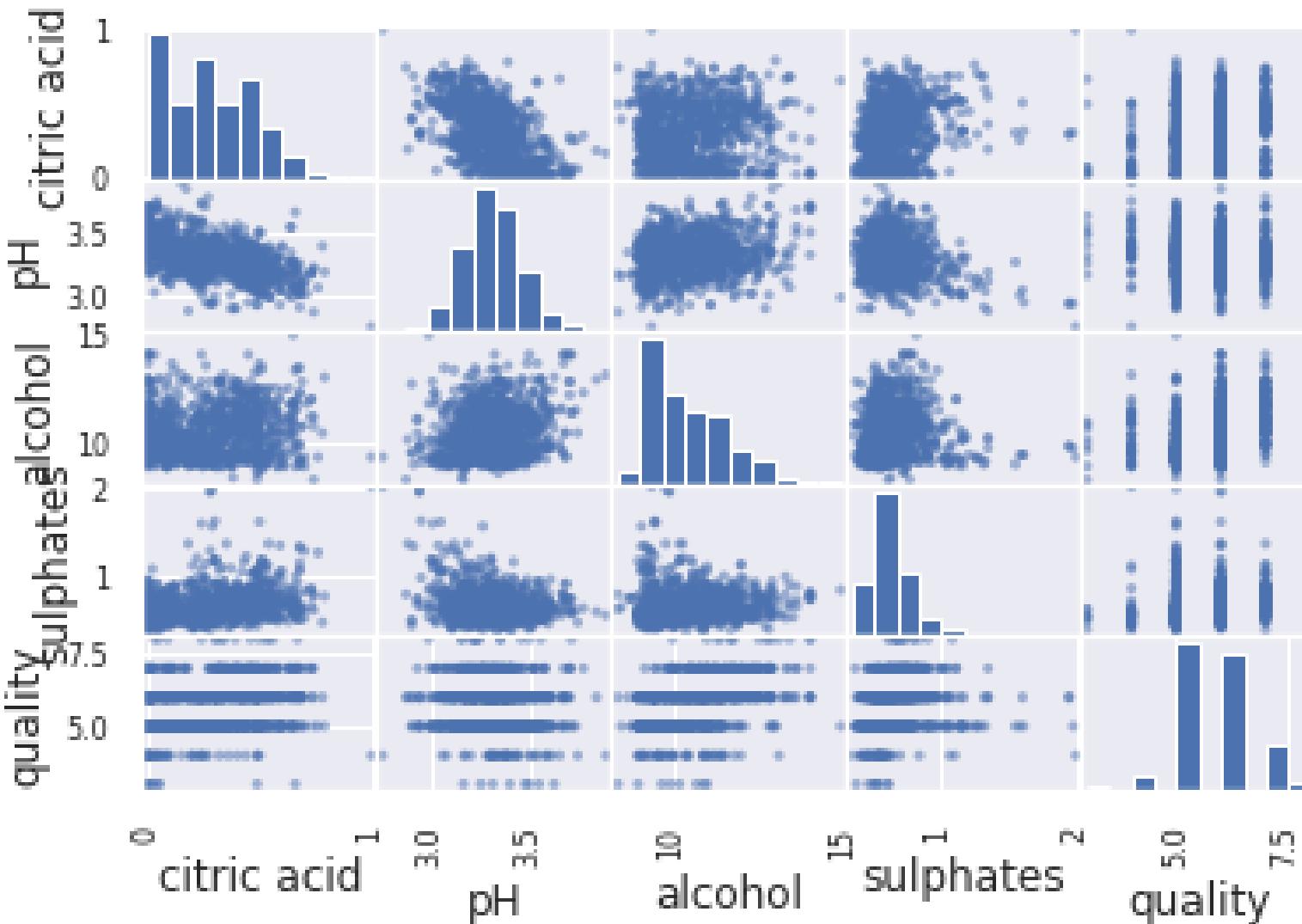
fixed acidity	1	-0.25	0.66
volatile acidity	-0.25	1	-0.54
citric acid	0.66	-0.54	1
residual sugar	0.12	-0.0028	0.17
chlorides	0.094	0.049	0.23
free sulfur dioxide	-0.12	-0.013	-0.032
total sulfur dioxide	-0.089	0.066	0.071
density	0.67	0.026	0.36
pH	-0.67	0.23	-0.55
sulphates	0.18	-0.26	0.31
alcohol	-0.06	-0.21	0.11
quality	0.11	-0.38	0.21

Another option to visualize the relationship between the feature is with scatter matrix.

fixed acidity
volatile acidity
citric acid



```
1 from pandas.plotting import scatter_matrix  
2 attribute_list = ['citric acid', 'pH', 'alcohol', 'sulphates', 'quality']  
3 scatter_matrix(exploration_set[attribute_list])
```



For convenience of visualization, we show it for a small number of attributes.

- Similar analysis can be carried out with combined features - features that are derived from the original features.

Note of wisdom

1. Visualization and data exploration do not have to be absolutely thorough.
2. Objective is to get quick insight into features and its relationship with other features and labels.
3. Exploration is an iterative process: Once we build model and obtain more insights, we can come back to this step.

Step 4: Prepare data for ML algorithm

We often need to preprocess the data before using it for model building due to variety of reasons:

- Due to errors in data capture, data may contain outliers or missing values.
- Different features may be at different scales.
- The current data distribution is not exactly amenable to learning.

Typical steps in data preprocessing are as follows:

1. Separate features and labels.
2. Handling missing values and outliers.
3. Feature scaling to bring all features on the same scale.
4. Applying certain transformations like log, square root on the features.

It's a good practice to make a copy of the data and apply preprocessing on that copy. This ensures that in case something goes wrong, we will at least have original copy of the data intact.

4.1 Separate features and labels from the training set.



```
1 # Copy all features leaving aside the label.  
2 wine_features = strat_train_set.drop("quality", axis=1)  
3  
4 # Copy the label list  
5 wine_labels = strat_train_set['quality'].copy()
```

4.2 Data cleaning

Let's first check if there are any missing values in feature set: One way to find that out is column-wise.



```
1 wine_features.isna().sum() # counts the number of NaN in each column of wine_fe
```

```
fixed acidity      0
volatile acidity   0
citric acid        0
residual sugar     0
chlorides          0
free sulfur dioxide 0
total sulfur dioxide 0
density            0
pH                 0
sulphates          0
alcohol            0
dtype: int64
```

In this dataset, we do not have any missing values.

In case, we have non-zero numbers in any columns, we have a problem of missing values.

- These values are missing due to errors in recording or they do not exist.
- If they are not recorded:
 - Use imputation technique to fill up the missing values.
 - Drop the rows containing missing values.
- If they do not exists, it is better to keep it as NaN.

Sklearn provides the following methods to drop rows containing missing values:

- *dropna()*
- *drop()*

It provides *SimpleImputer* class for filling up missing values with. say, median value.



```
1 from sklearn.impute import SimpleImputer  
2 imputer = SimpleImputer(strategy="median")
```

The strategy contains instructions as how to replace the missing values. In this case, we specify that the missing value should be replaced by the median value.



```
1 imputer.fit(wine_features)
```

```
SimpleImputer(add_indicator=False, copy=True, fill_value=None, missing_values=nan,  
strategy='median', verbose=0)
```

In case, the features contains non-numeric attributes, they need to be dropped before calling the fit method on imputer object.

Let's check the statistics learnt by the imputer on the training set:



```
1 imputer.statistics_
```

```
array([ 7.9 , 0.52 , 0.26 , 2.2 , 0.08 , 14. , 39. , 0.99675, 3.31 ,  
0.62 , 10.2 ])
```

Note that these are median values for each feature. We can cross-check it by calculating median on the feature set:



```
1 wine_features.median()
```

```
fixed acidity      7.90000  
volatile acidity   0.52000  
citric acid        0.26000  
residual sugar     2.20000  
chlorides          0.08000  
free sulfur dioxide 14.00000  
total sulfur dioxide 39.00000  
density            0.99675  
pH                 3.31000  
sulphates          0.62000  
alcohol            10.20000  
dtype: float64
```

Finally we use the trained imputer to transform the training set such that the missing values are replaced by the medians:



```
1 tr_features = imputer.transform(wine_features)
```

This returns a Numpy array and we can convert it to the dataframe if needed:



```
1 tr_features.shape
```

```
(1279, 11)
```



```
1 wine_features_tr = pd.DataFrame(tr_features, columns=wine_features.columns)
```

4.3 Handling text and categorical attributes

4.3.1 Converting categories to numbers:



```
1 from sklearn.preprocessing import OrdinalEncoder  
2 ordinal_encoder = OrdinalEncoder()
```

- Call `fit_transform()` method on `ordinal_encoder` object to convert text to numbers.
- The list of categories can be obtained via `categories_` instance variable.

One issue with this representation is that the ML algorithm would assume that the two nearby values are closer than the distinct ones.

4.3.2 Using one hot encoding

- Here we create one binary feature per category - the feature value is 1 when the category is present else it is 0.
- Only one feature is 1 (hot) and the rest are 0 (cold).
- The new features are referred to as *dummy features*.
- Scikit-Learn provides a **OneHotEncoder** class to convert categorical values into one-hot vectors.



```
1 from sklearn.preprocessing import OneHotEncoder  
2 cat_encoder = OneHotEncoder()
```

- We need to call **fit_transform()** method on **OneHotEncoder** object.
- The output is a SciPy sparse matrix rather than NumPy array. This enables us to save space when we have a huge number of categories.
- In case we want to convert it to dense representation, we can do so with **toarray()** method.
- The list of categories can be obtained via **categories_** instance variable.

- As we observed that when the number of categories are very large, the one-hot encoding would result in a very large number of features.
- This can be addressed with one of the following approaches:
 - Replace with categorical numerical features
 - Convert into low-dimensional learnable vectors called *embeddings*

4.4 Feature Scaling

- Most ML algorithms do not perform well when input features are on very different scales.
- Scaling of target label is generally not required.

4.5.1 Min-max scaling or Normalization

- We subtract minimum value of a feature from the current value and divide it by the difference between the minimum and the maximum value of that feature.
- Values are shifted and scaled so that they range between 0 and 1.
- Scikit-Learn provides **MinMaxScalar** transformer for this.
- One can specify hyperparameter **feature_range** to specify the range of the feature.

4.5.2 Standardization

- We subtract mean value of each feature from the current value and divide it by the standard deviation so that the resulting feature has a unit variance.
- While *normalization* bounds values between 0 and 1, *standardization* does not bound values to a specific range.
- Standardization is less affected by the outliers compared to the normalization.
- Scikit-Learn provides **StandardScalar** transformation for feature standardization.
- Note that all these transformers are learnt on the **training data** and then applied on the training and test data to transform them.
- **Never learn these transformers on the full dataset.**

Transformation Pipeline

- Scikit-Learn provides a Pipeline class to line up transformations in an intended order.
- Here is an example pipeline:



```
1 from sklearn.pipeline import Pipeline
2 from sklearn.preprocessing import StandardScaler
3 transform_pipeline = Pipeline([
4     ('imputer', SimpleImputer(strategy="median")),
5     ('std_scaler', StandardScaler()),])
6 wine_features_tr = transform_pipeline.fit_transform(wine_features)
```

Let's understand what is happening here:

- **Pipeline** has a sequence of transformations - missing value imputation followed by standardization.
- Each step in the sequence is defined by ***name, estimator*** pair.
- Each name should be unique and **should not contain __** (double underscore).



```
1 from sklearn.pipeline import Pipeline
2 from sklearn.preprocessing import StandardScaler
3 transform_pipeline = Pipeline([
4     ('imputer', SimpleImputer(strategy="median")),
5     ('std_scaler', StandardScaler()),])
6 wine_features_tr = transform_pipeline.fit_transform(wine_features)
```

- The output of one step is passed on the next one in sequence until it reaches the last step.
 - Here the pipeline first performs imputation of missing values and its result is passed for standardization.
- The pipeline exposes the same method as the final estimator.
 - Here **StandardScalar** is the last estimator and since it is a transformer, we call **fit_transform()** method on the **Pipeline** object.

How to transform mixed features?

- The real world data has both categorical as well as numerical features and we need to apply different transformations to them.
- Scikit-Learn introduced **ColumnTransformer** for this purpose.



```
1 from sklearn.compose import ColumnTransformer
```

In our dataset, we do not have features of mixed types. All our features are numeric.

For the illustration purpose, here is an example code snippet:



```
1 num_attribs = list(wine_features)
2 cat_attribs = ["place_of_manufacturing"]
3 full_pipeline = ColumnTransformer([
4     ("num", num_pipeline, num_attribs),
5     ("cat", OneHotEncoder(), cat_attribs),
6 ])
7 wine_features_tr = full_pipeline.fit_transform(wine_features)
```

- Here we apply `num_pipeline` on numerical features and `OneHotEncoder` transformation on the categorical features.
- The `ColumnTransformer` applies each transformation to the appropriate columns and then concatenates the outputs along the columns.
- Note that all transformers must return the same number of rows.
- The numeric transformers return dense matrix while the categorical ones return sparse matrix. The `ColumnTransformer` automatically determines the type of the output based on the density of the resulting matrix.

Step 5: Select and train ML model

- It's a good practice to build a quick baseline model on the preprocessed data and get an idea about model performance.



```
1 from sklearn.linear_model import LinearRegression  
2  
3 lin_reg = LinearRegression()  
4 lin_reg.fit(wine_features_tr, wine_labels)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,  
normalize=False)
```

Now that we have a working model of a regression, let's evaluate performance of the model on training as well as test sets.

- For regression models, we use mean squared error as an evaluation measure.



```
1 from sklearn.metrics import mean_squared_error  
2  
3 quality_predictions = lin_reg.predict(wine_features_tr)  
4 mean_squared_error(wine_labels, quality_predictions)
```

```
0.4206571060060278
```

Let's evaluate performance on the test set.

- We need to first apply transformation on the test set and then apply the model prediction function.

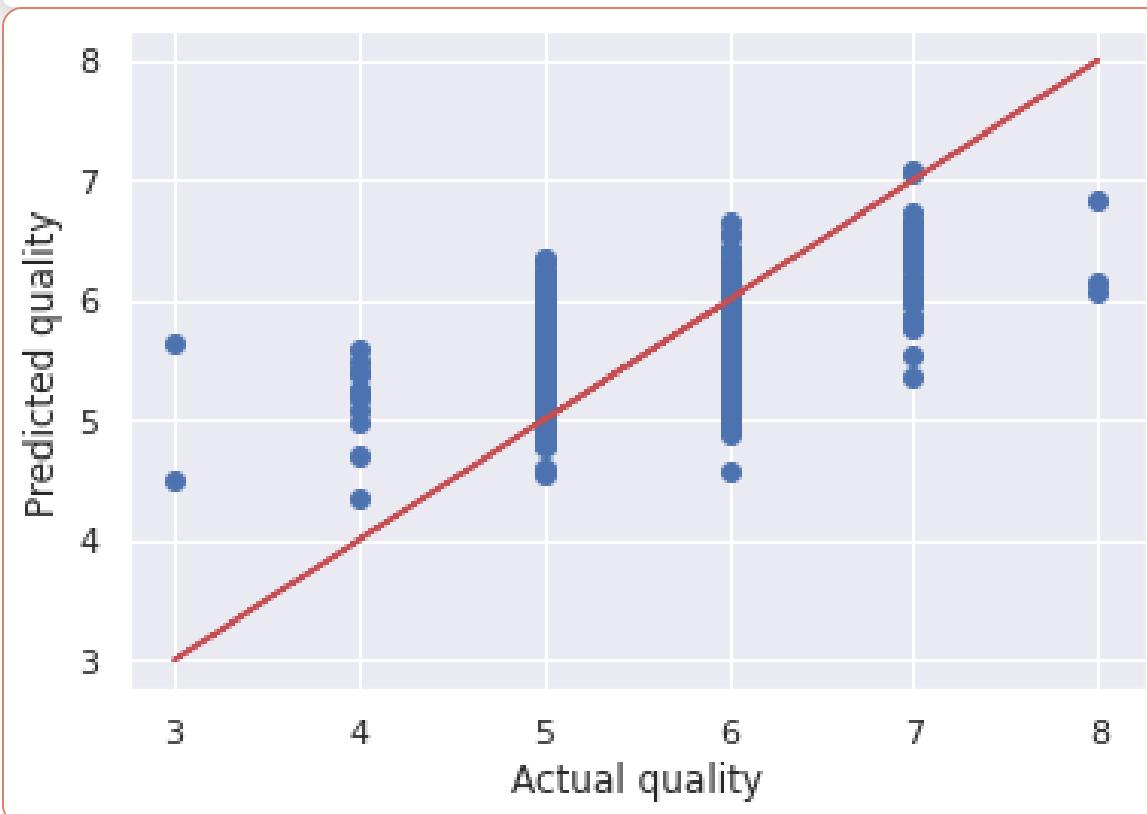
```
● ● ●  
1 # copy all features leaving aside the label.  
2 wine_features_test = strat_test_set.drop("quality", axis=1)  
3  
4 # copy the label list  
5 wine_labels_test = strat_test_set['quality'].copy()  
6  
7 # apply transformations  
8 wine_features_test_tr = transform_pipeline.fit_transform(wine_features_test)  
9  
10 # call predict function and calculate MSE.  
11 quality_test_predictions = lin_reg.predict(wine_features_test_tr)  
12 mean_squared_error(wine_labels_test, quality_test_predictions)
```

0.39759130875015164

Let's visualize the error between the actual and predicted values.



```
1 plt.scatter(wine_labels_test, quality_test_predictions)
2 plt.plot(wine_labels_test, wine_labels_test, 'r-')
3 plt.xlabel('Actual quality')
4 plt.ylabel('Predicted quality')
```



The model seem to be making errors on the best and poor quality wines.

Let's try another model: DecisionTreeRegressor.



```
1 from sklearn.tree import DecisionTreeRegressor  
2 tree_reg = DecisionTreeRegressor()  
3 tree_reg.fit(wine_features_tr, wine_labels)
```

```
DecisionTreeRegressor(ccp_alpha=0.0, criterion='mse', max_depth=None, max_features=None,  
max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None,  
min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0,  
presort='deprecated', random_state=None, splitter='best')
```

Notice similarity between two code snippets.

Linear regression	Decision Trees
lin_reg.fit(wine_features_tr, wine_labels)	tree_reg.fit(wine_features_tr, wine_labels)



```
1 quality_predictions = tree_reg.predict(wine_features_tr)
2 mean_squared_error(wine_labels, quality_predictions)
```

```
0.0
```



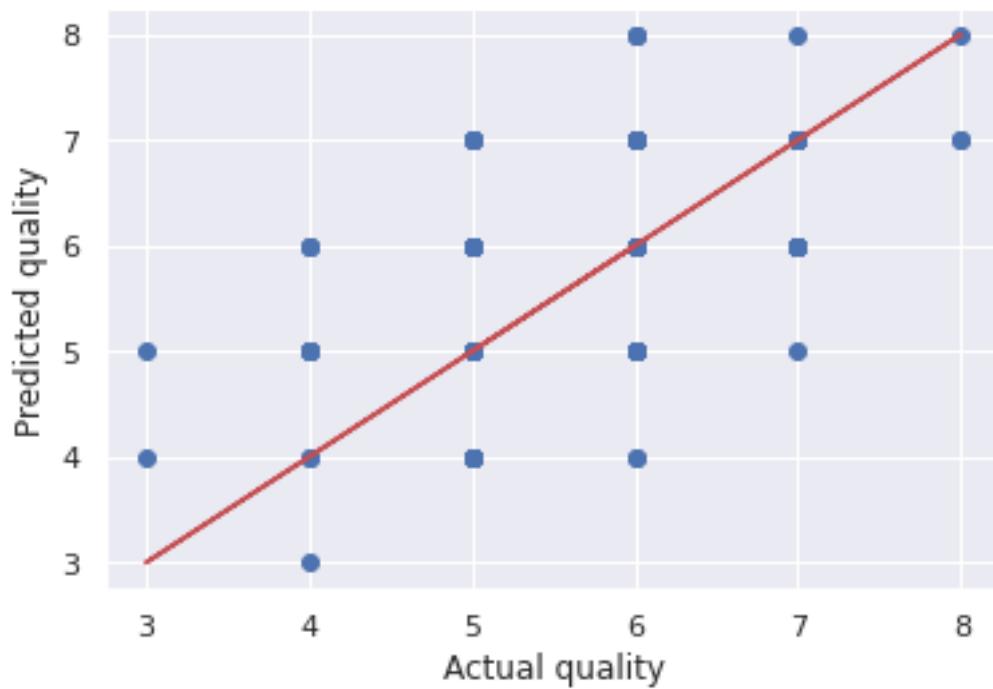
```
1 quality_test_predictions = tree_reg.predict(wine_features_test_tr)
2 mean_squared_error(wine_labels_test, quality_test_predictions)
```

```
0.58125
```

Note that the training error is 0, while the test error is 0.58. This is an example of an overfitted model.

● ● ●

```
1 plt.scatter(wine_labels_test, quality_test_predictions)
2 plt.plot(wine_labels_test, wine_labels_test, 'r-')
3 plt.xlabel('Actual quality')
4 plt.ylabel('Predicted quality')
```



We can use cross-validation (CV) for robust evaluation of model performance.



```
1 from sklearn.model_selection import cross_val_score
```

- Cross validation provides a separate MSE for each validation set, which we can use to get a mean estimation of MSE as well as the standard deviation, which helps us to determine how precise is the estimate.
- The additional cost we pay in cross validation is additional training runs, which may be too expensive in certain cases.



```
1 def display_scores(scores):
2     print("Scores:", scores)
3     print("Mean:", scores.mean())
4     print("Standard deviation:", scores.std())
```

Linear Regression CV



```
1 scores = cross_val_score(lin_reg, wine_features_tr, wine_labels,  
2                           scoring="neg_mean_squared_error", cv=10)  
3 lin_reg_mse_scores = -scores  
4 display_scores(lin_reg_mse_scores)
```

```
Scores: [ 0.56364537  0.4429824  0.38302744  0.40166681  0.29687635  0.37322622  
0.33184855  0.50182048  0.51661311  0.50468542]
```

```
Mean: 0.431639217212196
```

```
Standard deviation: 0.08356359730413976
```

Decision tree CV

```
● ● ●  
1 scores = cross_val_score(tree_reg, wine_features_tr, wine_labels,  
2                           scoring="neg_mean_squared_error", cv=10)  
3 tree_mse_scores = -scores  
4 display_scores(tree_mse_scores)  
  
Scores: [ 0.6171875  0.6875  0.6328125  0.5078125  0.4609375  0.640625  0.65625  0.7109375  
0.859375  1.07874016]  
Mean: 0.6852177657480315  
Standard deviation: 0.16668343331737054
```

Let's compare scores of Linear regression (LinReg) and decision tree (DT) regressions:

- LinReg has better MSE and more precise estimation compared to DT.

Random forest CV

- Random forest model builds multiple decision trees on randomly selected features and then average their predictions.
- Building a model on top of other model is called *ensemble learning*, which is often used to improve performance of ML models.



```
1 from sklearn.ensemble import RandomForestRegressor  
2  
3 forest_reg = RandomForestRegressor()  
4 forest_reg.fit(wine_features_tr, wine_labels)  
5  
6 scores = cross_val_score(forest_reg, wine_features_tr, wine_labels,  
7                           scoring="neg_mean_squared_error", cv=10)  
8 forest_mse_scores = -scores  
9 display_scores(forest_mse_scores)
```

```
Scores: [0.36989922 0.41363672 0.29063438 0.31722344 0.21798125 0.30233828 0.27124922  
0.38747344 0.42379219 0.46229449]
```

```
Mean: 0.34565226131889765
```

```
Standard deviation: 0.0736322184302973
```

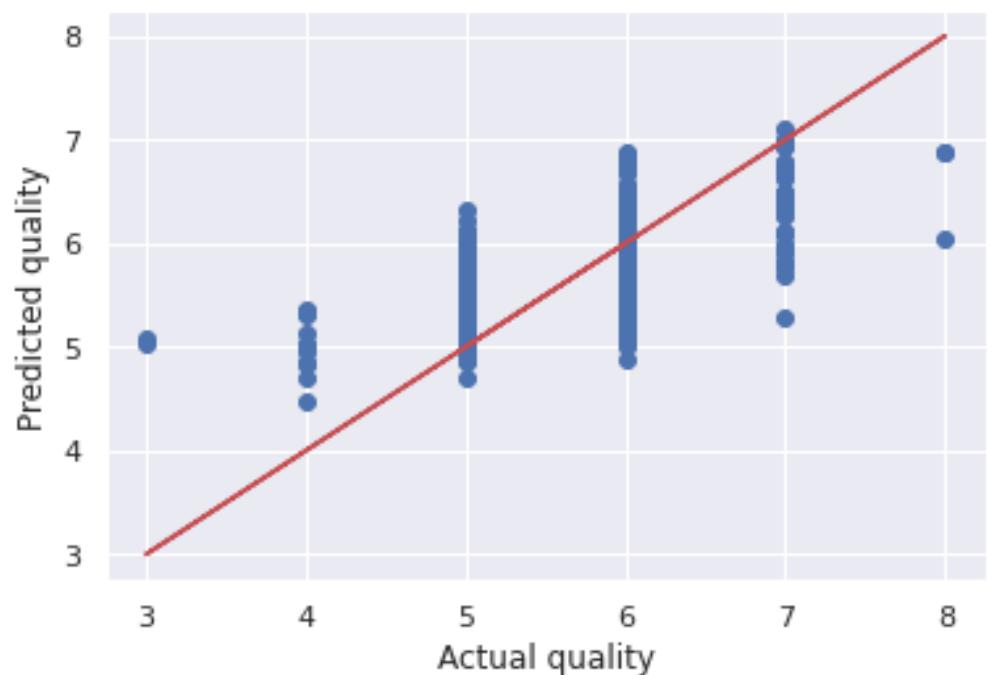


```
1 quality_test_predictions = forest_reg.predict(wine_features_test_tr)
2 mean_squared_error(wine_labels_test, quality_test_predictions)
```

```
0.34449875
```



```
1 plt.scatter(wine_labels_test, quality_test_predictions)
2 plt.plot(wine_labels_test, wine_labels_test, 'r-')
3 plt.xlabel('Actual quality')
4 plt.ylabel('Predicted quality')
```



Random forest looks more promising than the other two models.

- It's a good practice to build a few such models quickly without tuning their hyperparameters and shortlist a few promising models among them.
- Also save the models to the disk in Python **pickle** format.

What to do next?

Model diagnosis	Remedy
Underfitting	Models with more capacity
	Less constraints/regularization
Overfitting	More data
	Simpler model
	More constraints/regularization

Step 6: Finetune your model

- Usually there are a number of hyperparameters in the model, which are set manually.
- Tuning these hyperparameters lead to better accuracy of ML models.
- Finding the best combination of hyperparameters is a search problem in the space of hyperparameters, which is huge.

Grid search

- Scikit-Learn provides a class GridSearchCV that helps us in this pursuit.



```
1 from sklearn.model_selection import GridSearchCV
```

- We need to specify a list of hyperparameters along with the range of values to try.
- It automatically evaluates all possible combinations of hyperparameter values using cross-validation.

For example, there are number of hyperparameters in RandomForest regression such as:

- Number of estimators
- Maximum number of features

```
● ● ●  
1 param_grid = [  
2     {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},  
3     {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},  
4 ]
```

Here the parameter grid contains two combinations:

1. The first combination contains `n_estimators` with 3 values and `max_features` with 4 values.
2. The second combination has an additional `bootstrap` parameter, which is set to False. Note that it was set to its default value, which is `True`, in the first grid.

Let's compute the total combinations evaluated here:

1. The first one results in $3 \times 4 = 12$ combinations.
2. The second one has 2 values of `n_estimators` and 3 values of `max_features`, thus resulting $2 \times 3 = 6$ in total of values.

The total number of combinations evaluated by the parameter grid $12 + 6 = 18$

Let's create an object of `GridSearchCV`:

```
● ● ●  
1 grid_search = GridSearchCV(forest_reg, param_grid, cv=5,  
2                             scoring='neg_mean_squared_error',  
3                             return_train_score=True)
```

Let's create an object of **GridSearchCV**:

```
● ● ●  
1 grid_search = GridSearchCV(forest_reg, param_grid, cv=5,  
2                               scoring='neg_mean_squared_error',  
3                               return_train_score=True)
```

- In this case, we set **cv=5** i.e. using 5 fold cross validation for training the model.
- We need to train the model for 18 parameter combinations and each combination would be trained 5 times as we are using cross-validation here.
- The total model training runs = $18 \times 5 = 90$

Let's launch the hyperparameter search:



```
1 grid_search.fit(wine_features_tr, wine_labels)

GridSearchCV(cv=5, error_score=nan,
             estimator=RandomForestRegressor(bootstrap=True, ccp_alpha=0.0,
                                              criterion='mse', max_depth=None,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators=100, n_jobs=None,
                                              oob_score=False, random_state=None,
                                              verbose=0, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'max_features': [2, 4, 6, 8],
                          'n_estimators': [3, 10, 30]},
                         {'bootstrap': [False], 'max_features': [2, 3, 4],
                          'n_estimators': [3, 10]}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='neg_mean_squared_error', verbose=0)
```

The best parameter combination can be obtained as follows:

```
grid_search.best_params_
{'max_features': 6, 'n_estimators': 30}
```

Let's find out the error at different parameter settings:

```
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(-mean_score, params)
```



```
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(-mean_score, params)
```

```
0.5096674155773421 {'max_features': 2, 'n_estimators': 3}
0.38494794730392157 {'max_features': 2, 'n_estimators': 10}
0.35890284926470584 {'max_features': 2, 'n_estimators': 30}
0.4765907543572984 {'max_features': 4, 'n_estimators': 3}
0.37949047181372547 {'max_features': 4, 'n_estimators': 10}
0.3677285709422658 {'max_features': 4, 'n_estimators': 30}
0.47674223856209147 {'max_features': 6, 'n_estimators': 3}
0.39086173406862745 {'max_features': 6, 'n_estimators': 10}
0.35285364923747276 {'max_features': 6, 'n_estimators': 30}
0.47786049836601296 {'max_features': 8, 'n_estimators': 3}
0.37944690563725486 {'max_features': 8, 'n_estimators': 10}
0.35524742306644874 {'max_features': 8, 'n_estimators': 30}
0.4390253948801742 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
0.3897452818627451 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
0.4490985838779956 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
0.3858988664215686 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
0.45253914760348585 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
0.3858853860294117 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

As you can notice the lowest MSE is obtained for the best parameter combination.

Let's obtain the best estimator as follows:



```
1 grid_search.best_estimator_
```

```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features=6, max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=30, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False)
```

Note: `GridSearchCV` is initialized with `refit=True` option, which retrains the best estimator on the full training set. This is likely to lead us to a better model as it is trained on a larger dataset.

Randomized Search

- When we have a large hyperparameter space, it is desirable to try RandomizedSearchCV.
- It selects a random value for each hyperparameter at the start of each iteration and repeats the process for the given number of random combinations.
- It enables us to search hyperparameter space with appropriate budget control.



```
1 from sklearn.model_selection import RandomizedSearchCV
```

Analysis of best model and its errors

Analysis of the model provides useful insights about features. let's obtain the feature importance as learnt by the model:



```
1 feature_importances = grid_search.best_estimator_.feature_importances_
```



```
1 sorted(zip(feature_importances, feature_list), reverse=True)
[(0.2486711653610271, 'alcohol'),
 (0.14163642739406354, 'sulphates'),
 (0.12665569639367016, 'volatile acidity'),
 (0.08045272518319231, 'total sulfur dioxide'),
 (0.07275072016325315, 'density'),
 (0.05822554296729619, 'citric acid'),
 (0.05791188978825248, 'chlorides'),
 (0.057124416693656116, 'pH'),
 (0.056416454671447944, 'residual sugar'),
 (0.05388861091468478, 'fixed acidity'),
 (0.04626635046945642, 'free sulfur dioxide')]
```

- Based on this information, we may drop features that are not so important.
- It is also useful to analyze the errors in prediction and understand its causes and fix them.

Evaluation on test set

Now that we have a reasonable model, we evaluate its performance on the test set. The following steps are involved in the process:

1. Transform the test features.



```
1 # copy all features leaving aside the label.  
2 wine_features_test = strat_test_set.drop("quality", axis=1)  
3  
4 # copy the label list  
5 wine_labels_test = strat_test_set['quality'].copy()  
6  
7 # apply transformations  
8 wine_features_test_tr = transform_pipeline.fit_transform(wine_features_test)
```

2. Use the predict method with the trained model and the test set.



```
1 quality_test_predictions = grid_search.best_estimator_.predict(  
2     wine_features_test_tr)
```

3. Compare the predicted labels with the actual ones and report the evaluation metrics.



```
1 mean_squared_error(wine_labels_test, quality_test_predictions)
```

```
0.3534513888888883
```

4. It's a good idea to get 95% confidence interval of the evaluation metric. It can be obtained by the following code:



```
1 from scipy import stats
2 confidence = 0.95
3 squared_errors = (quality_test_predictions - wine_labels_test) ** 2
4 stats.t.interval(confidence, len(squared_errors) - 1,
5                   loc=squared_errors.mean(),
6                   scale=stats.sem(squared_errors))
```

```
(0.29159276569581916, 0.4153100120819586)
```

Step 7: Present your solution

Once we have satisfactory model based on its performance on the test set, we reach the prelaunch phase.

Before launch,

1. We need to present our solution that highlights learnings, assumptions and systems limitation.
2. Document everything, create clear visualizations and present the model.
3. In case, the model does not work better than the experts, it may still be a good idea to launch it and free up bandwidths of human experts.

Step 8: Launch, monitor and maintain your system

Launch

- Plug in input sources and
- Write test cases

Monitoring

- System outages
- Degradation of model performance
- Sampling predictions for human evaluation
- Regular assessment of data quality, which is critical for model performance

Maintenance

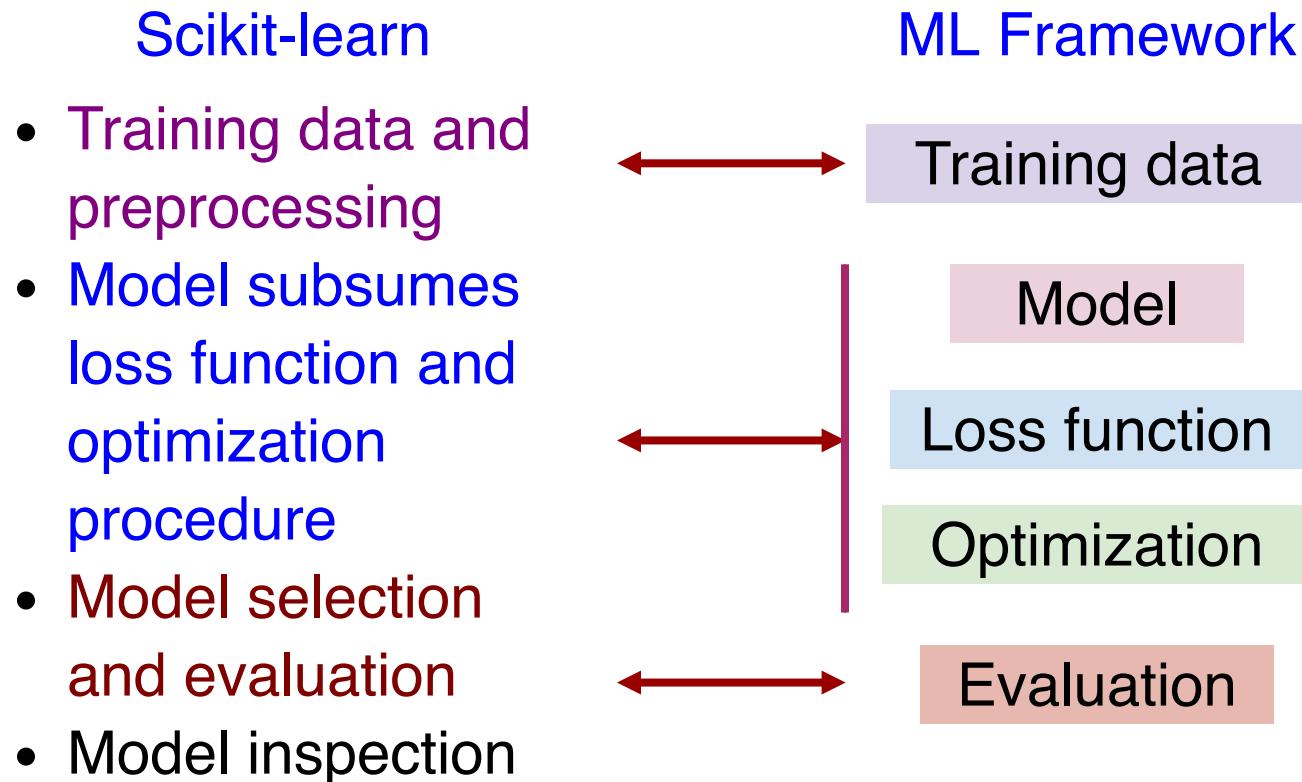
- Train model regularly every fixed interval with fresh data.
- Production roll out of the model.

Summary

In this module, we studied steps involved in end to end machine learning project with an example of prediction of wine quality.

Introduction to Scikit-Learn (sklearn)

sklearn APIs are organized on the lines of our ML framework.



API design principles

sklearn APIs are well designed with the following principles:

- **Consistency**: All APIs share a **simple and consistent** interface.
- **Inspection**: The **learnable parameters** as well as **hyperparameters** of all estimator's are **accessible directly** via public instance variables.
- **Nonproliferation of classes**: Datasets are represented as **Numpy arrays** or **Scipy sparse matrix** instead of custom designed classes.
- **Composition**: Existing building blocks are reduced as much as possible.
- **Sensible defaults** values are used for parameters that enables quick baseline building.

Types of sklearn objects

Transformers

- transforms dataset
- `transform()` for transforming dataset.
- `fit()` learns parameters.
- `fit_transform()` fits parameters and `transform()` the dataset.

Estimators

- Estimates model parameters based on training data and hyper parameters.
- `fit()` method

Predictors

- Makes prediction on dataset
- `predict()` method that takes dataset as an input and returns predictions.
- `score()` method to measure quality of predictions.

Data Preprocessing



Training



Inference

sklearn API

@sir, copied to 'Data
Preprocessing' slide deck

Data API

Provides functionality for **loading**, **generating** and **preprocessing** the training and test data.

Module	Functionality
<code>sklearn.datasets</code>	Loading datasets - custom as well as popular reference dataset.
<code>sklearn.preprocessing</code>	Scaling, centering, normalization and binarization methods
<code>sklearn.impute</code>	Filling missing values
<code>sklearn.feature_selection</code>	Implements feature selection algorithms
<code>sklearn.feature_extraction</code>	Implements feature extraction from raw data.

Model API

Implements **supervised** and **unsupervised** models

Regression

- `sklearn.linear_model`
(linear, ridge, lasso
models)
- `sklearn.trees`

Classification

- `sklearn.linear_model`
- `sklearn.svm`
- `sklearn.trees`
- `sklearn.neighbors`
- `sklearn.naive_bayes`
- `sklearn.multiclass`

`sklearn.multioutput` implements multi-output
classification and regression.

`sklearn.cluster` implements many popular clustering
algorithms

Model evaluation API

`sklearn.metrics` implements different metrics for model evaluation.

- Classification metrics
- Regression metrics
- Clustering metrics

Model selection API

`sklearn.model_selection` implements various model selection strategies like cross-validation, tuning hyperparameters and plotting learning curves.

Model inspection API

`sklearn.model_inspection` includes tools for model inspection.

Practical advice

- It is not possible to remember each and every sklearn API.
- Remember high level modules and API design principles.
- Use documentation for more information as follows:

```
1 import sklearn.linear_model import LogisticRegression  
2 ?LogisticRegression
```

- Keep the following links handy:
 - [API reference](#)
 - [sklearn user guide](#)
 - [Worked examples](#) for reference implementations

Data loading

General dataset API has **three** main kind of interfaces:

- The dataset **loaders** are used to **load** toy datasets bundled with sklearn.
- The dataset **fetchers** are used to **download** and load datasets from the internet.
- The dataset **generators** are used to **generate** controlled synthetic datasets.

Dataset API

Loaders

Load small standard datasets

Fetchers

Fetch and load larger datasets

Generator

Controlled synthetic datasets

Both loaders and fetchers return a `Bunch` object, which is a dictionary with two keys of our interest:

Key	Values
data	Array of shape (n, m)
target	Array of shape (n,)

`load_*`

`fetch_*`

`make_*`

`return_X_y = True`

Returns tuple (`X`, `y`) of numpy arrays:

- `X` has shape (n, m)
- `y` has shape $(n,)$

Dataset Loaders

Dataset Loader	# samples (n)	# features (m)	# labels	Type
<code>load_iris</code>	150	3	1	Classification
<code>load_diabetes</code>	442	10	1	Regression
<code>load_digits</code>	1797	64	1	Classification
<code>load_linnerud</code>	20	3	3	Regression (multi output)
<code>load_wine</code>	178	13	1	Classification
<code>load_breast_cancer</code>	569	30	1	Classification

Note: These datasets are bundled with sklearn and we do not require to download them from external sources.

Dataset Fetchers

Dataset Loader	# samples (n)	# features (m)	# labels	Type
<code>fetch_olivetti_faces</code>	400	4096	1 (40)	multi-class image classification
<code>fetch_20newsgroups</code>	18846	1	1 (20)	(multi-class) text classification
<code>fetch_lfw_people</code>	13233	5828	1 (5749)	(multi-class) image classification
<code>fetch_covtype</code>	581012	54	1 (7)	(multi-class) classification
<code>fetch_rcv1</code>	804414	47236	1 (103)	(multi-class) classification
<code>fetch_kddcup99</code>	4898431	41	1	(multi-class) classification
<code>fetch_california_housing</code>	20640	8	1	regression

Dataset generators

Regression

`make_regression()` produces regression targets as a sparse random linear combination of random features with noise. The informative features are either uncorrelated or low rank.

Classification

Single label

`make_blobs()` and `make_classification()` first creates a bunch of normally-distributed clusters of points and then assign one or more clusters to each class thereby creating multi-class datasets.

Multilabel

`make_multilabel_classification()` generates random samples with multiple labels with a specific generative process and rejection sampling.

Dataset generators

Clustering

`make_blobs()` generates a **bunch** of **normally-distributed** clusters of points with specific mean and standard deviations for each cluster.

Loading external datasets

`fetch_openml()` fetches datasets from [openml.org](#), which is a public repository for machine learning data and experiments.

`pandas.io` provides tools to read from [common formats](#) like CSV, excel, json, SQL.

`scipy.io` specializes in [binary formats](#) used in scientific computing like .mat and .arff.

`numpy/routines.io` specializes in loading [columnar data](#) into numpy arrays.

`dataset.load_files` loads directories of [text files](#) where directory name is a label and each file is a sample.

Loading external datasets

`datasets.load_svmlight_files()` loads data in [svmlight](#) and [libSVM](#) sparse format.

`skimage.io` provides tools to load [images](#) and [videos](#) in numpy arrays.

`scipy.io.wavfile.read` specializes [reading](#) WAV file into a numpy array.

For managing numerical data, sklearn recommends using an optimized file format such as [HDF5 \(Hierarchical Data Format version 5\)](#) to reduce data load times.

Pandas, Py Tables and H5Py provides an interface to read and write data in that format.

Data transformation

Types of transformers

sklearn provides a library of transformers for

- Data cleaning ([sklearn.preprocessing](#)) such as
- Feature extraction ([sklearn.feature_extraction](#))
- Feature reduction
- Feature expansion ([sklearn.kernel_approximation](#))

Transformer methods

Each transformer has the following methods:

- `fit()` method learns model parameters from a training set.
- `transform()` method applies the learnt transformation to the new data.
- `fit_transform()` performs function of both `fit()` and `transform()` methods and is more convenient and efficient to use.

Transformers are combined with one another or with other estimators such as classifiers or regressors to build composite estimators.

Tool	Usage
Pipeline	Chaining multiple estimators to execute a fixed sequence of steps in data preprocessing and modelling.
FeatureUnion	Combines output from several transformer objects by creating a new transformer from them.
ColumnTransformer	Enables different transformations on different columns of data based on their types.

Data Preprocessing

Machine Learning Practice

Dr. Ashish Tendulkar

IIT Madras

The real world training data is usually not clean and has many issues such as **missing values** for certain features, features on **different scales**, **non-numeric attributes** etc.

Often there is a need to **pre-process** the data to make it amenable for training the model.

Sklearn provides a rich set of transformers for this job.

The **same pre-processing** should be applied to both training and test set.

Sklearn provides **pipeline** for making it easier to chain multiple transforms together and apply them **uniformly** across train, eval and test sets.

Once you get the training data, the first job is to **explore** the **data** and list down **preprocessing** needed.

Typical problems include

Missing values in features

Numerical features are **not on the same scale**.

Categorical attributes need to be represented with sensible numerical representation.

Too many features, reduce them.

Extract features from non-numeric data.

Sklearn provides a **library of transformers** for
data preprocessing.

- Data cleaning (`sklearn.preprocessing`) such as standardization, missing value imputation, etc.
- Feature extraction (`sklearn.feature_extraction`)
- Feature reduction (`sklearn.decomposition.pca`)
- Feature expansion (`sklearn.kernel_approximation`)

Transformer methods

Each transformer has the following methods:

- `fit()` method learns model parameters from a training set.
- `transform()` method applies the learnt transformation to the new data.
- `fit_transform()` performs function of both `fit()` and `transform()` methods and is more convenient and efficient to use.

Part 1. Feature extraction

`sklearn.feature_extraction` has useful APIs to extract features from data:

DictVectorizer

FeatureHasher

Let's study these APIs one by one.

DictVectorizer

Converts lists of mappings of feature name and feature value, into a matrix.

Original data

```
data =  
[{'age': 4, 'height':96.0},  
{'age': 1, 'height':73.9},  
{'age': 3, 'height':88.9},  
{'age': 2, 'height':81.6}]
```

Transformed feature matrix \mathbf{X}'

$$\mathbf{X}'_{4 \times 2} = \begin{bmatrix} 4 & 96.0 \\ 1 & 73.9 \\ 3 & 88.9 \\ 2 & 81.6 \end{bmatrix}$$

```
dv = DictVectorizer(sparse=False)  
dv.fit_transform(data)
```

FeatureHasher

- High-speed, low-memory vectorizer that uses feature hashing technique.
- Instead of building a hash table of the features, as the vectorizers do, it applies a hash function to the features to determine their column index in sample matrices directly.
- This results in increased speed and reduced memory usage, at the expense of inspectability; the hasher does not remember what the input features looked like and has no inverse_transform method.
- Output of this transformer is `scipy.sparse` matrix.

Feature Extraction from images and text

- `sklearn.feature_extraction.image.*` has useful APIs to extract features from image data. Find out more about them in sklearn user guide at the following link: [Feature Extraction from Images](#).
- `sklearn.feature_extraction.text.*` has useful APIs to extract features from text data. Find out more about them in sklearn user guide at the following link: [Feature Extraction from Text](#).

Part 2: Data Cleaning

Handling missing values

Missing values occur due to errors in data capture such as sensor malfunctioning, measurement errors etc.

Many ML algorithms do not work with missing data and need all features to be present.

Discarding records containing missing values would result in loss of valuable training samples.

`sklearn.impute` API provides functionality to fill missing values in a dataset.

`SimpleImputer`

`KNNImputer`

`MissingIndicator` provides indicators for missing values.

SimpleImputer

- Fills missing values with one of the following strategies:
`'mean'`, `'median'`, `'most_frequent'` and `'constant'`.

Original feature matrix \mathbf{X}

$$\mathbf{X}_{4 \times 2} = \begin{bmatrix} 7 & 1 \\ \textcolor{red}{nan} & 8 \\ 2 & \textcolor{red}{nan} \\ 9 & 6 \end{bmatrix}$$

```
si = SimpleImputer(strategy='mean')  
si.fit_transform(X)
```

Transformed feature matrix \mathbf{X}'

$$\mathbf{X}'_{4 \times 2} = \begin{bmatrix} 7 & 1 \\ 6 & 8 \\ 2 & 5 \\ 9 & 6 \end{bmatrix}$$

$$\frac{7 + 2 + 9}{3} = 6$$
$$\frac{1 + 8 + 6}{3} = 5$$

KNNImputer

- Uses **k-nearest neighbours** approach to fill missing values in a dataset.
 - The missing value of an attribute in a specific example is filled with the **mean** value of the same attribute of **n_neighbors** **closest neighbors**.
- The nearest neighbours are decided based on **Euclidean distance**.

Example: KNNImputer

- Consider following feature matrix.

$$\mathbf{X}_{4 \times 3} = \begin{bmatrix} 1. & 2. & \text{nan} \\ 3. & 4. & 3. \\ \text{nan} & 6. & 5. \\ 8. & 8. & 7. \end{bmatrix}$$

- It has 4 samples and 2 missing values.
- Let's fill in missing values with KNNImputer.

Let's fill the missing value in first sample/row.

$$\mathbf{X}_{4 \times 3} = \begin{bmatrix} 1. & 2. & \text{nan} \\ 3. & 4. & 3. \\ \text{nan} & 6. & 5. \\ 8. & 8. & 7. \end{bmatrix}$$

Distance with [1. 2. nan.]

$$\begin{bmatrix} 3. & 4. & 3. \\ \text{nan} & 6. & 5. \\ 8. & 8. & 7. \end{bmatrix}$$

$$\sqrt{(1 - 3)^2 + (2 - 4)^2} \approx 2.82$$

$$\sqrt{(2 - 6)^2} = 4$$

$$\sqrt{(1 - 8)^2 + (2 - 8)^2} \approx 9.21$$

2 nearest
neighbours

Values of the feature from
2 nearest neighbours



$$\frac{3 + 5}{2} = 4$$



$$[1. \quad 2. \quad 4.]$$

of neighbours

In this way, we can fill up the missing values with `KNNImputer`.

Original feature
matrix \mathbf{X}

$$\mathbf{X}_{4 \times 4} = \begin{bmatrix} 1. & 2. & \text{nan.} \\ 3. & 4. & 3. \\ \text{nan} & 6. & 5. \\ 8. & 8. & 7. \end{bmatrix}$$

Transformed feature
matrix \mathbf{X}'

$$\mathbf{X}'_{4 \times 4} = \begin{bmatrix} 1. & 2. & 4. \\ 3. & 4. & 3. \\ 5.5 & 6. & 5. \\ 8. & 8. & 7. \end{bmatrix}$$

```
knni = KNNImputer(n_neighbors=2, weights="uniform")
knni.fit_transform(X)
```

Marking imputed values

- It is useful to indicate the presence of missing values in the dataset.
- `MissingIndicator` helps us get those indications.
 - It returns a binary matrix,
 - True values correspond to missing entries in original dataset.

1.2 Numeric transformers

1. Feature scaling
2. Polynomial transformation
3. Discretization

Feature scaling

Numerical features with different scales leads to slower convergence of iterative optimization procedures.

It is a good practice to scale numerical features so that all of them are on the same scale.

Let's learn how to scale numerical features with sklearn API.

Three feature scaling APIs are available in sklearn

StandardScaler

MaxAbsScaler

MinMaxScaler

StandardScaler

Transforms the original features vector \mathbf{x} into new feature vector \mathbf{x}' using following formula

$$\mathbf{x}' = \frac{\mathbf{x} - \mu}{\sigma}$$

Learns parameters μ and σ .

$$\mathbf{x}_{5 \times 1} = \begin{bmatrix} 4 \\ 3 \\ 2 \\ 5 \\ 6 \end{bmatrix}$$

```
ss = StandardScaler()  
ss.fit_transform(x)
```

$$\mathbf{x}'_{5 \times 1} = \begin{bmatrix} 0 \\ -1/\sqrt{2} \\ -2/\sqrt{2} \\ 1/\sqrt{2} \\ 2/\sqrt{2} \end{bmatrix}$$

$$\mu = 4, \sigma = \sqrt{2}$$

$$\mu' = 0, \sigma' = 1$$

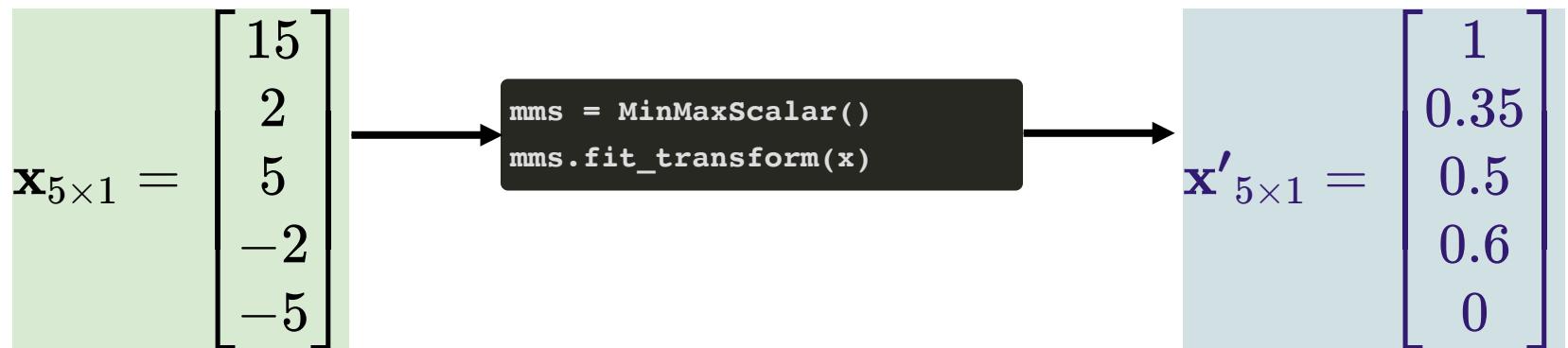
Note that the transformed feature vector \mathbf{x}' has mean (μ) = 0 and standard deviation (σ) = 1.

MinMaxScaler

It transforms the original feature vector \mathbf{x} into new feature vector \mathbf{x}' so that all values fall within range [0, 1]

$$\mathbf{x}' = \frac{\mathbf{x} - \mathbf{x}.min}{\mathbf{x}.max - \mathbf{x}.min}$$

where $\mathbf{x}.max$ and $\mathbf{x}.min$ are largest and smallest values of that feature respectively, of the original feature vector \mathbf{x} .



$\mathbf{x}.max = 15, \mathbf{x}.min = -5$

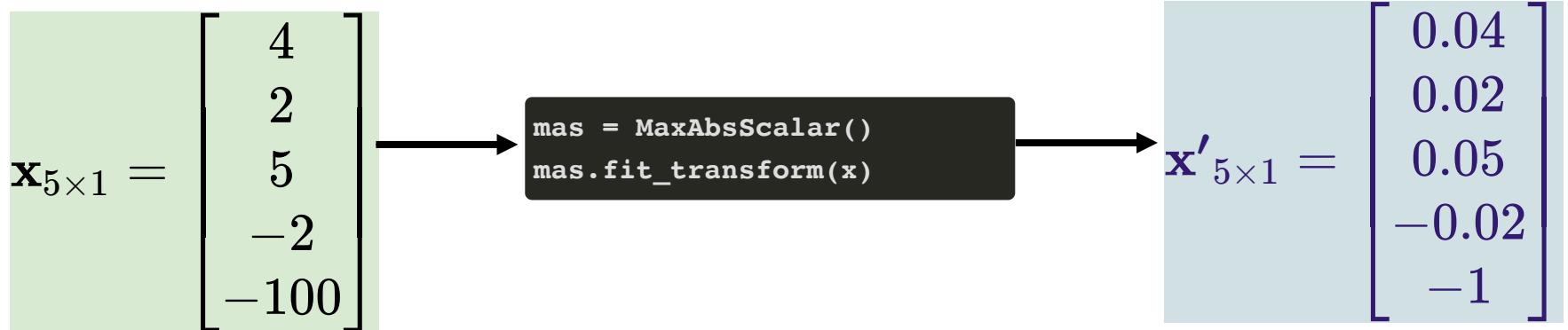
The largest number is transformed to 1 and the smallest number is transformed to 0.

MaxAbsScaler

It transforms the original features vector \mathbf{x} into new feature vector \mathbf{x}' so that **all values fall within range $[-1, 1]$**

$$\mathbf{x}' = \frac{\mathbf{x}}{\text{MaxAbsoluteValue}}$$

where $\text{MaxAbsoluteValue} = \max(\mathbf{x}.max, |\mathbf{x}.min|)$



$$\text{MaxAbsoluteValue} = \max(5, |-100|) = 100$$

FunctionTransformer

Constructs transformed features by applying a user defined function.

$$\mathbf{X}_{4 \times 2} = \begin{bmatrix} 128 & 2 \\ 2 & 256 \\ 4 & 1 \\ 512 & 64 \end{bmatrix}$$

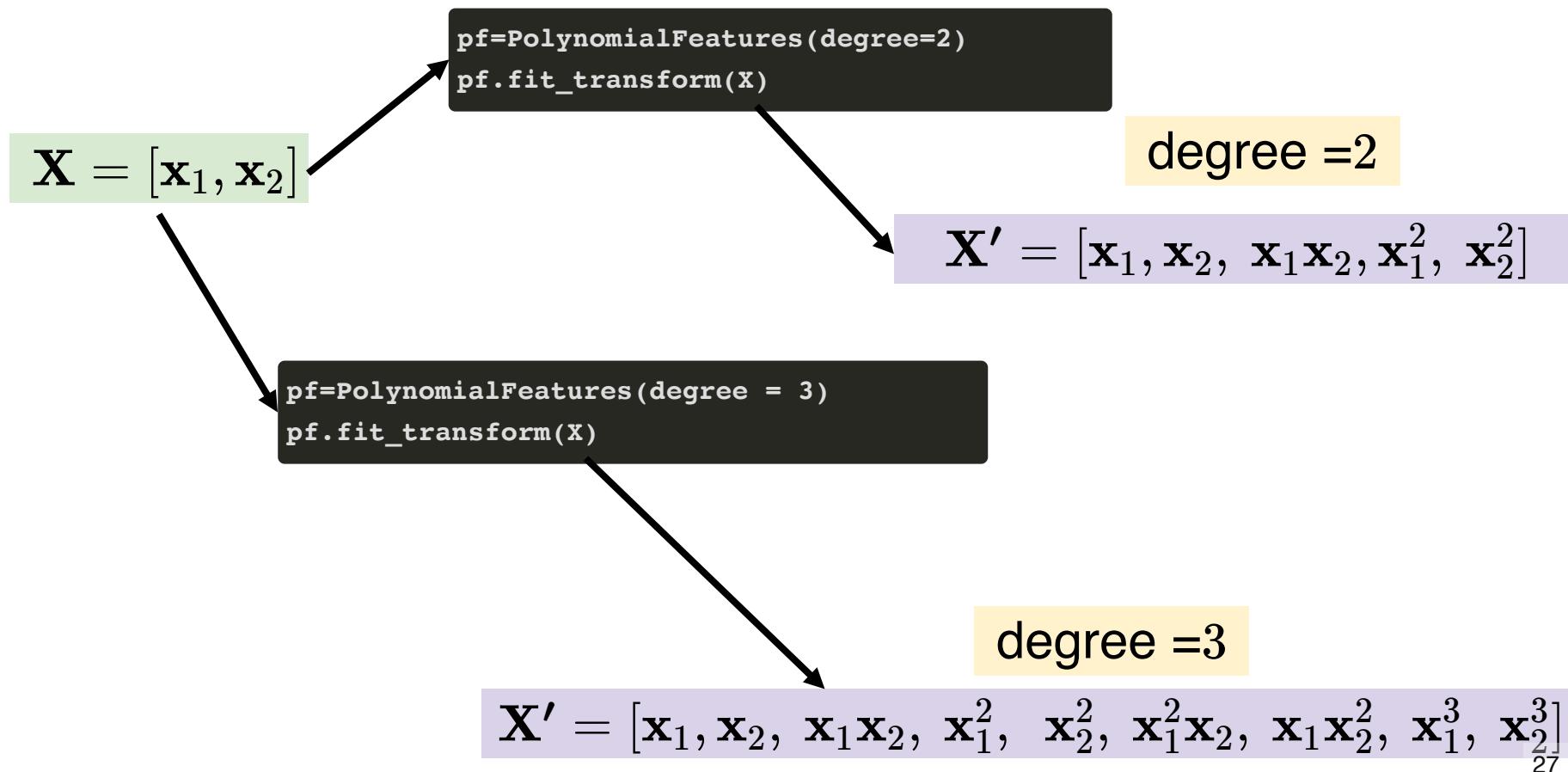
$$\mathbf{X'}_{4 \times 2} = \begin{bmatrix} 7 & 1 \\ 1 & 8 \\ 2 & 0 \\ 9 & 6 \end{bmatrix}$$

```
ft = FunctionTransformer(numpy.log2)
ft.fit_transform(X)
```

Applies \log_2 function to the features.

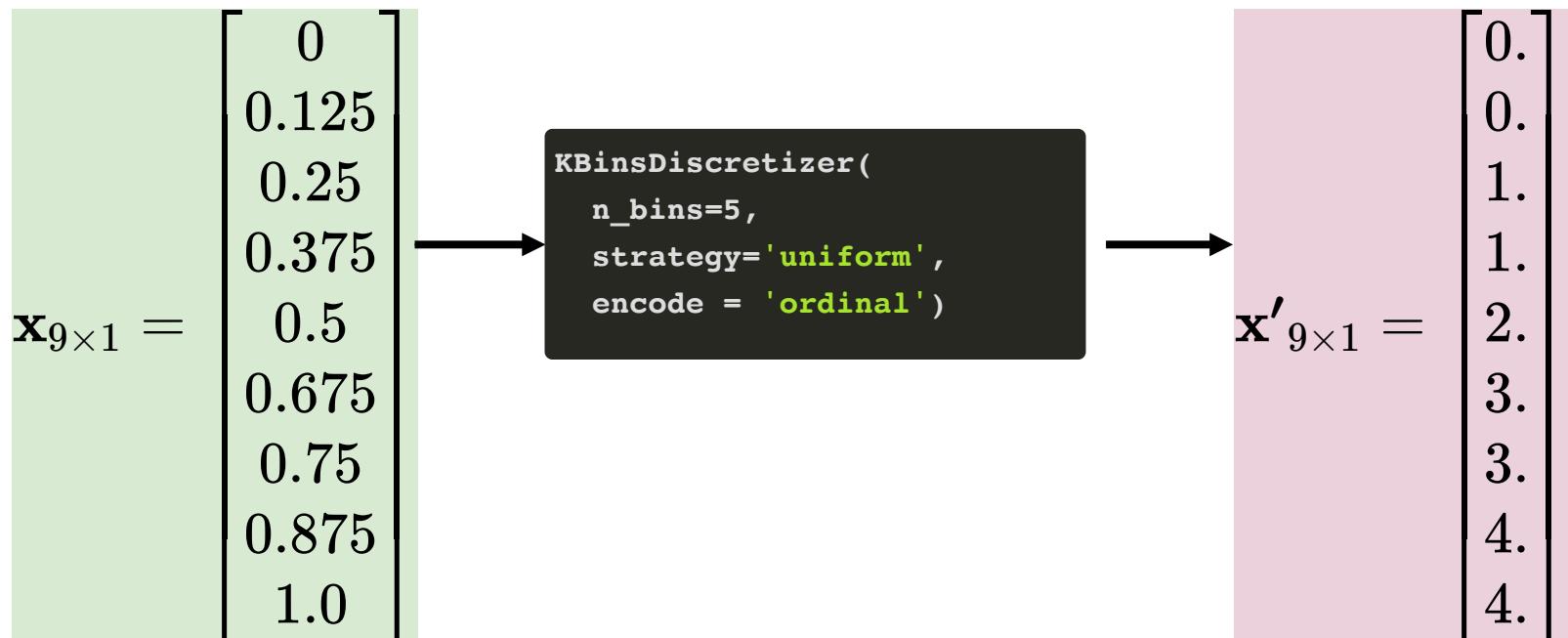
Polynomial transformation

Generates a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree.



KBinsDiscretizer

- Divides a **continuous variable** into bins.
- One hot encoding or ordinal encoding is further applied to the **bin labels**.



1.2 Categorical transformers

1. Feature encoding
2. Label encoding

OneHotEncoder

- Encodes **categorical feature** or **label** as a **one-hot numeric array**.
- Creates **one binary column** for each of K unique values.
- **Exactly one column has 1** in it and rest have 0.



unique values:
 $K = 3$

columns in
transformed matrix = 3

LabelEncoder

Encodes target labels with value between 0 and $K - 1$, where K is number of distinct values.



Here $K = 4$: $\{1, 2, 6, 8\}$

1 is encoded as 0, 2 as 1, 6 as 2, and 8 as 3.

OrdinalEncoder

Encodes categorical features with value between 0 and $K - 1$, where K is number of distinct values.



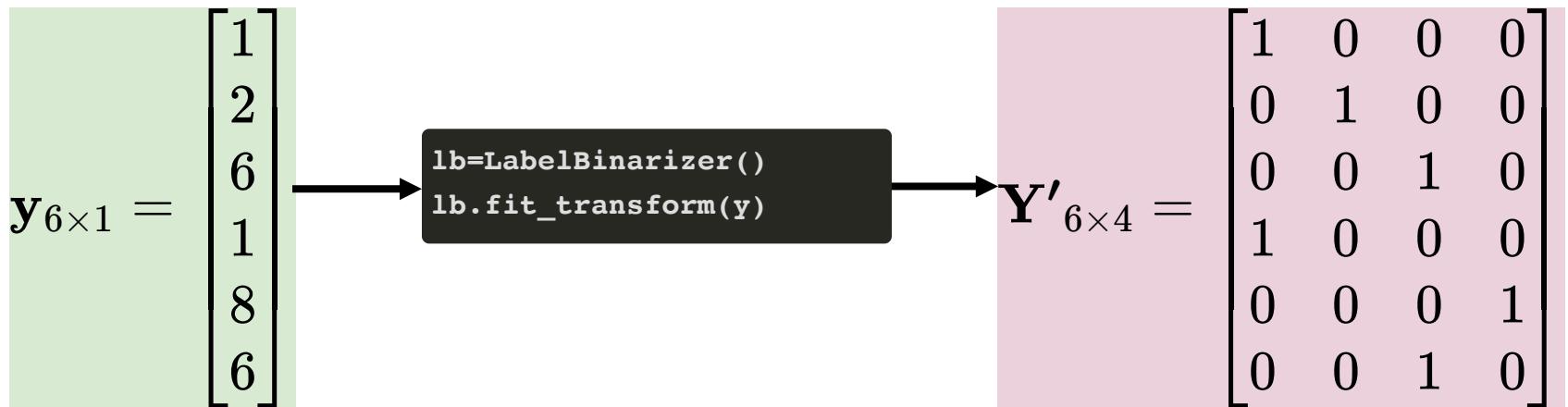
OrdinalEncoder can operate multi dimensional data, while LabelEncoder can transform only 1D data.

LabelBinarizer

Several regression and binary classification can be extended to multi-class setup in **one-vs-all** fashion.

This involves training a single regressor or classifier per class.

For this, we need to **convert multi-class labels to binary labels**, and **LabelBinarizer** performs this task.



If estimator **supports multiclass** data, **LabelBinarizer** is not needed.

MultiLabelBinarizer

Encodes **categorical features** with value between **0** and **$K - 1$** , where **K** is number of classes.

In this example $K = 4$, since there are only 4 genres of movies.

```
movie_genres =  
[{'action', 'comedy'},  
 {'comedy'},  
 {'action', 'thriller'},  
 {'science-fiction', 'action', 'thriller'}]
```

$$\mathbf{X}'_{4 \times 4} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

```
mlb = MultiLabelBinarizer()  
mlb.fit_transform(movie_genres)
```

add_dummy_feature

Augments dataset with a column vector, each value in the column vector is 1.

$$\mathbf{X}_{4 \times 2} = \begin{bmatrix} 7 & 1 \\ 1 & 8 \\ 2 & 0 \\ 9 & 6 \end{bmatrix} \xrightarrow{\text{add_dummy_feature}(\mathbf{x})} \mathbf{X'}_{4 \times 3} = \begin{bmatrix} 1 & 7 & 1 \\ 1 & 1 & 8 \\ 1 & 2 & 0 \\ 1 & 9 & 6 \end{bmatrix}$$

Part 2: Feature selection

- Filter based
- Wrapper based

- Sometimes in a real world dataset, **all features do not contribute well enough towards fitting a model.**
- The features that do not contribute significantly, can be removed. It leads to **decrease in size of the dataset** and hence, the **computation cost** of fitting a model.
- `sklearn.feature_selection` provides many APIs to accomplish this task.

Filter

`VarianceThreshold`
`SelectKBest`
`SelectPercentile`
`GenericUnivariateSelect`

Wrapper

`RFE`
`RFECV`
`SelectFromModel`
`SequentialFeatureSelector`

Note: Tree based and kernel based feature selection algorithms will be covered in later weeks.

Filter based feature selection methods

Removing features with low variance

VarianceThreshold

Removes all features with variance below a certain threshold, as specified by the user, from input feature matrix

By default removes a feature which has same value, i.e. zero variance.

Univariate feature selection

Univariate feature selection **selects** features based on **univariate statistical tests**.

There are three APIs for univariate feature selection:

`SelectKBest`

Removes **all but the k highest scoring features**

`SelectPercentile`

Removes **all but a user-specified $\text{highest scoring percentage}$ of features**

`GenericUnivariateSelect`

Performs univariate feature selection with a **configurable strategy**, which can be found via **hyper-parameter search**.

`sklearn` provides one more class of univariate feature selection methods that work on **common univariate statistical tests** for each feature:

`SelectFpr` selects features based on a false positive rate test.

`SelectFdr` selects features based on an estimated false discovery rate.

`SelectFwe` selects features based on family-wise error rate.

Univariate scoring function

- Each API need a **scoring function** to score each feature.
- **Three classes** of scoring functions are proposed:

Mutual information (MI)

Chi-square

F-statistics

- MI and F-statistics can be used in both **classification** and **regression** problems.

mutual_info_regression

f_regression

mutual_info_classif

f_classif

- Chi-square can be used only in **classification** problems.

chi2

Mutual information (MI)

- Measures dependency between two variables.
- It returns a non-negative value.
 - MI = 0 for independent variables.
 - Higher MI indicates higher dependency.

Chi-square

- Measures dependence between two variables.
- Computes chi-square stats between non-negative feature (boolean or frequencies) and class label.
- Higher chi-square values indicates that the features and labels are likely to be correlated.

MI and chi-squared feature selection is recommended for sparse data.

SelectKBest

```
skb = SelectKBest(chi2, k=20)
X_new = skb.fit_transform(X, y)
```

Selects 20 best features based on chi-square scoring function.

SelectPercentile

```
sp = SelectPercentile(chi2, percentile=20)
X_new = sp.fit_transform(X, y)
```

Selects top 20 percentile best features based on chi-square scoring function.

'percentile' (default), 'k_best',

'fpr', 'fdr', 'fwe'

GenericUnivariateSelect

```
transformer = GenericUnivariateSelect(chi2, mode='k_best', param=20)
X_new = transformer.fit_transform(X, y)
```

Selects 20 best features based on chi-square scoring function.

GenericUnivariateSelect

```
transformer = GenericUnivariateSelect(chi2, mode='k_best', param=20)
X_new = transformer.fit_transform(X, y)
```

- Selects set of features based on a feature selection mode and a scoring function.
- The `mode` could be `'percentile'` (default), `'k_best'`, `'fpr'`, `'fdr'`, `'fwe'`.
- The `param` argument takes value corresponding to the `mode`.

Do not use regression feature scoring function with a classification problem. It will lead to useless results.

Wrapper based filter selection

Unlike filter based methods, wrapper based methods use **estimator class** rather than a **scoring function**.

Recursive Feature Elimination (RFE)

- Uses an estimator to recursively remove features.
 - Initially fits an estimator on all features.
 - Obtains feature importance from the estimator and removes the least important feature.
 - Repeats the process by removing features one by one, until desired number of features are obtained.
-
- Use `RFECV` if we do not want to specify the desired number of features in `RFE`.
 - It performs `RFE` in a cross-validation loop to find the optimal number of features.

SelectFromModel

Selects desired number of important features (as specified with `max_features` parameter) above certain threshold of feature importance as obtained from the trained estimator.

- The feature importance is obtained via `coef_`,
`feature_importances_` or an `importance_getter` callable from the trained estimator
- The feature importance threshold can be specified either numerically or through string argument based on built-in heuristics such as `'mean'`, `'median'` and float multiples of these like `'0.1*mean'`.

Let's look at a concrete example of SelectFromModel

```
clf = LinearSVC(C=0.01, penalty="l1", dual=False)
clf = clf.fit(X, y)
clf.coef_

model = SelectFromModel(clf, prefit=True)
X_new = model.transform(X)
```

- Here we use a linear support vector classifier to get coefficients of features for `SelectFromModel` transformer.
- It ends up selecting features with non-zero weights or coefficients.

Sequential feature selection

Performs feature selection by selecting or deselecting features one by one in a greedy manner.

Uses one of the two approaches

Forward selection

Starting with a zero feature, it finds one feature that obtains the best cross validation score for an estimator when trained on that feature.

Repeats the process by adding a new feature to the set of selected features.

Backward selection

Starting with all features and removes least important features one by one following the idea of forward selection.

Stops when reach the desired number of features.

- The `direction` parameter **controls** whether **forward** or **backward SFS** is used.
- In general, forward and backward selection **do not yield equivalent results**.
- Select the direction that is **efficient** for the required number of selected features:
 - When we want to select 7 out of 10 features,
 - Forward selection would need to perform 7 iterations.
 - Backward selection would only need to perform 3.
 - Backward selection seems to be a reasonable choice here.

- SFS does not require the underlying model to expose a `coef_` or `feature_importances_` attributes unlike in `RFE` and `SelectFromModel`.
- SFS may be slower than `RFE` and `SelectFromModel` as it needs to evaluate more models compared to the other two approaches.

For example in backward selection, the iteration going from m features to $m - 1$ features using k -fold cross-validation requires fitting $m \times k$ models, while

- `RFE` would require only a single fit, and
- `SelectFromModel` performs a single fit and requires no iterations.

Applying transformations to
diverse features

Generally training data contains diverse features such as numeric and categorical.

Different feature types are processed with different transformers.

Need a way to combine different feature transformers seamlessly.

Composite Transformer

`sklearn.compose` has useful classes and methods to apply transformation on subset of features and combine them:

ColumnTransformer

TransformedTargetRegressor

ColumnTransformer

- It applies **a set of transformers** to columns of an array or `pandas.DataFrame`, concatenates the transformed outputs from different transformers into a **single matrix**.
- It is useful for **transforming heterogenous data** by applying **different transformers** to separate subsets of **features**.
- It combines different feature selection mechanisms and transformation into a single transformer object.

- **ColumnTransformer()**
- Each tuple has format
 - **(estimatorName, estimator(...), columnIndices)**

```
column_trans = ColumnTransformer(  
    [('ageScaler', CountVectorizer(), [0]),  
     ('genderEncoder', OneHotEncoder(dtype='int'), [1])],  
    remainder='drop', verbose_feature_names_out=False)
```

Illustration of Column Transformer

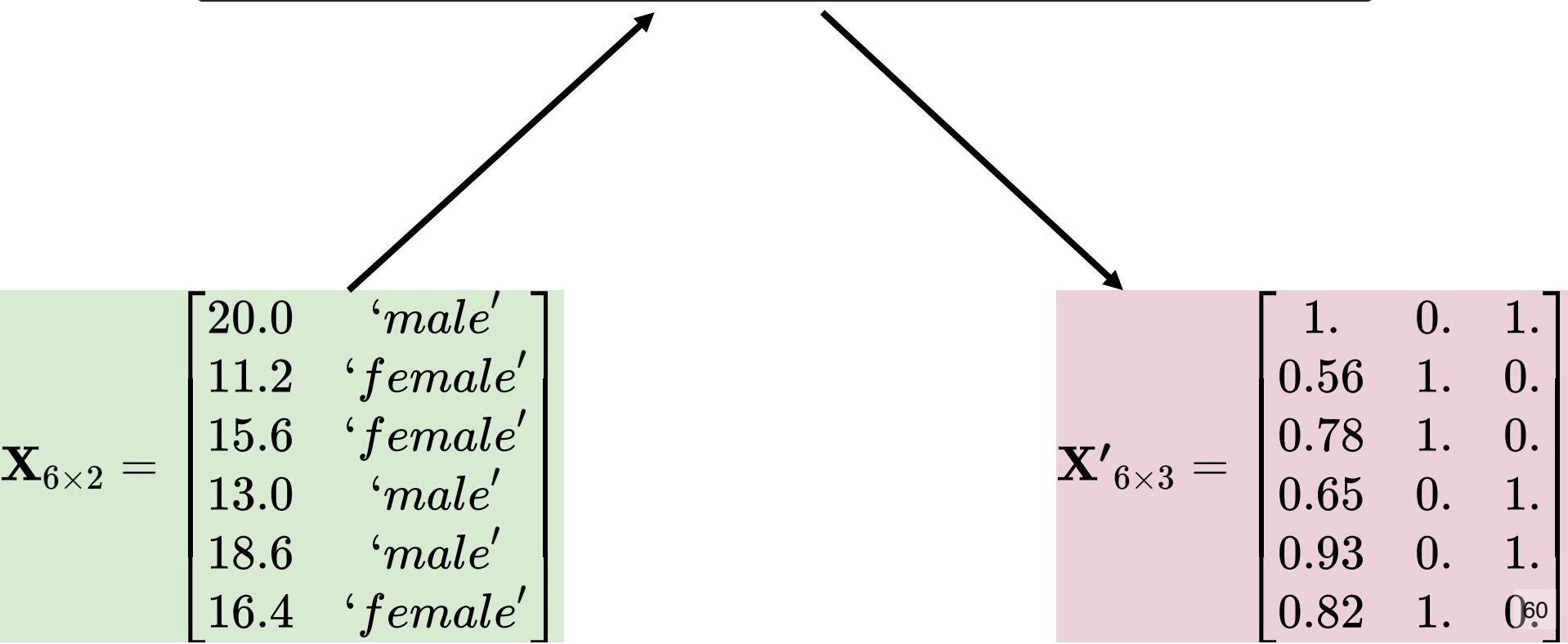
Consider following feature matrix, which represent weight and gender of a class of students.

$$\mathbf{X}_{6 \times 2} = \begin{bmatrix} 20.0 & 'male' \\ 11.2 & 'female' \\ 15.6 & 'female' \\ 13.0 & 'male' \\ 18.6 & 'male' \\ 16.4 & 'female' \end{bmatrix}$$

Here, first column is numeric, however, second column is categorical, therefore different transformers have to be applied on them.

In this example, let's apply **MaxAbsScaler** on the numeric column and **OneHotEncoder** on categorical column.

```
column_trans = ColumnTransformer(  
    [ ('ageScaler', MaxAbsScaler(), [0]),  
     ('genderEncoder', OneHotEncoder(dtype='int'), [1])],  
    remainder='drop', verbose_feature_names_out=False)  
column_trans.fit_transform(X)
```



Transforming Target for Regression

TransformedTargetRegressor

- Transforms the target variable `y` before fitting a regression model.
- The predicted values are mapped back to the original space via an inverse transform.
- `TransformedTargetRegressor` takes `regressor` and `transformer` to be applied to the target variable as arguments.

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.compose import TransformedTargetRegressor
tt = TransformedTargetRegressor(regressor=LinearRegression(),
                                 func=np.log, inverse_func=np.exp)
X = np.arange(4).reshape(-1, 1)
y = np.exp(2 * X).ravel()
tt.fit(X, y)
```

Part 3: Dimensionality reduction

Another way to reduce the number of feature is through **unsupervised dimensionality reduction** techniques.

`sklearn.decomposition` module has a number of APIs for this task.

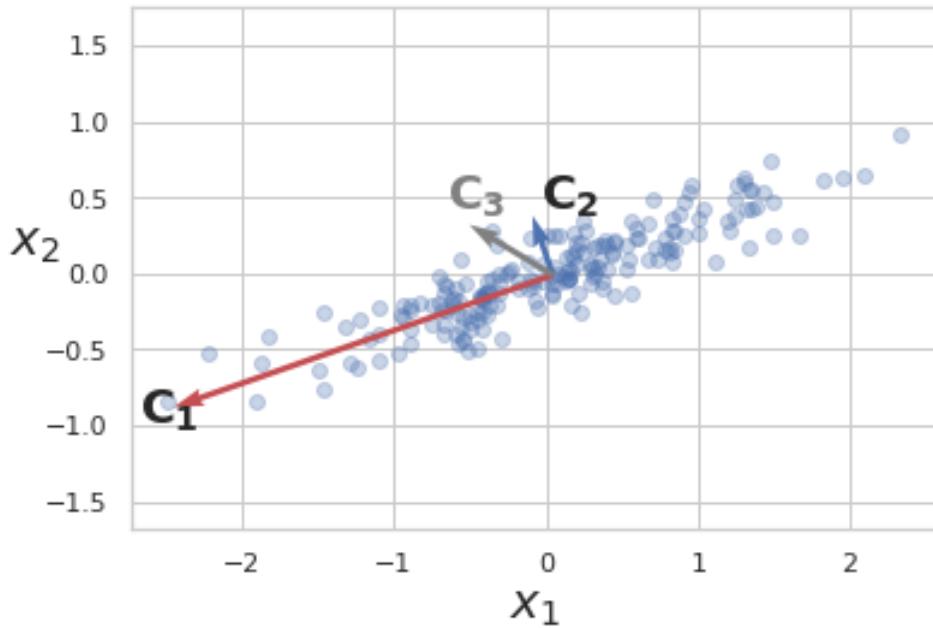
We will focus on how to perform feature reduction with **principle component analysis (PCA)** in `sklearn`.

PCA 101

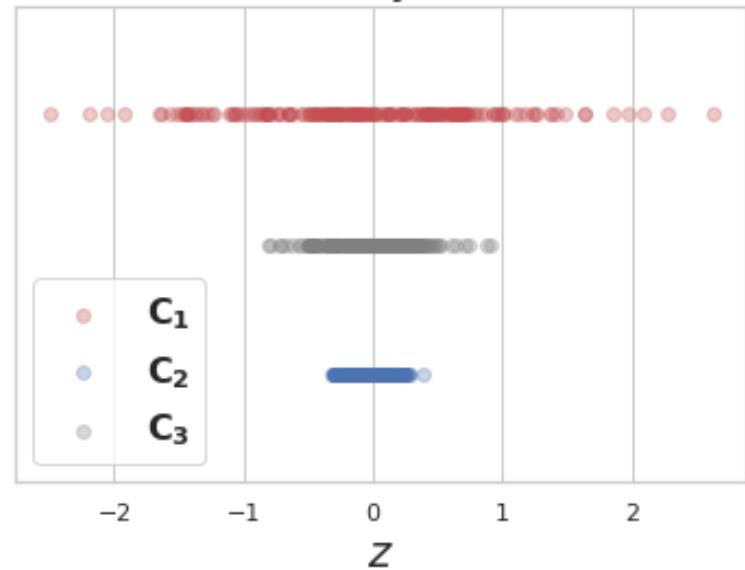
- PCA, is a linear dimensionality reduction technique.
- It uses singular value decomposition (SVD) to project the feature matrix or data to a lower dimensional space.
- The first principle component (PC) is in the direction of maximum variance in the data.
 - It captures bulk of the variance in the data.
- The subsequent PCs are orthogonal to the first PC and gradually capture lesser and lesser variance in the data.
- We can select first k PCs such that we are able to capture the desired variance in the data.

`sklearn.decomposition.PCA` API is used for performing PCA based dimensionality reduction.

PCA illustration

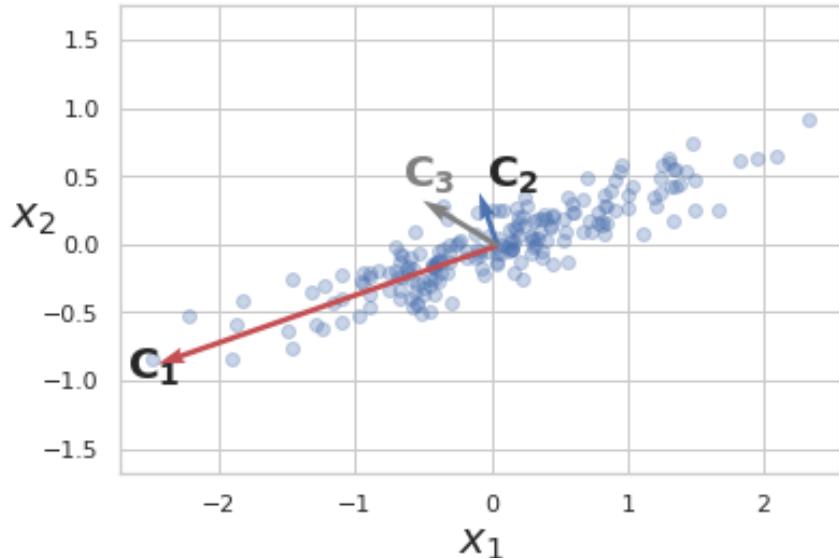


Variance covered by different vectors

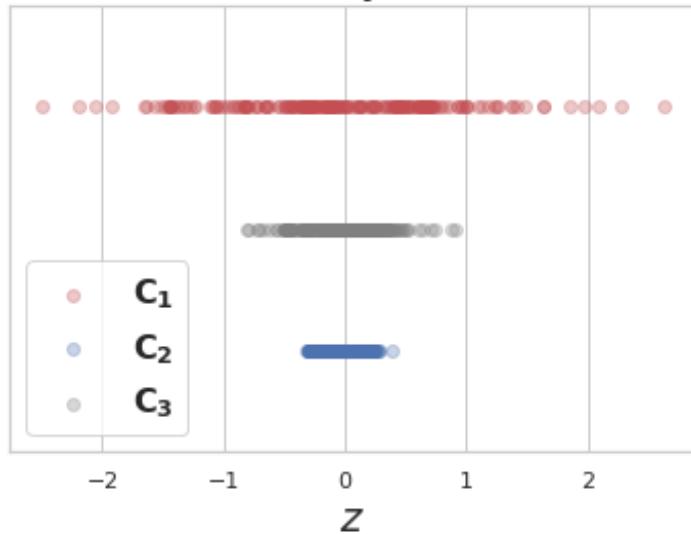


- Blue dots are data points.
- x_1 and x_2 are features.
- C_1 , C_2 and C_3 are candidate PCs.

- z represents projection of data points on a candidate vector.

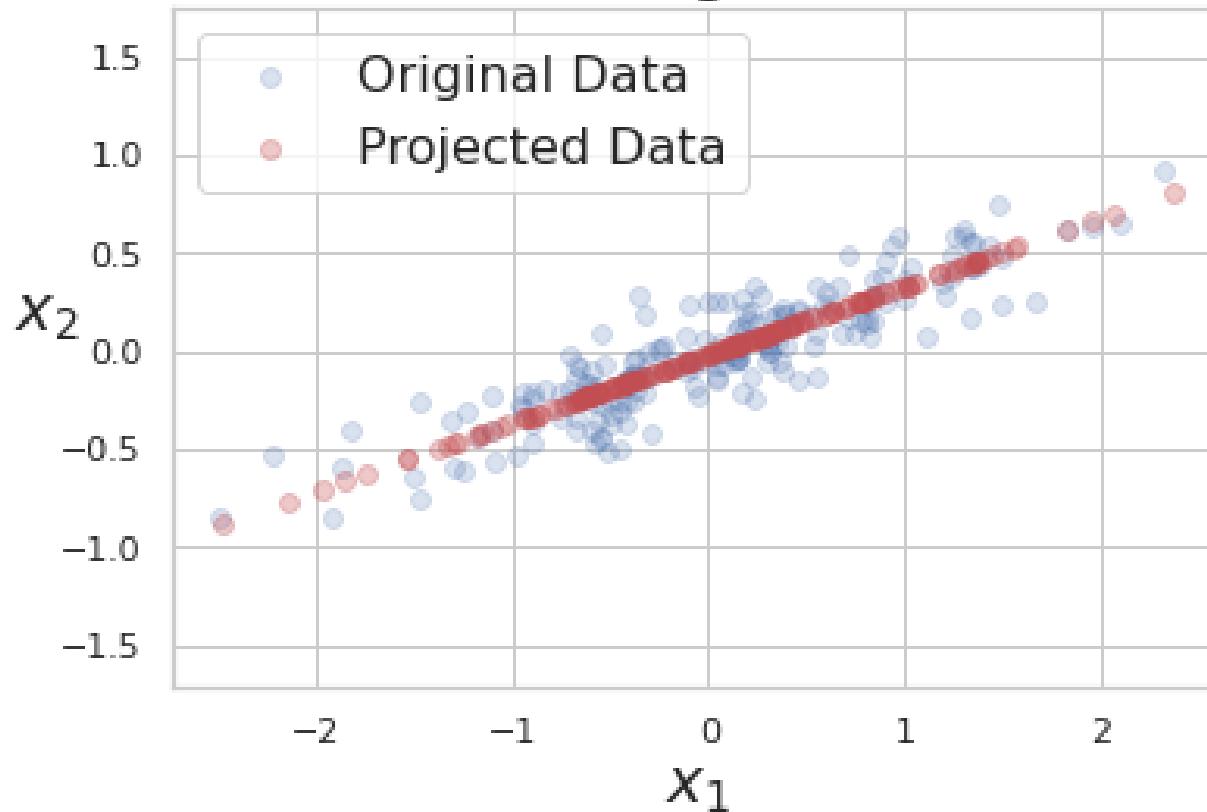


Variance covered by different vectors



Out of 3 candidate vectors to project data on, vector C_1 captures most of the variance, hence it is the first PC and C_2 , which is orthogonal to it is the second PC

PCA: Reducing Dimensions



After applying PCA and choosing only first PC to reduce dimension of data.

Part 4: Chaining transformers

The preprocessing transformations are applied one after another on the input feature matrix.

```
si = SimpleImputer()  
X_imputed = si.fit_transform(X)  
ss = StandardScaler()  
X_scaled = ss.fit_transform(X_imputed)
```

It is important to apply **exactly same transformation** on training, evaluation and test set **in the same order**.

Failing to do so would lead to **incorrect predictions** from model due to **distribution shift** and hence **incorrect performance evaluation**.

The `sklearn.pipeline` module provides utilities to build a **composite estimator**, as a **chain of transformers and estimators**.

There are two classes: (i) `Pipeline` and (ii) `FeatureUnion`.

Class	Usage
Pipeline	Constructs a chain of multiple transformers to execute a fixed sequence of steps in data preprocessing and modelling.
FeatureUnion	Combines output from several transformer objects by creating a new transformer from them.

`sklearn.pipeline.Pipeline`

Sequentially apply a list of **transformers** and **estimators**.

Intermediate steps of the pipeline must be '**transformers**' that is, they must implement **fit** and **transform** methods.

The **final estimator** only needs to implement **fit**.

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters.

Creating Pipelines

Two ways to create a pipeline object.

Pipeline()

- It takes a list of
`('estimatorName', estimator(...))` tuples.
- The pipeline object exposes interface of the last step.

```
estimators = [  
    ('simpleImputer', SimpleImputer()),  
    ('standardScaler', StandardScaler()),  
]  
pipe = Pipeline(steps=estimators)
```

make_pipeline

- It takes a number of estimator objects only.

```
pipe = make_pipeline(SimpleImputer(),  
                     StandardScaler())
```

Without pipeline:

```
si = SimpleImputer()  
X_imputed = si.fit_transform(X)  
ss = StandardScaler()  
X_scaled = ss.fit_transform(X_imputed)
```

With pipeline:

```
estimators = [  
    ('simpleImputer', SimpleImputer()),  
    ('standardScaler', StandardScaler()),  
]  
pipe = Pipeline(steps=estimators)  
pipe.fit_transform(X)
```

Accessing individual steps in Pipeline

```
estimators = [
    ('simpleImputer', SimpleImputer()),
    ('pca', PCA()),
    ('regressor', LinearRegression())
]
pipe = Pipeline(steps=estimators)
```

Total # steps: 3

1. SimpleImputer
2. PCA
3. LinearRegression

The second estimator can be accessed in following 4 ways:

- `pipe.named_steps.pca`
- `pipe.steps[1]`
- `pipe[1]`
- `pipe['pca']`

Accessing parameters of each step in Pipeline

Parameters of the estimators in the pipeline can be accessed using the `<estimator>__<parameterName>` syntax, note there are two underscores between `<estimator>` and `<parameterName>`

```
estimators = [
    ('simpleImputer', SimpleImputer()),
    ('pca', PCA()),
    ('regressor', LinearRegression())
]
pipe = Pipeline(steps=estimators)

pipe.set_params(pca__n_components = 2)
```

In above example `n_components` of `PCA()` step is set after the pipeline is created.

Performing grid search with pipeline

By using naming convention of nested parameters, grid search can implemented.

```
param_grid = dict(imputer=['passthrough',
                           SimpleImputer(),
                           KNNImputer()],
                  clf=[SVC(), LogisticRegression()],
                  clf__C=[0.1, 10, 100])
grid_search = GridSearchCV(pipe, param_grid=param_grid)
```

- `c` is an inverse of regularization, lower its value stronger the regularization is.
- In the example above `clf__c` provides a set of values for grid search.

Caching transformers

- Transforming data is a computationally expensive step.
- For grid search, transformers need not be applied for every parameter configuration. They can be applied only once, and the transformed data can be reused.
- This can be achieved by setting `memory` parameter of a pipeline object.
- `memory` can take either location of a directory in string format or `joblib.Memory` object.

```
estimators = [  
    ('simpleImputer', SimpleImputer()),  
    ('pca', PCA(2)),  
    ('regressor', LinearRegression())  
]  
pipe = Pipeline(steps=estimators, memory = '/path/to/cache/dir')
```

Advantages of pipeline

- Combines multiple steps of end to end ML into single object such as missing value imputation, feature scaling and encoding, model training and cross validation.
- Enables joint grid search over parameters of all the estimators in the pipeline.
- Makes configuring and tuning end to end ML quick and easy.
- Offers convenience, as a developer has to call `fit()` and `predict()` methods only on a `Pipeline` object (assuming last step in the pipeline is an estimator).
- Reduces code duplication: With a `Pipeline` object, one doesn't have to repeat code for preprocessing and transforming the test data.

`sklearn.pipeline.FeatureUnion`

- Concatenates results of multiple transformer objects.
- Applies a list of transformer objects in parallel, and their outputs are concatenated side-by-side into a larger matrix.
- `FeatureUnion` and `Pipeline` can be used to create complex transformers.

Combining Transformers and Pipelines

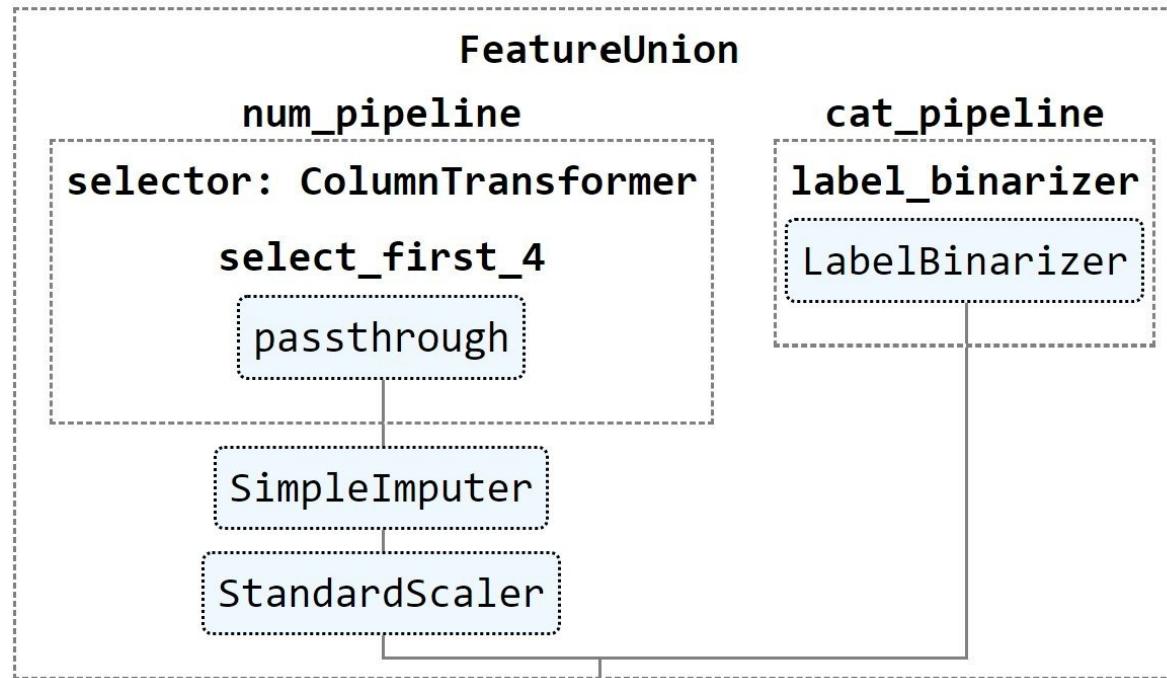
- `FeatureUnion()` accepts a list of tuples.
- Each tuple is of the format:
 - (`'estimatorName'`, `estimator(...)`)

```
num_pipeline = Pipeline([('selector', ColumnTransformer([('select_first_4',
                                                       'passthrough',
                                                       slice(0,4))]),
                         ('imputer', SimpleImputer(strategy="median")),
                         ('std_scaler', StandardScaler())),
                        ])
cat_pipeline = ColumnTransformer([('label_binarizer', LabelBinarizer(), [4]),
                                  ])
full_pipeline = FeatureUnion(transformer_list=
    [("num_pipeline", num_pipeline),
     ("cat_pipeline", cat_pipeline),])
```

Visualizing Composite Transformers

```
from sklearn import set_config  
set_config(display='diagram')  
# displays HTML representation in a jupyter context  
full_pipeline
```

It creates the following visualization



That's it from data preprocessing.

Only way to master is to practice it with examples.

Read more about these methods in [sklearn user guide on data transformation](#) and documentation of different classes.

Linear Regression

Machine Learning Practice

Dr. Ashish Tendulkar

IIT Madras

How to build baseline regression model?

DummyRegressor helps in creating a **baseline** for regression.

```
1 from sklearn.dummy import DummyRegressor  
2  
3 dummy_regr = DummyRegressor(strategy="mean")  
4 dummy_regr.fit(X_train, y_train)  
5 dummy_regr.predict(X_test)  
6 dummy_regr.score(X_test, y_test)
```

- It makes a prediction as specified by the **strategy**.
- Strategy is based on **some statistical property** of the training set or **user specified value**.

Strategy

mean

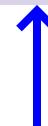
median

quantile

constant

quantile

constant



How is Linear Regression model trained?

Step 1: Instantiate **object** of a suitable **linear regression estimator** from one of the following two options

Normal
equation

```
1 from sklearn.linear_model import LinearRegression  
2 linear_regressor = LinearRegression()
```

Iterative
optimization

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor()
```

Step 2: Call **fit** method on **linear regression object** with **training feature matrix** and **label vector** as arguments.

```
1 # Model training with feature matrix X_train and  
2 # label vector or matrix y_train  
3 linear_regressor.fit(X_train, y_train)
```

Works for both single and multi-output regression.

SGDRegressor Estimator

SGDRegressor Estimator

- Implements stochastic gradient descent
- Use for large training set up (> 10k samples)
- Provides greater control on optimization process through provision for hyperparameter settings.

- `loss= 'squared error'`
- `loss = 'huber'`

- `penalty = 'l1'`

- `penalty = 'l2'`

- `penalty = 'elasticnet'`

SGDRegressor

- `learning_rate = 'constant'`
- `learning_rate = 'optimal'`
- `learning_rate = 'invscaling'`
- `learning_rate = 'adaptive'`

- `early_stopping = 'True'`
- `early_stopping = 'False'`

It's a good idea to use a `random seed` of your choice while instantiating SGDRegressor object. It helps us get **reproducible results**.

Set `random_state` to seed of your choice.

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(random_state=42)
```

Note: In the rest of the presentation, we won't set the random seed for sake of brevity. However while coding, always set the random seed in the constructor.

How to perform feature scaling for SGDRegressor?

SGD is **sensitive to feature scaling**, so it is **highly recommended to scale** input feature matrix.

```
1 from sklearn.linear_model import SGDRegressor
2 from sklearn.pipeline import Pipeline
3 from sklearn.preprocessing import StandardScaler
4
5 sgd = Pipeline([
6         ('feature_scaling', StandardScaler()),
7         ('sgd_regressor', SGDRegressor())])
8
9 sgd.fit(X_train, y_train)
```

Note

- Feature scaling **is not needed** for **word frequencies** and **indicator features** as they have intrinsic scale.
- Features extracted using PCA should be **scaled by some constant c** such that the average L2 norm of the training data equals one.

How to shuffle training data after each epoch in SGDRegressor?

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(shuffle=True)
```

How to use set learning rate in SGDRegressor?

- `learning_rate = 'constant'`
- `learning_rate = 'invscaling'`
- `learning_rate = 'adaptive'`

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(random_state=42)
```

What is the default setting?

- `learning_rate = 'invscaling'`
- `eta0 = 1e-2`
- `power_t = 0.25`

Learning rate reduces after every iteration:

$$\text{eta} = \text{eta0} / \text{pow}(t, \text{power_t})$$

Note: You can make changes to these parameters to speed up or slow down the training process.

How to use set constant **learning rate** ?

- **learning_rate = 'constant'**

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(learning_rate='constant',  
3                                 eta0=1e-2)
```

Constant learning rate **eta0 = 1e-2** is used throughout the training.

How to set adaptive learning rate?

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(learning_rate='adaptive',  
3                                 eta0=1e-2)
```

- The learning rate is kept to **initial value as long as the training loss decreases.**
- When the **stopping criterion** is reached, the **learning rate is divided by 5**, and the **training loop continues**.
- The algorithm stops when the learning rate goes below 10^{-6} .

How to set #epochs in SGDRegressor?

Set `max_iter` to desired `#epochs`. The default value is 1000.

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(max_iter=100)
```

Remember `one epoch` is `one full pass over the training data`.

Practical tip

SGD converges after observing approximately 10^6 training samples. Thus, a reasonable first guess for the number of iterations for n sampled training set is

$$\text{max_iter} = \text{np.ceil}(10^6/n)$$

How to set stopping criteria in SGDRegressor?

Option #1 `tol`, `n_iter_no_change`, `max_iter`.

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(loss='squared_error',  
3                                  max_iter=500,  
4                                  tol=1e-3,  
5                                  n_iter_no_change=5)
```

The SGDRegressor stops

- when the training loss does not improve (`loss > best_loss - tol`) for `n_iter_no_change` consecutive epochs
- else after a maximum number of iteration `max_iter`.

How to set **stopping criteria** in SGDRegressor?

Option #2 **early_stopping**, **validation_fraction**

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(loss='squared_error',  
3                                     early_stopping=True  
4                                     max_iter=500,  
5                                     tol=1e-3,  
6                                     validation_fraction=0.2,  
7                                     n_iter_no_change=5)
```

Set aside **validation_fraction** percentage records from training set as validation set. Use **score** method to obtain validation score.

The SGDRegressor **stops** when

- **validation score** does not improve by at least **tol** for **n_iter_no_change** consecutive epochs.
- else after a maximum number of iteration **max_iter**.

How to use different **loss** functions in SGDRegressor?

Set **loss** parameter to one of the supported values

'**squared_error**' {studied in this course}

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(loss='squared_error')
```

It also supports other losses as documented in [sklearn API](#)

How to use averaged SGD?

Averaged SGD updates the weight vector to **average of weights** from previous updates.

Option #1: Averaging across all updates **average=True**

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(average=True)
```

Option #2: Set **average** to int value.

Averaging begins once the total number of samples seen reaches **average**

Setting **average=10** starts averaging after seeing 10 samples

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(average=10)
```

Averaged SGD works **best** with a **larger number of features** and a **higher eta0**

How do we initialize SGD with weight vector of the previous run?

Set **warm_start = TRUE**

while instantiating object of SGDRegressor

```
1 from sklearn.linear_model import SGDRegressor  
2 linear_regressor = SGDRegressor(warm_start=True)
```

By default **warm_start = False**

How to monitor SGD loss iteration after iteration?

Make use of **warm_start = TRUE**

```
1 sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
2                         penalty=None, learning_rate="constant", eta0=0.0005)
3
4 for epoch in range(1000):
5     sgd_reg.fit(X_train, y_train) # continues where it left off
6     y_val_predict = sgd_reg.predict(X_val)
7     val_error = mean_squared_error(y_val, y_val_predict)
```

Model inspection

How to access the weights of trained [Linear Regression](#) model?

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_m x_m = \mathbf{w}^T \mathbf{x}$$

The weights w_1, w_2, \dots, w_m are stored in `coef_` class variable.

```
1 linear_regressor.coef_
```

The intercept w_0 is stored in `intercept_` class variable.

```
1 linear_regressor.intercept_
```

Note: These code snippets works for both [LinearRegression](#) and [SGDRegressor](#), and for that matter to [all regression estimators](#) that we will study in this module. Why?

All of them are estimators.

Model inference

How to make predictions on new data in Linear Regression model?

Step 1: Arrange data for prediction in a feature matrix of shape (#samples, #features) or in sparse matrix format.

Step 2: Call `predict` method on `linear regression` object with `feature matrix` as an argument.

```
1 # Predict labels for feature matrix X_test  
2 linear_regressor.predict(X_test)
```

Same code works for all regression estimators.

Model evaluation

General steps in model evaluation

STEP 1: Split data into train and test

```
1 from sklearn.model_selection import train_test_split  
2 X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

STEP 2: Fit linear regression estimator on training set.

STEP 3: Calculate training error (a.k.a. empirical error)

STEP 4: Calculate test error (a.k.a. generalization error)

Compare training and test errors

How to evaluate trained Linear Regression model?

Using `score` method on linear regression object:

```
1 # Evaluation on the eval set with
2 # 1. feature matrix
3 # 2. label vector or matrix (single/multi-output)
4 linear_regressor.score(x_test, y_test)
```

The score returns R^2 or coefficient of determination

$$R^2 = \left(1 - \frac{u}{v}\right)$$

residual sum of squares:

$$u = (\mathbf{X}\mathbf{w} - \mathbf{y})^T(\mathbf{X}\mathbf{w} - \mathbf{y})$$

Sum of squared error
(actual and predicted label)

total sum of square

Sum of squared error
(actual and mean predicted
label)

$$v = (\mathbf{y} - \hat{\mathbf{y}}_{\text{mean}})^T(\mathbf{y} - \hat{\mathbf{y}}_{\text{mean}})$$

The score returns R^2 or coefficient of determination

$$R^2 = \left(1 - \frac{u}{v}\right)$$

When?

- The best possible score is 1.0.

u , sum of squared error = 0

- A constant model that always predicts the expected value of y , would get a score of 0.0.

$u = v$

- The score can be negative (because the model can be arbitrarily worse).

Evaluation metrics

sklearn provides a **bunch of regression metrics** to evaluate **performance** of the **trained estimator** on the **evaluation set**.

mean_absolute_error

```
1 from sklearn.metrics import mean_absolute_error  
2 eval_score = mean_absolute_error(y_test, y_predicted)
```

mean_squared_error

```
1 from sklearn.metrics import mean_squared_error  
2 eval_score = mean_squared_error(y_test, y_predicted)
```

r2_score Same as output of **score**

```
1 from sklearn.metrics import r2_score  
2 eval_score = r2_score(y_test, y_predicted)
```

These metrics can also be used in **multi-output regression** setup.

mean_squared_log_error

```
1 from sklearn.metrics import mean_squared_log_error  
2 eval_score = mean_squared_log_error(y_test, y_predicted)
```

- Useful for **targets with exponential growths** like population, sales growth etc,
- **Penalizes under-estimation heavier than the over-estimation.**

mean_absolute_percentage_error

```
1 from sklearn.metrics import mean_absolute_percentage_error  
2 eval_score = mean_absolute_percentage_error(y_test, y_predicted)
```

- Sensitive to relative error.

median_absolute_error

```
1 from sklearn.metrics import median_absolute_error  
2 eval_score = median_absolute_error(y_test, y_predicted)
```

- Robust to outliers

How to evaluate regression model on worst case error?

Use metrics **max_error**

Worst case error on train set can be calculated as follows:

```
1 from sklearn.metrics import max_error  
2 train_error = max_error(y_train, y_predicted)
```

Worst case error on test set can be calculated as follows:

```
1 from sklearn.metrics import max_error  
2 test_error = max_error(y_test, y_predicted)
```

This metrics can, however, be used only for **single output regression**. It **does not support multi-output regression**.

Scores and Errors

- Score is a metric for which higher value is better.
- Error is a metric for which lower value is better.

Convert error metric to score metric by adding `neg_` suffix.

Function	Scoring
<code>metrics.mean_absolute_error</code>	<code>neg_mean_absolute_error</code>
<code>metrics.mean_squared_error</code>	<code>neg_mean_squared_error</code>
<code>metrics.mean_squared_error</code>	<code>neg_root_mean_squared_error</code>
<code>metrics.mean_squared_log_error</code>	<code>neg_mean_squared_log_error</code>
<code>metrics.median_absolute_error</code>	<code>neg_median_absolute_error</code>

In case, we get comparable performance on train and test with this split, is this performance guaranteed on other splits too?

- Is test set sufficiently large?
 - In case it is small, the test error obtained may be unstable and would not reflect the true test error on large test set.
- What is the chance that the easiest examples were kept aside as test by chance?
 - This if happens would lead to optimistic estimation of the true test error.

We use cross validation for robust performance evaluation.

Cross-validation performs **robust evaluation** of model performance

- by **repeated splitting** and
- providing **many training and test errors**

This enables us to **estimate variability in generalization performance** of the model.

sklearn implements the following cross validation iterators

KFold

RepeatedKFold

LeaveOneOut

ShuffleSplit

How to obtain cross validated performance measure using KFold?

```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.linear_model import linear_regression
3
4 lin_reg = linear_regression()
5 score = cross_val_score(lin_reg, X, y, cv=5)
```

- Uses KFold cross validation iterator, that divides training data into 5 folds.
- In each run, it uses 4 folds for training and 1 for evaluation.

Alternate way of writing the same thing

```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.model_selection import KFold
3 from sklearn.linear_model import linear_regression
4
5 lin_reg = linear_regression()
6 kfold_cv = KFold(n_splits=5, random_state=42)
7 score = cross_val_score(lin_reg, X, y, cv=kfold_cv)
```

How to obtain cross validated performance measure using `LeaveOneOut`?

```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.model_selection import LeaveOneOut
3 from sklearn.linear_model import linear_regression
4
5 lin_reg = linear_regression()
6 loocv = LeaveOneOut()
7 score = cross_val_score(lin_reg, X, y, cv=loocv)
```

which is same as

```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.model_selection import KFold
3 from sklearn.linear_model import linear_regression
4
5 lin_reg = linear_regression()
6 n = X.shape[0]
7 kfold_cv = KFold(n_splits=n)
8 score = cross_val_score(lin_reg, X, y, cv=kfold_cv)
```

How to obtain cross validated performance measure using **ShuffleSplit**?

```
1 from sklearn.linear_model import linear_regression  
2 from sklearn.model_selection import cross_val_score  
3 from sklearn.model_selection import ShuffleSplit  
4  
5 lin_reg = linear_regression()  
6 shuffle_split = ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)  
7 score = cross_val_score(lin_reg, X, y, cv=shuffle_split)
```

It is also called **random permutation** based **cross validation strategy**.

- Generates user defined number of train/test splits.
- It is robust to class distribution.

In each iteration, it shuffles order of data samples and then splits it into train and test.

How to specify a performance measure in **`cross_val_score`**

```
1 from sklearn.linear_model import linear_regression
2 from sklearn.model_selection import cross_val_score
3 from sklearn.model_selection import ShuffleSplit
4
5 lin_reg = linear_regression()
6 shuffle_split = ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
7 score = cross_val_score(lin_reg, X, y, cv=shuffle_split,
8                         scoring='neg_mean_absolute_error')
```

scoring parameter can be set to one of the scoring schemes implemented in sklearn as follows

max_error **r2**

neg_mean_absolute_error **neg_mean_squared_error**

neg_mean_squared_log_error **neg_median_absolute_error**

neg_root_mean_squared_error

How to obtain **test scores** from different folds?

```
1 from sklearn.model_selection import cross_validate
2 from sklearn.model_selection import ShuffleSplit
3
4 cv = ShuffleSplit(n_splits=40, test_size=0.3, random_state=0)
5 cv_results = cross_validate(
6     regressor, data, target, cv=cv, scoring="neg_mean_absolute_error")
```

The results are stored in python dictionary with the following keys:

fit_time

score_time

test_score

estimator (optional)

train_score (optional)

How to obtain **trained estimators** and **scores** on training data during cross validation?

- For trained estimator, set **`return_estimator = True`**
- For scores on training set, set **`return_train_score = True`**

```
1 from sklearn.model_selection import cross_validate
2 from sklearn.model_selection import ShuffleSplit
3
4 cv = ShuffleSplit(n_splits=40, test_size=0.3,
5                     random_state=0)
6 cv_results = cross_validate(
7     regressor, data, target,
8     cv=cv, scoring="neg_mean_absolute_error",
9     return_train_score=True,
10    return_estimator=True)
```

The estimators can be accessed through **`estimator`** key of the dictionary returned by **`cross_validate`**

How to evaluate **multiple metrics** of regression in cross validation set up?

```
1 from sklearn.model_selection import cross_validate
2 from sklearn.model_selection import ShuffleSplit
3
4 cv = ShuffleSplit(n_splits=40, test_size=0.3,
5                    random_state=0)
6 cv_results = cross_validate(
7     regressor, data, target,
8     cv=cv,
9     scoring=[ "neg_mean_absolute_error", "neg_mean_squared_error" ]
10    return_train_score=True,
11    return_estimator=True)
```

cross_validate allows us to specify multiple scoring metrics
unlike **cross_val_score**

How to study effect of #samples on training and test errors?

STEP 1: Instantiate an object of `learning_curve` class with `estimator`, `training data`, `size`, `cross validation strategy` and `scoring scheme` as arguments.

```
1 from sklearn.model_selection import learning_curve
2
3 results = learning_curve(
4     lin_reg, X_train, y_train, train_sizes=train_sizes, cv=cv,
5     scoring="neg_mean_absolute_error")
6 train_size, train_scores, test_scores = results[:3]
7 # Convert the scores into errors
8 train_errors, test_errors = -train_scores, -test_scores
```

STEP 2: Plot `training and test scores` as function of the `size` of training sets. And make assessment about model fitment: `under/overfitting` or `right fit`.

Underfitting/Overfitting diagnosis

STEP 1: Fit linear models with different number of features.

STEP 2: For each model, obtain training and test errors.

STEP 3: Plot #features vs error graph - one each for training and test errors.

STEP 4: Examine the graphs to detect under/overfitting.

We can replace **#features** with **any other tunable hyperparameter** to do this diagnosis for **setting that hyperparameter to the appropriate value**.

Polynomial regression

How is polynomial regression model trained?

Step 1: Apply polynomial transformation on the feature matrix.

Step 2: Learn linear regression model (via normal equation or SGD) on the transformed feature matrix.

Implementation tips: Make use of pipeline construct for polynomial transformation followed by linear regression estimator.

Set up polynomial regression model with normal equation

```
1 from sklearn.linear_model import LinearRegression
2 from sklearn.pipeline import Pipeline
3 from sklearn.preprocessing import PolynomialFeatures
4
5 # Two steps:
6 # 1. Polynomial features of desired degree (here degree=2)
7 # 2. Linear regression
8 poly_model = Pipeline([
9         ('polynomial_transform', PolynomialFeatures(degree=2))),
10        ('linear_regression', LinearRegression())])
11
12 # Train with feature matrix X_train and label vector y_train
13 poly_model.fit(X_train, y_train)
```

Set up polynomial regression model with SGD

```
1 from sklearn.linear_model import SGDRegressor
2 from sklearn.pipeline import Pipeline
3 from sklearn.preprocessing import PolynomialFeatures
4
5 poly_model = Pipeline([
6         ('polynomial_transform', PolynomialFeatures(degree=2))),
7         ('sgd_regression', SGDRegressor())])
8 poly_model.fit(X_train, y_train)
```

Notice that there is a **single line code change** in two code snippets.

How to use only interaction features for polynomial regression?

```
1 from sklearn.preprocessing import PolynomialFeatures  
2 poly_transform = PolynomialFeatures(degree=2, interaction_only=True)
```

$[x_1, x_2]$ is transformed to $[1, x_1, x_2, x_1x_2]$.

Note that $[x_1^2, x_2^2]$ are excluded.

Hyperparameter tuning (HPT)

How to recognize hyperparameters in any sklearn estimator?

- Hyper-parameters are parameters that are not directly learnt within estimators.
- In `sklearn`, they are passed as arguments to the constructor of the estimator classes.

For example,

- `degree` in `PolynomialFeatures`
- `learning rate` in `SGDRegressor`

How to set these hyperparameters?

Select hyperparameters that results in the **best cross validation scores**.

Hyper parameter search consists of

- an estimator (regressor or classifier);
- a parameter space;
- a method for searching or sampling candidates;
- a cross-validation scheme; and
- a score function.

Two generic HPT approaches implemented in sklearn are:

- **GridSearchCV** exhaustively considers all parameter combinations for specified values.

```
1 param_grid = [
2     {'C': [1, 10, 100, 1000], 'kernel': ['linear']}
3 ]
```

- **RandomizedSearchCV** samples a given number of candidate values from a parameter space with a specified distribution.

```
1 param_dist = {
2     "average": [True, False],
3     "l1_ratio": stats.uniform(0, 1),
4     "alpha": loguniform(1e-4, 1e0),
5 }
```

What are the differences between `grid` and randomized search?

Grid search

Specifies exact values of parameters in grid

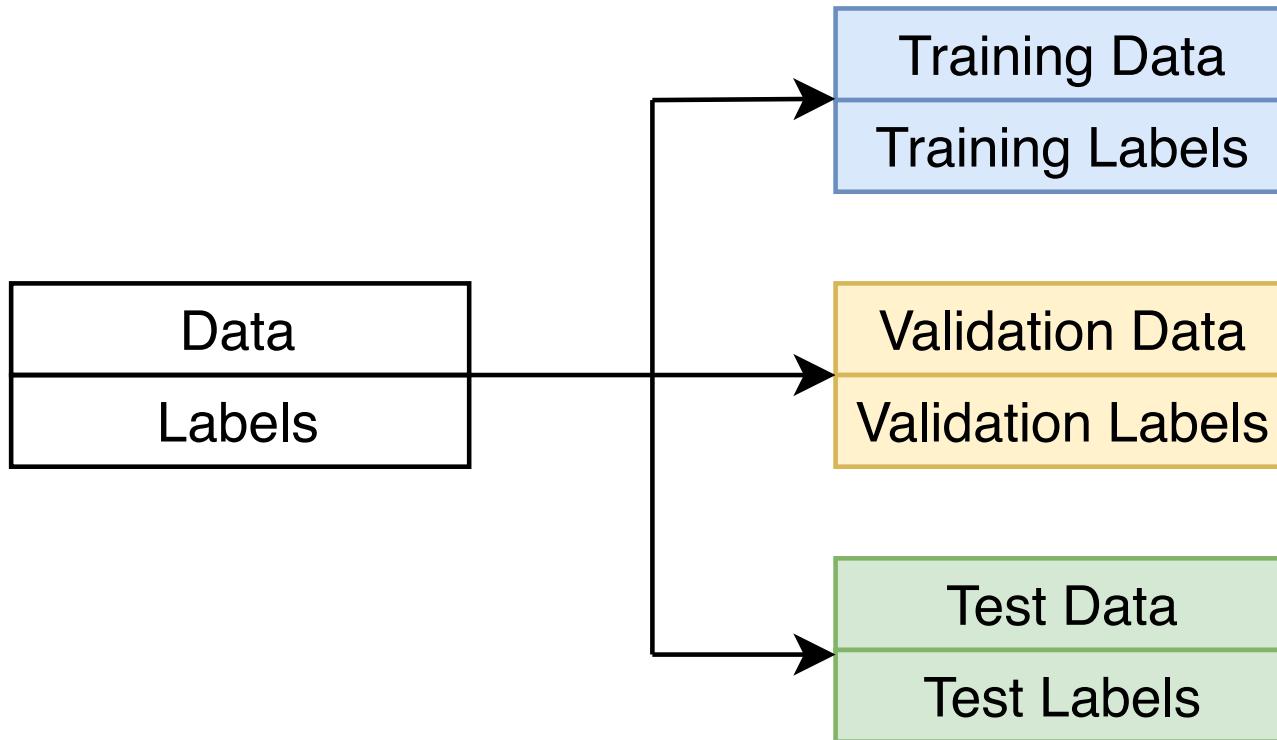
Randomized search

Specifies distributions of parameter values and values are sampled from those distributions.

Computational budget can be chosen independent of number of parameters and their possible values.

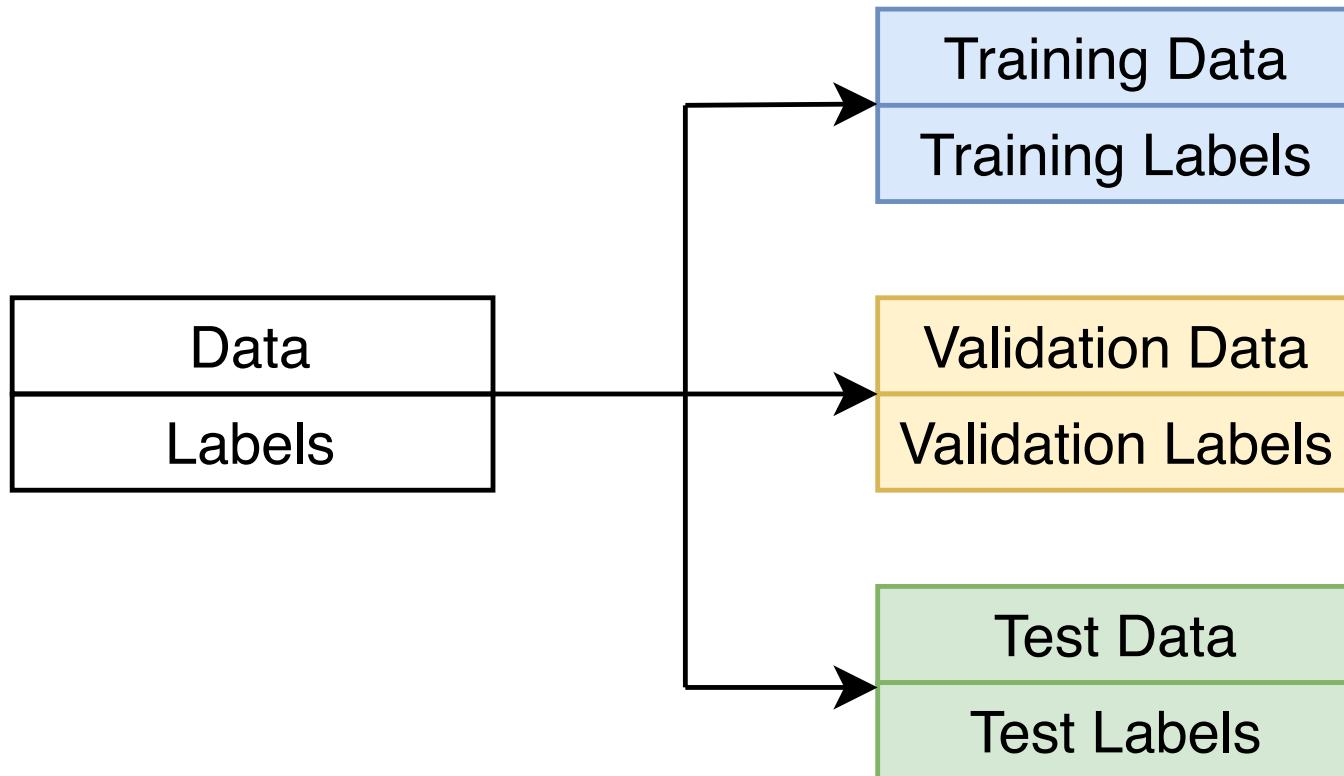
The budget is chosen in terms of the number of sampled candidates or the number of training iterations. Specified in `n_iter` argument

What data split is recommended for HPT?

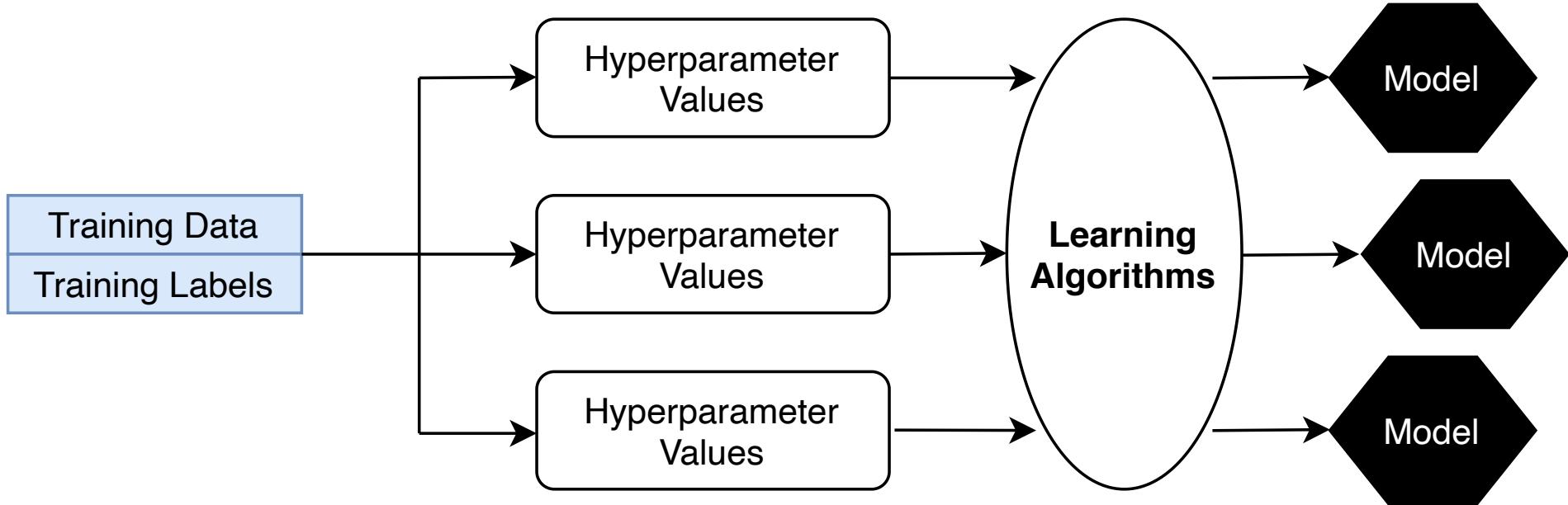


What are the steps in HPT?

STEP 1: Divide training data into **training**, **validation** and **test** sets.



STEP 2: For each combination of hyper-parameter values learn a model with training set.

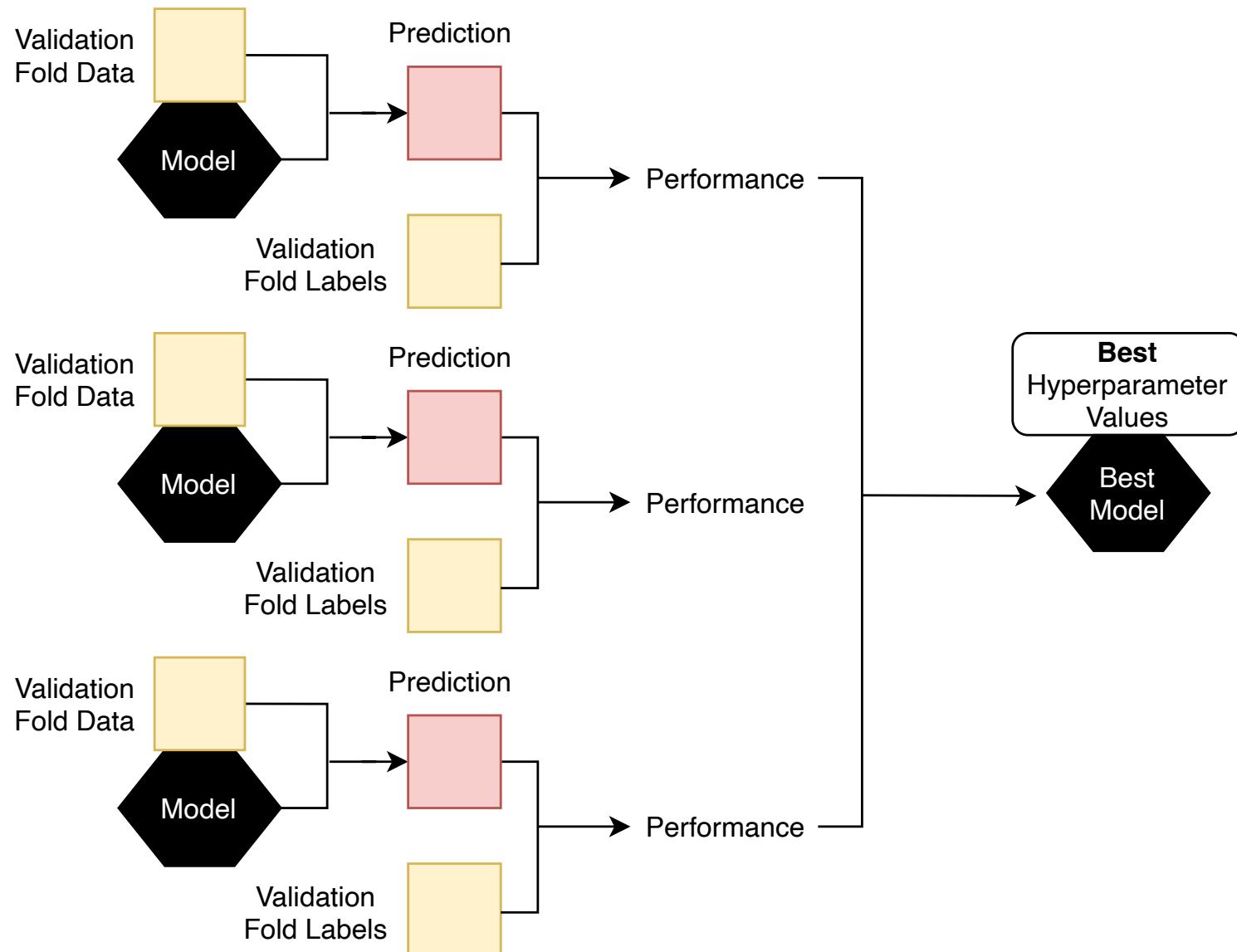


This step creates multiple models.

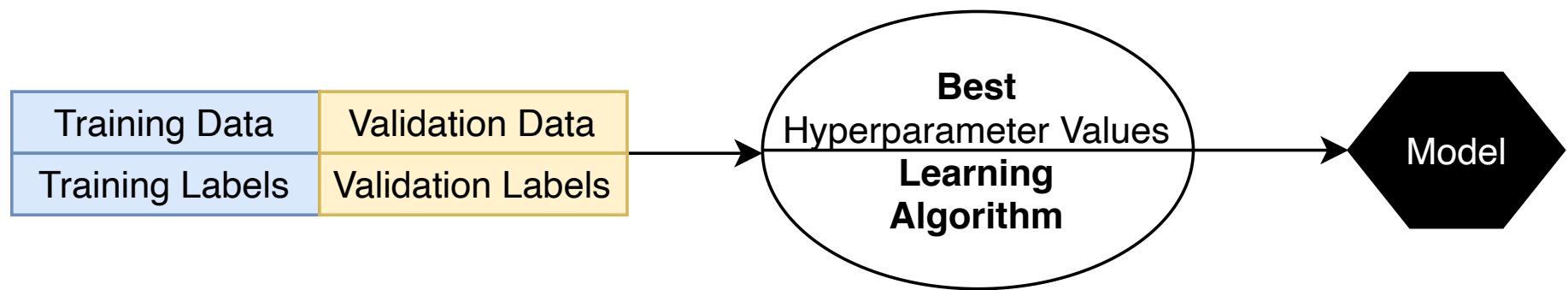
Tips

- This step can be run **in parallel** by setting `n_jobs = -1`.
- Some parameter combinations may cause failure in fitting one or more folds of data. This may cause the search to fail. Set `error_score = 0` (or `np.NaN`) to set score for the problematic fold to 0 and complete the search.

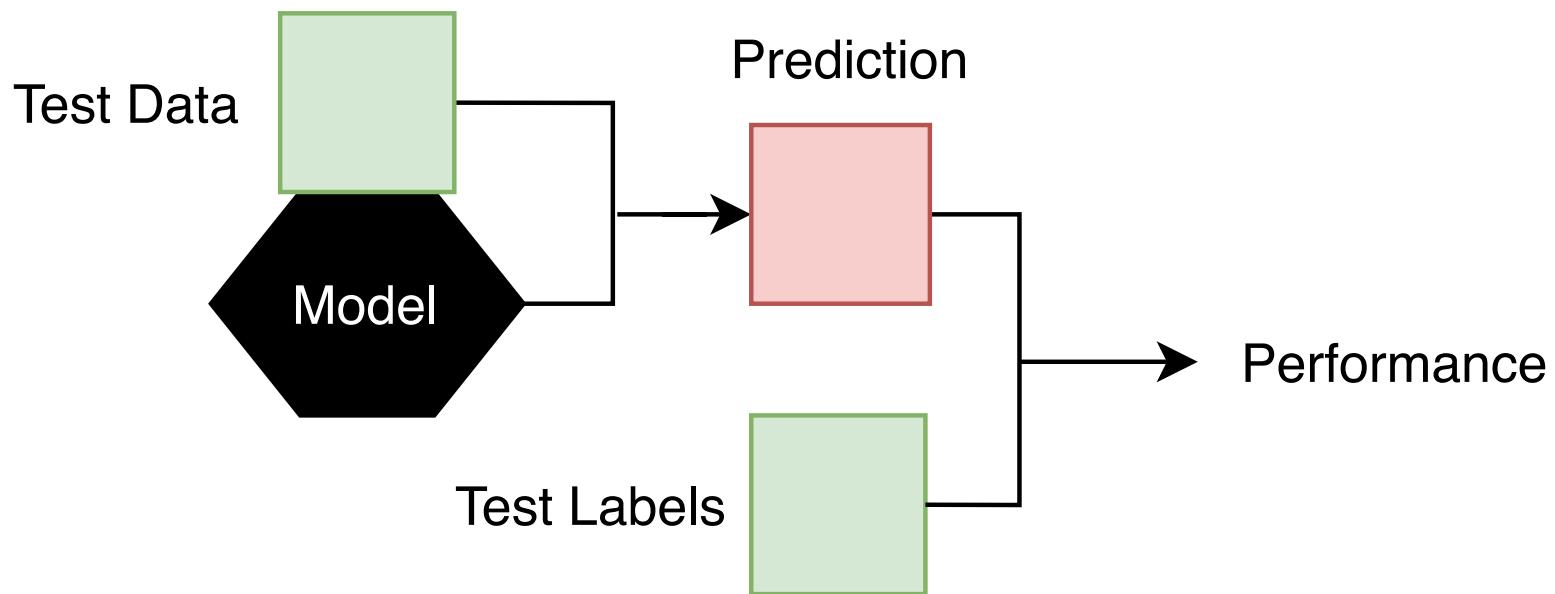
STEP 3: Evaluate performance of each model with validation set and select a model with the best evaluation score.



STEP 4: Retrain model with the best hyper-parameter settings on training and validation set combined.



STEP 5: Evaluate the model performance on the test set.



Note that the test set was not used in hyper-parameter search and model retraining .

What are some of model specific HPT available for regression tasks?

- Some models can fit data for a **range of values of some parameter** almost **as efficiently as** fitting the estimator for a **single value** of the parameter.
- This feature can be leveraged to perform **more efficient cross-validation** used for model selection of this parameter.

- `linear_model.LassoCV`
- `linear_model.RidgeCV`
- `linear_model.ElasticNetCV`

How to determine degree of polynomial regression with grid search?

```
1 from sklearn.model_selection import GridSearchCV
2 from sklearn.pipeline import Pipeline
3 from sklearn.preprocessing import PolynomialFeatures
4 from sklearn.linear_model import SGDRegressor
5
6 param_grid = [
7     {'poly_degree': [2, 3, 4, 5, 6, 7, 8, 9]}
8 ]
9
10 pipeline = Pipeline(steps=[('poly', PolynomialFeatures()),
11                         ('sgd', SGDRegressor())])
12
13 grid_search = GridSearchCV(pipeline, param_grid, cv=5,
14                           scoring='neg_mean_squared_error',
15                           return_train_score=True)
16
17 grid_search.fit(x_train.reshape(-1, 1), y_train)
```

Regularization

How to perform ridge regularization with specific regularization rate?

[Option #1]

Step 1: Instantiate object of **Ridge** estimator

Step 2: Set parameter **alpha** to the required regularization rate.

```
1 from sklearn.linear_model import Ridge  
2 ridge = Ridge(alpha=1e-3)
```

fit, **score**, **predict** work exactly like other linear regression estimators

[Option #2]

Step 1: Instantiate object of **SGDRegressor** estimator

Step 2: Set parameter **alpha** to the required regularization rate and **penalty = l2**.

```
1 from sklearn.linear_model import SGDRegressor  
2 sgd = SGDRegressor(alpha=1e-3, penalty='l2')
```

How to search the **best regularization** parameter for ridge?

[Option #1]

Search for the best regularization rate with built-in cross validation in [RidgeCV](#) estimator.

[Option #2]

Use cross validation with [Ridge](#) or [SVDRRegressor](#) to search for best regularization.

- Grid search
- Randomized search

How to perform ridge regularization in polynomial regression?

Set up a pipeline of **polynomial transformation** followed by the **ridge regressor**.

```
1 from sklearn.linear_model import Ridge  
2 from sklearn.pipeline import Pipeline  
3 from sklearn.preprocessing import PolynomialFeatures  
4  
5 poly_model = Pipeline([  
6     ('polynomial_transform', PolynomialFeatures(degree=2)),  
7     ('ridge', Ridge(alpha=1e-3))])  
8 poly_model.fit(x_train, y_train)
```

Instead of **Ridge**, we can use **SGDRegressor**, as shown on previous slide, to get equivalent formulation.

How to perform lasso regularization with specific regularization rate?

[Option #1]

Step 1: Instantiate object of **Lasso** estimator

Step 2: Set parameter **alpha** to the required regularization rate.

```
1 from sklearn.linear_model import Lasso  
2 lasso = Lasso(alpha=1e-3)
```

fit, **score**, **predict** work exactly like other linear regression estimators

[Option #2]

Step 1: Instantiate object of **SGDRegressor** estimator

Step 2: Set parameter **alpha** to the required regularization rate and **penalty = l1**.

```
1 from sklearn.linear_model import SGDRegressor  
2 sgd = SGDRegressor(alpha=1e-3, penalty='l1')
```

How to search the best regularization parameter for lasso regularization?

[Option #1]

Search for the best regularization rate with built-in cross validation in `LassoCV` estimator.

[Option #2]

Use cross validation with `Lasso` or `SVDRegressor` to search for best regularization.

- Grid search
- Randomized search

How to perform lasso regularization in polynomial regression?

Set up a pipeline of **polynomial transformation** followed by the **lasso regressor**.

```
1 from sklearn.linear_model import Lasso  
2 from sklearn.pipeline import Pipeline  
3 from sklearn.preprocessing import PolynomialFeatures  
4  
5 poly_model = Pipeline([  
6     ('polynomial_transform', PolynomialFeatures(degree=2)),  
7     ('lasso', Lasso(alpha=1e-3))])  
8 poly_model.fit(x_train, y_train)
```

Instead of **Lasso**, we can use **SGDRegressor** to get equivalent formulation.

How to perform both lasso and ridge regularization in polynomial regression?

Set up a pipeline of polynomial transformation followed by the SGDRegressor with `penalty = 'elasticnet'`

```
1 from sklearn.linear_model import Lasso
2 from sklearn.pipeline import Pipeline
3 from sklearn.preprocessing import PolynomialFeatures
4
5 poly_model = Pipeline([
6     ('polynomial_transform', PolynomialFeatures(degree=2)),
7     ('elasticnet', SGDRegressor(penalty='elasticnet',
8                                 l1_ratio=0.3))]
9 poly_model.fit(X_train, y_train)
```

Remember `elasticnet` is a convex combination of L1 (Lasso) and L2 (Ridge) regularization.

- In this example, we have set `l1_ratio` to 0.3, which means `l2_ratio = 1 - l1_ratio = 0.7`. L2 takes higher weightage in this formulation.

Summary

How to implement

- Different regression models: standard linear regression and polynomial regression.
- Regularization.
- Model evaluation through different error metrics and scores derived from them.
- Cross validation - different iterators
- Hyperparameter tuning via grid search and randomized search.

Appendix - More Details

Introduction

In this module, we will be covering the implementation aspects of models of linear regression.

First we will learn linear regression models with:

- Normal equation, which estimates model parameter with a closed-form solution.
- Iterative optimization approach of gradient descent and its variants namely batch, mini-batch and stochastic gradient descent.

Further, we will study the implementation of the **polynomial regression model**, that is capable of modelling **non-linear relationships** between features and labels.

- Since the **polynomial regression** uses more parameters (due to polynomial representation of the input), it is more **prone to overfitting**.
- We will study how to detect **overfitting with learning curves** and use of **regularization** to mitigate the risk of overfitting.

Recap

Let's recall components of Linear regression

Training Data

(features, label) or (X , y), where label y is a real number.

Model

The label is obtained by a **linear combination** (or **weighted sum**) of the **input features** and a **bias** (or *intercept*) term. The model $h_{\mathbf{w}}$ is given by

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_m x_m = \mathbf{w}^T \mathbf{x}$$

where,

- \hat{y} is the predicted value.
- \mathbf{X} is a **feature vector** $\{x_1, x_2, \dots, x_m\}$ for a given example with m features in total.
 - i -th feature: x_i
- Weight or parameter vector \mathbf{w} includes bias term too:
 $\{w_0, w_1, w_2, \dots, w_m\}$
- w_i is i -the model parameter associated with i -the feature.
- $h_{\mathbf{w}}$ is a model with parameter vector \mathbf{w} .

Loss function

The model parameters \mathbf{w} are learnt such that the square of difference between the actual and the predicted values is minimized.

$$J(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^n (\hat{y}^{(i)} - y^{(i)})^2$$

$$J(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^n (\mathbf{w}^T \mathbf{x}^{(i)} - y^{(i)})^2$$

Optimization

1. Normal equation
2. Iterative optimization with gradient descent: full batch, mini-batch or stochastic.

Evaluation measure

1. Mean squared error
2. Root mean squared error

Implementing with sklearn

Normal equation

sklearn provides LinearRegression estimator for weight vector estimation via normal equation

```
1 from sklearn.linear_model import LinearRegression
```

As like other estimator object, it implements fit method that takes dataset as an input along with any other hyperparameters and returns estimated weight vector.

```
1 lin_reg = LinearRegression(normalize=True)
2 lin_reg.fit(X_train, y_train)
```

It's a good practice to scale or normalize features. We can set normalize flag to True for normalizing the input features. By default, normalize is False.

It also implements a couple of other methods:

- `predict`: Predicts label for a new examples based on the learnt model.
- `score`: Returns R^2 of the linear regression model.

Coefficient of determination R^2

$$R^2 = \left(1 - \frac{u}{v}\right)$$

where

- u is the **residual sum of squares** and is calculated as

$$u = (\mathbf{X}\mathbf{w} - \mathbf{y})^T(\mathbf{X}\mathbf{w} - \mathbf{y})$$

- and v is the **total sum of square**. Let, $\hat{\mathbf{y}}_{mean} = \frac{1}{n} (\mathbf{X}\mathbf{w})$, then v is calculated as

$$v = (\mathbf{y} - \hat{\mathbf{y}}_{mean})^T(\mathbf{y} - \hat{\mathbf{y}}_{mean})$$

$$R^2 = \left(1 - \frac{u}{v}\right)$$

- The **best possible score** is 1.0.
- The **score** can be **negative** (because the model can be arbitrarily worse).
- A **constant model** that always predicts the expected value of y , disregarding the input features, would get a **score** of 0.0.

Model inspection

The learnt weights can be obtained by accessing the following class variables of `LinearRegression` estimator:

- The `intercept` weight w_0 can be obtained via `intercept_` class variable.
- The `weights` can be obtained via `coef_` class variable.

```
1 lin_reg.intercept_, lin_reg.coef_
```

Computational Complexity

- The normal equation uses the following equation to obtain

$$\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

- This involves matrix inversion operation of feature matrix \mathbf{X} .
- The `LinearRegression` estimator uses SVD for this task and has the complexity of $O(m^2)$ where m is the number of features.

- This implies that if we double the number of features, the training computation grows roughly 4 times.
 - As the **number of features grows large**, the approach of **normal equation slows down significantly**.
 - These approaches are **linear w.r.t. the number of training examples** as long as the training set fits in the memory.
- The **inference process** is **linear** w.r.t. both the number of examples and number of features.

Weight vector estimation via SGD

- SGD is a **simple yet very efficient** approach of learning weight vectors in linear regression problems especially **in large scale problem settings**.
- SGD offers **provisions for tuning** the optimization process. However as a downside of this, we need to **set a few hyperparameters**.
- SGD is **sensitive** to **feature scaling**.

In `sklearn`, an estimator `SGDRegressor` implements a plain stochastic gradient descent learning routine which supports different loss functions and penalties to fit linear regression models.

- `SGDRegressor` is well suited for regression problems with a large number of training samples ($> 10,000$). For learning problems with small number of training examples, `sklearn` user guide recommends `Ridge` or `Lasso`.

Key functionalities of `SGDRegressor`

Loss function:

- Can be set via the `loss` parameter.
- `SGDRegressor` supports the following loss functions.
 - `loss= 'squared_error'`: Ordinary least squares,
 - `loss = 'huber'`: Huber loss for robust regression

- **Regularization**

SGD supports the following penalties:

- `Penalty = 'l2'` : L2 norm penalty on `coef_` . This is default setting.
- `penalty = 'l1'`: L1 norm penalty on `coef_`. This leads to sparse solutions.
- `penalty = 'elasticnet'`: Convex combination of L2 and L1;

$$(1 - l1_ratio) * L2 + l1_ratio * L1$$

Learning rate

The learning rate η can be either constant or gradually decaying. There are following options for learning rate schedule specification in SGD:

1. invscaling

For regression the default learning rate schedule is inverse scaling

`learning_rate = 'invscaling'`. The learning rate in t -th iteration or time step is calculated as

$$\eta^{(t)} = \frac{\eta_0}{t^{power_t}}$$

where, η_0 and `power_t` are hyperparameters chosen by the user.

2. Constant

For a constant learning rate use `learning_rate = 'constant'` and use η_0 to specify the learning rate.

3. Adaptive

For an adaptively decreasing learning rate, use `learning_rate = 'adaptive'` and use η_0 to specify the starting learning rate.

- When the `stopping criterion` is reached, the learning rate is divided by 5, and the training loop continues.
- The algorithm stops when the learning rate goes below 10^{-6} .

4. Optimal

Used as a default setting for classification problems. The learning rate η for t -th iteration is calculated as follows:

$$\eta^{(t)} = \frac{1}{\alpha(t_0 + t)}$$

Here

- α is a regularization rate.
- t is the time step (there are a total of `n_samples*n_iter` time steps)
- t_0 is determined based on a heuristic proposed by Léon Bottou such that the expected initial updates are comparable with the expected size of the weights (this assuming that the norm of the training samples is approx. 1).

Stoping creteria

SGDRegressor provides two stopping criteria to stop the algorithm when a given level of convergence is reached:

1. With `early_stopping = True`

- The input data is split into a training set and a validation set based on the `validation_fraction` parameter.
- The model is fitted on the training set, and the stopping criterion is based on the prediction score (using the `scoring` method) computed on the validation set.

2. With `early_stopping = False`

- The model is fitted on the entire input data and
- The stopping criterion is based on the objective function computed on the training data.

- In both cases, the criterion is evaluated once by epoch, and the algorithm stops when the criterion does not improve `n_iter_no_change` times in a row.
- The improvement is evaluated with absolute tolerance `tol`.
- The algorithm stops in any case after a maximum number of iteration `max_iter`

SGD variation: Average SGD

SGDRegressor supports averaged SGD (ASGD). Averaging can be enabled by setting `average = True`

- ASGD performs the same updates as the regular SGD, and sets `coef_` attribute to the average value of the coefficients across all updates.
 - SGD sets `coef_` attribute to the last value of the coefficients (i.e. the values of the last update)
- The same process is followed for the `intercept_` attribute.

When using ASGD the learning rate can be larger and even constant, leading to a speed up in training.

Model inspection

We obtain the weight vector from the trained model as follows:

- `coef_` variable stores weights assigned to the features.
- `intercept_`, as name suggests, stores the intercept term.

Complexity

The major advantage of SGD is its **efficiency**, which is basically **linear** in the number of training examples.

If \mathbf{X} is a matrix of size (n, m) training has a cost of $O(knp)$.

- where k is the **number of iterations (epochs)** and p is the **average number of non-zero attributes per sample**.

Recent theoretical results, however, show that the runtime to get some desired optimization accuracy does not increase as the training set size increases.

Polynomial regression

Polynomial regression

Polynomial regression = Polynomial transformation + Linear Regression

`PolynomialFeatures` transformer transforms an input data matrix into a new data matrix of a given degree.

Example:

```
1 from sklearn.preprocessing import PolynomialFeatures  
2 import numpy as np  
3 X = np.arange(6).reshape(3, 2)  
4 print ("Data matrix: \n", X)  
5 poly = PolynomialFeatures(degree=2)  
6 print ("\n\nAfter transformation: \n", poly.fit_transform(X))
```

Output:

```
1 Data matrix:  
2 [[0 1]  
3 [2 3]  
4 [4 5]]  
5  
6  
7 After transformation:  
8 [[ 1.  0.  1.  0.  0.  1.]  
9 [ 1.  2.  3.  4.  6.  9.]  
10 [ 1.  4.  5. 16. 20. 25.]]
```

In the above example, the features of \mathbf{X} have been transformed from $[x_1, x_2]$ to $[1, x_1, x_2, x_1^2, x_1x_2, x_2^2]$, and can now be used within any linear model.

In some cases it's not necessary to include higher powers of any single feature, but only the so-called **interaction features** that multiply together all most distinct features. These can be gotten from '**PolynomialFeatures**' with the setting `interaction_only = True`.

In this case, the features of \mathbf{X} would be transformed from $[x_1, x_2]$ to $[1, x_1, x_2, x_1x_2]$.

Ridge regression

Ridge regression

Ridge regression minimizes L_2 penalized sum of squared error.

Ridge Loss = Sum of squared error + regularization_rate * penalty

- We use 'Ridge' estimator for implementing ridge regression. It takes parameter 'alpha' which is the regularization rate.
- 'RidgeCV' estimator implements ridge regression with cross validation for regularization rate.

RidgeCV parameters:

1. `alphas` is the list of regularization rates to try.

- The regularization rate must be positive.
- Larger values indicate stronger regularization.

2. `cv` determines the cross-validation splitting strategy.

- `None`, to use the efficient Leave-One-Out cross-validation
- `integer`, to specify the number of folds.
- `cv splitter` specifies how to generate cross validation sets.
- An iterable yielding (train, test) splits as arrays of indices.

In case of a binary or multiclass problems, for 'cv=None' or 'cv=5' (i.e. integer), 'StratifiedKFold' cross validation strategy is used. In other cases, 'KFold' cross validation strategy is used.

Model inspection

'RidgeCV' provides an additional output apart from usual outputs like `coef_` and `intercept_`:

- `alphas` provides the estimated regularization parameter.

Lasso regression

Lasso regression

Lasso uses $L1$ norm of weight vector as a penalty in linear regression loss function.

'sklearn' provides two implementations for learning weight vector in Lasso.

- 'Lasso' uses coordinated descent algorithm.
- 'LassoLars' uses least angle regression algorithm. It is adaption of forward stepwise feature selection for solving Lasso regression.

Classification functions in sci-kit learn

Dr. Ashish Tendulkar

IIT Madras

Machine Learning Practice

- In this week, we will study **sklearn functionality** for implementing **classification** algorithms.
- We will cover sklearn APIs for
 - Specific classification algorithms for **least square classification**, **perceptron**, and **logistic regression classifier**.
 - with regularization
 - multiclass, multilabel and multi-output setting
 - Various classification metrics.

- Cross validation and **hyper parameter search** for classification works exactly like how it works in regression setting.
 - However there are a couple of CV strategies that are specific to classification

Part I: sklearn API for classification

There are broadly two types of APIs based on their functionality:

Generic

- SGD classifier

Specific

- Logistic regression
- Perceptron
- Ridge classifier (for LSC)
- K-nearest neighbours (KNNs)
- Support vector machines (SVMs)
- Naive Bayes

Uses gradient descent for opt

Need to specify loss function

Specialized solvers for opt

All sklearn estimators for classification implement a few common methods for **model training, prediction and evaluation.**

Model training

```
fit(X, y[, coef_init, intercept_init, ...])
```

Prediction

`predict(X)` predicts **class label** for samples

`decision_function(X)` predicts **confidence score** for samples.

Evaluation

```
score(X, y[, sample_weight])
```

Return the **mean accuracy** on the given test data and labels.

There are a few common **miscellaneous methods** as follows:

`get_params([deep])` gets parameter for this estimator.

`set_params(**params)` sets the parameters of this estimator.

`densify()` converts coefficient matrix to dense array format.

`sparsify()` converts coefficient matrix to sparse format.

Now let's study how to implement different classifiers
with sklearn APIs.

Let's start with implementation of least square classification (LSC) with RidgeClassifier API.

Ridge classifier

- RidgeClassifier is a classifier variant of the Ridge regressor.

Binary classification:

- classifier first converts binary targets to $\{-1, 1\}$ and then treats the problem as a regression task, optimizing the objective of regressor:
 - minimize a penalized residual sum of squares
 - $\min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \alpha \|\mathbf{w}\|_2^2$
 - sklearn provides different solvers for this optimization
 - sklearn uses α to denote regularization rate
 - predicted class corresponds to the sign of the regressor's prediction

Multiclass classification:

- treated as multi-output regression
- predicted class corresponds to the output with the highest value

How to train a least square classifier?

Step 1: Instantiate a **classification estimator** without passing any arguments to it. This creates a ridge classifier object.

```
1 from sklearn.linear_model import RidgeClassifier  
2 ridge_classifier = RidgeClassifier()
```

Step 2: Call **fit** method on **ridge classifier object** with **training feature matrix** and **label vector** as arguments.

Note: The model is fitted using **X_train** and **y_train**.

```
1 # Model training with feature matrix x_train and  
2 # label vector or matrix y_train  
3 ridge_classifier.fit(X_train, y_train)
```

How to set regularization rate in RidgeClassifier?

Set **alpha** to float value. The default value is 0.1.

```
1 from sklearn.linear_model import RidgeClassifier  
2 ridge_classifier = RidgeClassifier(alpha=0.001)
```

- alpha should be **positive**.
- Larger alpha values specify **stronger regularization**.

How to solve optimization problem in RidgeClassifier?

Using one of the following solvers

svd

uses a Singular Value Decomposition of the feature matrix to compute the Ridge coefficients.

cholesky

uses `scipy.linalg.solve` function to obtain the closed-form solution

sparse_cg

uses the conjugate gradient solver of `scipy.sparse.linalg.cg`.

lsqr

uses the dedicated regularized least-squares routine `scipy.sparse.linalg.lsqr` and it is fastest.

sag , saga

uses a Stochastic Average Gradient descent iterative procedure
'saga' is unbiased and more flexible version of 'sag'

lbfgs

uses L-BFGS-B algorithm implemented in `scipy.optimize.minimize`.

can be used only when coefficients are forced to be positive.

Uses of `solver` in RidgeClassifier

- For large scale data, use '`sparse_cg`' solver.
- When both `n_samples` and `n_features` are large, use '`sag`' or '`saga`' solvers.
 - Note that fast convergence is only guaranteed on features with approximately the same scale.

How to make RidgeClassifier select the solver automatically?

```
1 ridge_classifier = RidgeClassifier(solver=auto)
```

auto

chooses the solver automatically based on
the type of data

```
1 if solver == 'auto':
2     if return_intercept:
3         # only sag supports fitting intercept directly
4         solver = "sag"
5     elif not sparse.issparse(X):
6         solver = "cholesky"
7     else:
8         solver = "sparse_cg"
```

Default choice for solver is **auto** .

Is `intercept` estimation necessary for RidgeClassifier?

If data is already centered, set `fit_intercept` as false, so that no intercept will be used in calculations.

Default:

```
1 ridge_classifier = RidgeClassifier(fit_intercept=True)
```

How to make **predictions** on new data samples?

Use **predict** method to predict class labels for samples

Step 1: Arrange data for prediction in a feature matrix of shape (#samples, #features) or in sparse matrix format.

Step 2: Call **predict** method on **classifier object** with **feature matrix** as an argument.

```
1 # Predict labels for feature matrix x_test  
2 y_pred = ridge_classifier.predict(x_test)
```

Other classifiers also use the same **predict** method.

`RidgeClassifierCV` implements
`RidgeClassifier` with built-in cross validation.

Let's implement **perceptron classifier** with
Perceptron API.

Perceptron classification

- It is a simple classification algorithm suitable for large-scale learning.
- Shares the same underlying implementation with `SGDClassifier`

`Perceptron()`



```
SGDClassifier(loss="perceptron", eta0=1,  
learning_rate="constant", penalty=None)
```

Perceptron uses SGD for training.

How to implement perceptron classifier?

Step 1: Instantiate a **Perceptron** estimator without passing any arguments to it to create a classifier object.

```
1 from sklearn.linear_model import Perceptron  
2 perceptron_classifier = Perceptron()
```

Step 2: Call **fit** method on **perceptron estimator** object with **training feature matrix** and **label vector** as arguments.

```
1 # Model training with feature matrix x_train and  
2 # label vector or matrix y_train  
3 perceptron_classifier.fit(x_train, y_train)
```

Perceptron can be further customized with the following parameters:

`penalty`

(default = 'l2')

`l1_ratio`

(default = 0.15)

`alpha`

(default = 0.0001)

`early_stopping`

(default = False)

`fit_intercept`

(default = True)

`max_iter`

(default = 1000)

`n_iter_no_change`

(default = 5)

`tol`

(default = 1e-3)

`eta0`

(default = 1)

`validation_fraction`

(default = 0.1)

- Perceptron classifier can be trained in an **iterative manner** with `partial_fit` method
- Perceptron classifier can be initialized to the weights of the previous run by specifying `warm_start = True` in the constructor.

Let's implement logistic regression classifier
with [LogisticRegression API](#).

LogisticRegression API

- Implements logistic regression classifier, which is also known by a few different names like logit regression, maximum entropy classifier (`maxent`) and log-linear classifier.

$$\arg \min_{\mathbf{w}, C} \text{regularization penalty} + C \text{ cross entropy loss}$$

- This implementation can fit
 - binary classification
 - one-vs-rest (OVR)
 - multinomial logistic regression
- Provision for ℓ_1 , ℓ_2 or elastic-net regularization

How to train a LogisticRegression classifier?

Step 1: Instantiate a **classifier estimator** without passing any arguments to it. This creates a logistic regression object.

```
1 from sklearn.linear_model import LogisticRegression  
2 logit_classifier = LogisticRegression()
```

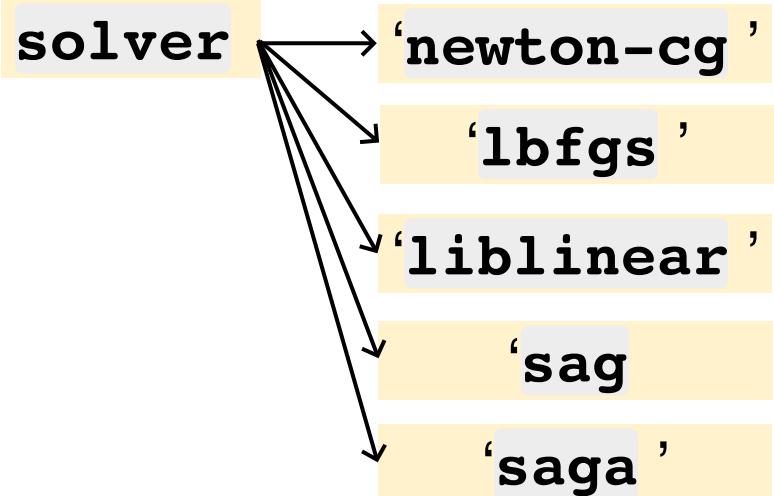
Step 2: Call **fit** method on **logistic regression classifier object** with **training feature matrix** and **label vector** as arguments

```
1 # Model training with feature matrix x_train and  
2 # label vector or matrix y_train  
3 logit_classifier.fit(x_train, y_train)
```

Logistic regression uses **specific algorithms** for solving the optimization problem in training. These algorithms are known as **solvers**.

The **choice** of the solver depends on the **classification** problem set up such as **size of the dataset**, **number of features** and **labels**.

How to select solvers for Logistic Regression classifier?



- For small datasets, ‘liblinear’ is a good choice, whereas ‘sag’ and ‘saga’ are faster for large ones.

- For unscaled datasets, ‘liblinear’, ‘lbfgs’ and ‘newton-cg’ are robust.
- For multiclass problems, only ‘newton-cg’, ‘sag’, ‘saga’ and ‘lbfgs’ handle multinomial loss.
- ‘liblinear’ is limited to one-versus-rest schemes

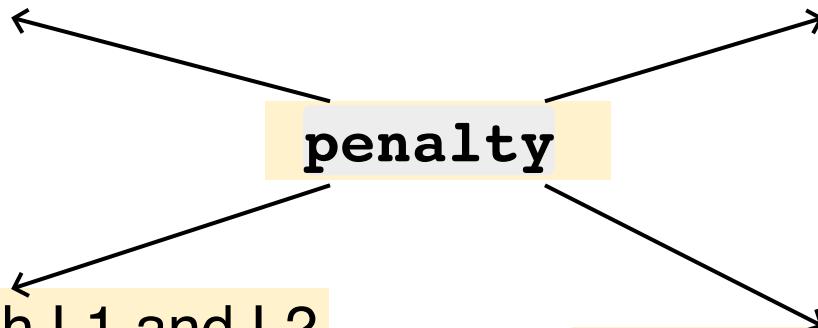
By default, logistic regression uses `lbfgs` solver.

```
1 logit_classifier = LogisticRegression(solver='lbfgs')
```

How to add regularization in Logistic Regression classifier?

- *l2* - adds a L2 penalty term

- *l1* - adds a L1 penalty term



- *elasticnet* - both L1 and L2 penalty terms are added

- *none* - no penalty is added

Regularization is applied by default because it improves numerical stability.

By default, it uses *L2* penalty.

```
1 logit_classifier = LogisticRegression(penalty='l2')
```

- Not all the solvers supports all the penalties.
- Select appropriate solver for the desired penalty.
 - L2 penalty is supported by all solvers
 - L1 penalty is supported only by a few solvers.

Solver	Penalty
‘ newton-cg ’	[‘l2’, ‘none’]
‘ lbfgs ’	[‘l2’, ‘none’]
‘ liblinear ’	[‘l1’, ‘l2’]
‘ sag ’	[‘l2’, ‘none’]
‘ saga ’	[‘elasticnet’, ‘l1’, ‘l2’, ‘none’]

How to control amount of regularization in logistic regression?

- sklearn implementation uses parameter **C**, which is **inverse of regularization rate** to control regularization.
- Recall

$$\arg \min_{\mathbf{w}, C} \text{regularization penalty} + C \text{ cross entropy loss}$$

- C is specified in the constructor and must be positive
 - Smaller value leads to **stronger** regularization.
 - Larger value leads to **weaker** regularization.

LogisticRegression classifier has a `class_weight` parameter in its constructor.

What purpose does it serve?

- Handles `class imbalance` with `differential class weights`.
- Mistakes in a class are `penalized by the class weight`.
 - `Higher value here would mean higher emphasis` on the class.

This parameter is available in classifier estimators in sklearn.

Exercise: Read [stack overflow discussion](#) on this parameter.

`LogisticRegressionCV` implements logistic regression with in built cross validation support to find the best values of `C` and `l1_ratio` parameters according to the specified scoring attribute.

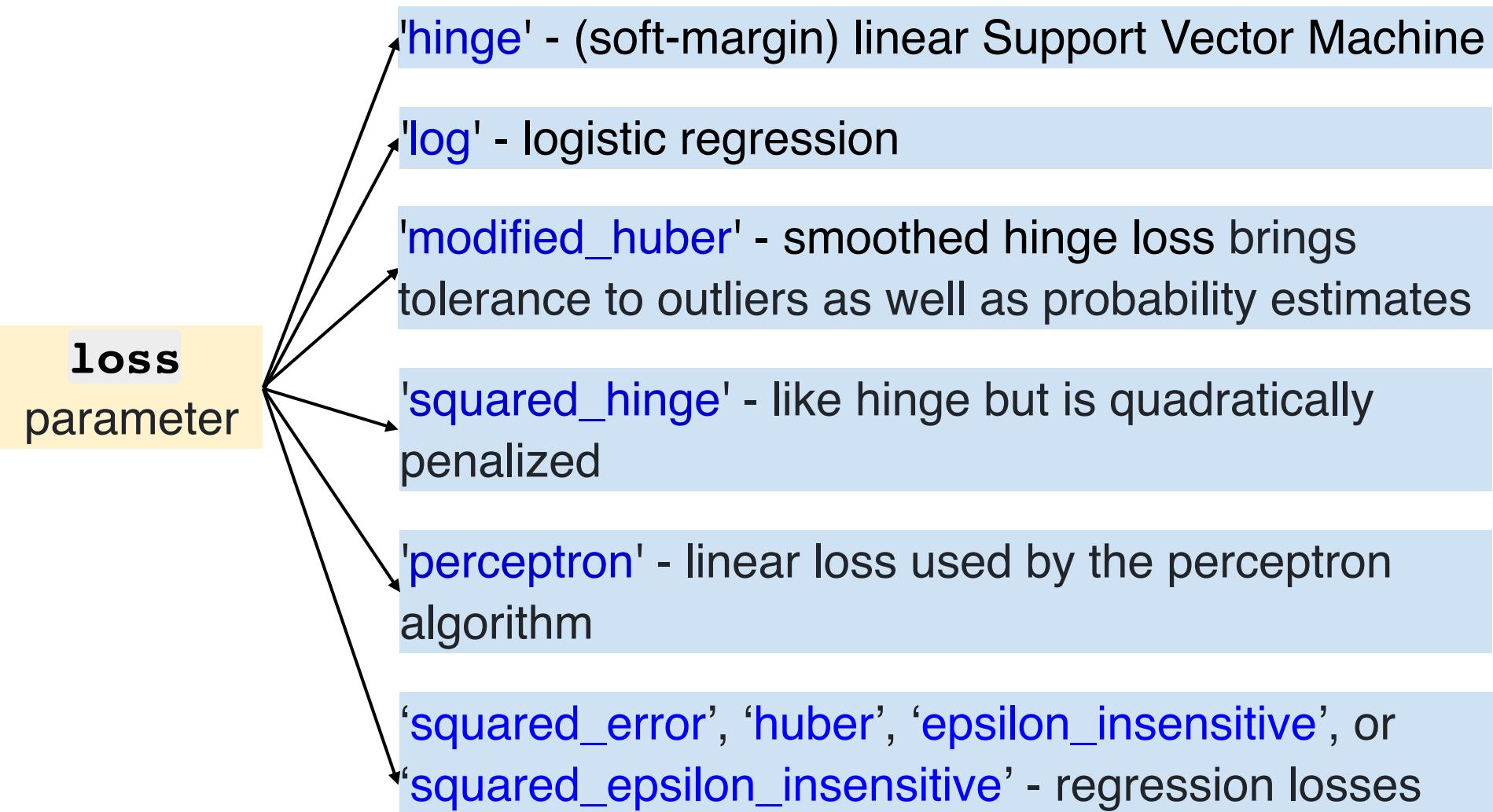
These classifiers can also be implemented with a generic `SGDClassifier` API by setting the `loss parameter` appropriately.

Let's study [SGDClassifier](#) API.

SGDClassifier

- SGD is a simple yet very efficient approach to fitting linear classifiers under convex loss functions
- This API uses SGD as an optimization technique and can be applied to build a variety of linear classifiers by adjusting the loss parameter.
- It supports multi-class classification by combining multiple binary classifiers in a “one versus all” (OVA) scheme.
- Easily scales up to large scale problems with more than 10^5 training examples and 10^5 features. It also works with sparse machine learning problems
 - Text classification and natural language processing

We need to set **loss parameter** appropriately to build train classifier of our interest with **SGDClassifier**



By default **SGDClassifier** uses **hinge loss** and hence trains **linear support vector machine classifier**.

- An instance of `SGDClassifier` might have an equivalent estimator in the scikit-learn API.

```
SGDClassifier(loss='log')
```



```
LogisticRegression(solver='sgd')
```

```
SGDClassifier(loss='hinge')
```



```
Linear Support vector machine
```

How does SGDClassifier work?

- SGDClassifier implements a plain stochastic gradient descent learning routine.
 - the gradient of the loss is estimated with one sample at a time and the model is updated along the way with a decreasing learning rate (or strength) schedule.

Advantages:

- Efficiency.
- Ease of implementation

Disadvantages:

- Requires a number of hyperparameters.
- Sensitive to feature scaling.

- It is important
 - to permute (shuffle) the training data before fitting the model.
 - to standardize the features.

How to use **SGDClassifier** for training a classifier?

Step 1: Instantiate a **SGDClassifier** estimator by setting appropriate loss parameter to define classifier of interest. By default it uses **hinge loss**, which is used for training linear support vector machine.

```
1 from sklearn.linear_model import SGDClassifier  
2 SGD_classifier = SGDClassifier(loss='log')
```

Here we have used `log` loss that defines a logistic regression classifier.

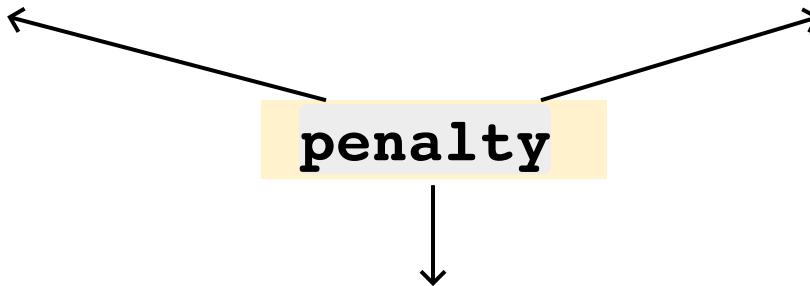
Step 2: Call **fit** method on **SGD classifier object** with **training feature matrix** and **label vector** as arguments.

```
1 # Model training with feature matrix x_train and  
2 # label vector or matrix y_train  
3 SGD_classifier.fit(x_train, y_train)
```

How to perform regularization in SGD classifier?

- *l2* - adds a L2 penalty term

- *l1* - adds a L1 penalty term



- **elasticnet** - Convex combination of L2 and L1

$$(1 - l1_ratio) * L2 + l1_ratio * L1$$

`(l1_ratio` controls the convex combination of L1 and L2 penalty. default=0.15)

Default:

```
1 SGD_classifier = SGDClassifier(penalty='l2')
```

alpha

|

- Constant that multiplies the regularization term.
- Has float values and **default = 0.0001**

How to set **maximum number of epochs** for SGD Classifier?

The maximum number of passes over the training data (aka epochs) is an integer that can be set by the `max_iter` parameter.

```
1 SGD_classifier = SGDClassifier(max_iter=100)
```

Default:

`max_iter = 1000`

Some common parameters between SGDClassifier and SGDRegressor

learning_rate

- ‘constant’
- ‘optimal’
- ‘invscaling’
- ‘adaptive’

warm_start

- ‘True’
- ‘False’

average

- SGDClassifier also supports averaged SGD (ASGD)

Stopping criteria

tol

n_iter_no_change

max_iter

early_stopping

validation_fraction

Summary

We learnt how to implement the following classifiers with sklearn APIs:

- Least square classification ([RidgeClassifier](#))
- Perceptron ([Perceptron](#))
- Logistic regression ([LogisticRegression](#))

Alternatively we can use [SGDClassifier](#) with appropriate [loss](#) setting for implementing these classifiers:

- `loss = 'log'` for [logistic regression](#)
- `loss = 'perceptron'` for [perceptron](#)
- `loss = 'squared_error'` for [least square classification](#)

Classification estimators implements a few common methods like [fit](#), [score](#), [decision_function](#), and [predict](#).

- These estimators can be readily used in **multiclass setting**.
- They support **regularized loss function** optimization.
- All classification estimators have ability to deal with **class imbalance** through **class_weight** parameter in the constructor.

Part II: Multi-learning classification set up

Let's extend these classifiers to multi-learning (multi-class, multi-label & multi-output) settings.

Basics of multiclass, multilabel and multioutput classification

- **Multiclass classification** has **exactly one output label** and the total **number of labels > 2**.
- For **more than one output**, there are **two types** of classification models:

Multilabel

total #labels = 2

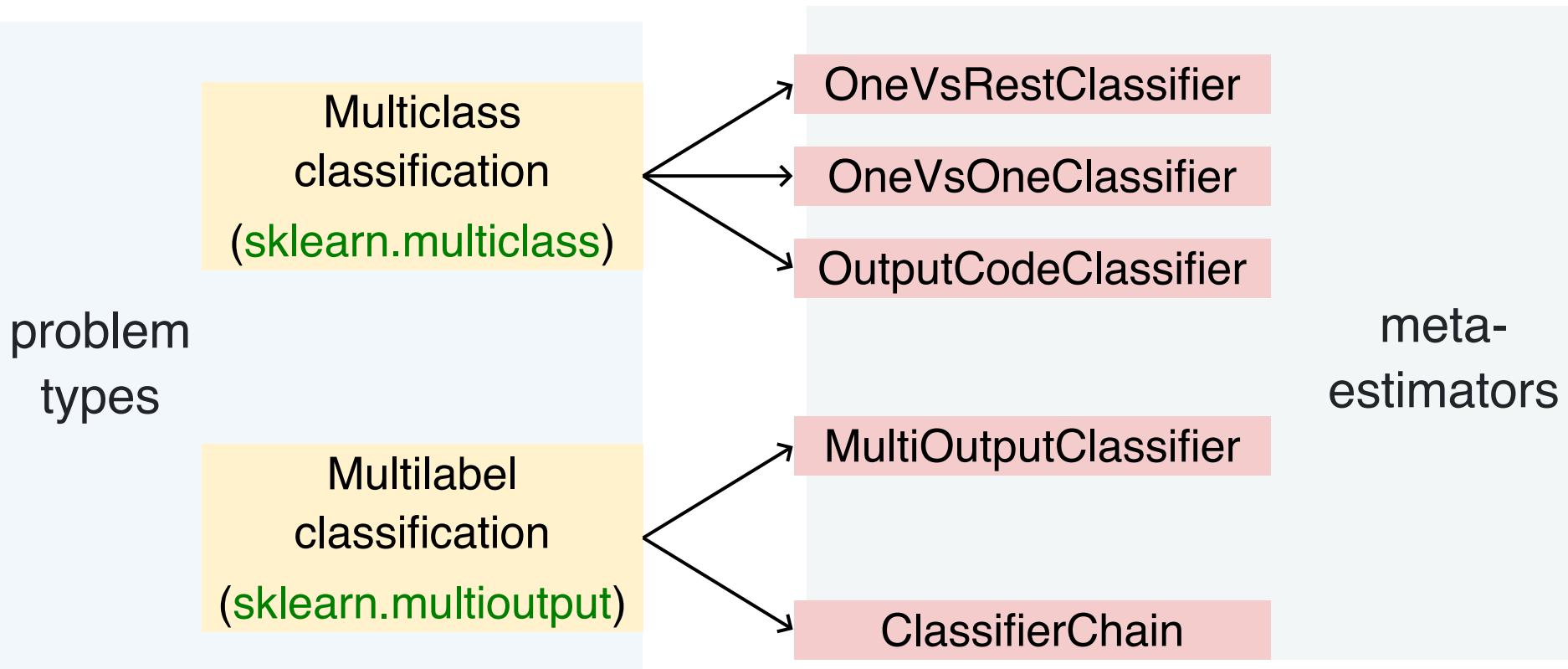
Multiclass multioutput

total #labels > 2

We will refer both these models as **multi-label classification** models, where **# of output labels > 1**.

Multiclass, multilabel, multioutput problems are referred to as **multi-learning problems**.

- sklearn provides a bunch of **meta-estimators**, which **extend** the functionality of **base estimators** to support multi-learning problems.
- The meta-estimators **transform** the multi-learning problem into **a set of simpler problems** and **fit one estimator per problem**.



- Many sklearn estimators have **built-in support** for multi-learning problems.
 - Meta-estimators are not needed for such estimators, however meta-estimators can be used in case we want to use these base estimators with **strategies** beyond the built-in ones.

Inherently
multiclass

Multiclass as
OVO

Multiclass as
OVR

Multilabel

Inherently
multiclass

LogisticRegression (multi_class = 'multinomial')
LogisticRegressionCV (multi_class = 'multinomial')
RidgeClassifier
RidgeClassifierCV

Multiclass as
OVR

LogisticRegression (multi_class = 'ovr')
LogisticRegressionCV (multi_class = 'ovr')
SGDClassifier
Perceptron

Multilabel

RidgeClassifier
RidgeClassifierCV

First we will study **multiclass APIs** in sklearn.

Multi-class classification

- Classification task with **more than two classes**.
- **Each example** is labeled with **exactly one class**

In **Iris dataset**,

- There are three class labels: **setosa**, **versicolor** and **virginica**.
- Each example has **exactly one label** of the three available class labels.
- Thus, this is an instance of a **multi-class classification**.

In **MNIST digit recognition dataset**,

- There are 10 class labels: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Each example has **exactly one label** of the 10 available class labels.
- Thus, this is an instance of a **multi-class classification**.

How to represent class labels in multi-class setup?

- Each example is marked with a single label out of k labels. The shape of label vector is $(n, 1)$.
- Use [LabelBinarizer](#) transformation to convert the class label to multi-class format.

```
1 from sklearn.preprocessing import LabelBinarizer  
2 y = np.array(['apple', 'pear', 'apple', 'orange'])  
3 y_dense = LabelBinarizer().fit_transform(y)
```

- The resulting label vector has shape of (n, k) .

```
[ [ 1  0  0 ]  
  [ 0  0  1 ]  
  [ 1  0  0 ]  
  [ 0  1  0 ] ]
```

Let's say, you are given labels as part of the training set, how do we check if they are suitable for multi-class classification?

- Use `type_of_target` to determine the type of the label.

```
1 from sklearn.utils.multiclass import type_of_target  
2 type_of_target(y)
```

- In case, y is a vector with more than two discrete values, `type_of_target` returns `multiclass`.

`type_of_target` can determine different types
of multi-learning targets.

target_type

‘multiclass’

‘multiclass-
multioutput’

‘multilabel-
indicator’

‘unknown’

y

- contains more than two discrete values
- not a sequence of sequences
- 1d or a column vector

- 2d array that contains more than two discrete values
- not a sequence of sequences
- dimensions are of size > 1

- label indicator matrix
- an array of two dimensions with at least two columns, and at most 2 unique values.

- array-like but none of the above, such as a 3d array,
- sequence of sequences, or an array of non-sequence objects.

Examples

multiclass

```
1 >>> type_of_target([1, 0, 2])
2 'multiclass'
3 >>> type_of_target([1.0, 0.0, 3.0])
4 'multiclass'
5 >>> type_of_target(['a', 'b', 'c'])
6 'multiclass'
```

multiclass-multioutput

```
1 >>> type_of_target(np.array([[1, 2], [3, 1]]))
2 'multiclass-multioutput'
```

multilabel-indicator

```
1 type_of_target(np.array([[0, 1], [1, 1]]))
2 'multilabel-indicator'
3 >>> type_of_target([[1, 2]])
4 'multilabel-indicator'
```

Apart from these, there are three more types, `type_of_target` can determine targets corresponding to `regression` and `binary classification`.

- `continuous` - regression target
- `continuous-multioutput` - multi-output target
- `binary` - classification

All classifiers in scikit-learn perform multiclass classification out-of-the-box.

- Use `sklearn.multiclass` module only when you want to experiment with different multiclass strategies.
- Using different multi-class strategy than the one implemented by default may affect performance of classifier in terms of either generalization error or computational resource requirement.

What are different multi-class classification strategies implemented in sklearn?

- One-vs-all or one-vs-rest (OVR)
 - One-vs-One (OVA)
-
- OVR is implemented by `OneVsRestClassifier` API.
 - OVA is implemented by `OneVsOneClassifier` API.

OVR - OneVsRestClassifier

- Fits one classifier per class c - c vs not c .
- This approach is computationally efficient and requires only k classifiers.
- The resulting model is interpretable.

```
1 from sklearn.multiclass import OneVsRestClassifier  
2 OneVsRestClassifier(LinearSVC(random_state=0)).fit(X, y)
```

- We need to supply estimator as an argument in the constructor.
- Support methods like other classifiers - fit, predict, predict_proba, partial_fit.

OneVsRest classifier also supports multilabel classification.
We need to supply labels as indicator matrix of shape (n, k) .

OVA - OneVsOneClassifier

- Fits one classifier per pair of classes. Total classifiers = $\binom{k}{2}$.
- Predicts class that receives maximum votes.
 - The tie among classes is broken by selecting the class with the highest aggregate classification confidence.

```
1 from sklearn.multiclass import OneVsOneClassifier  
2 OneVsOneClassifier(LinearSVC(random_state=0)).fit(X, y)
```

- We need to supply estimator as an argument in the constructor.
- Support methods like other classifiers - fit, predict, predict_proba, partial_fit.

OneVsOne classifier processes subset of data at a time and is useful in cases where the classifier does not scale with the data.

What is the difference between OVR and OVA?

OneVsRestClassifier

- Fits one classifier per class.
- For each classifier, the class is fitted against all the other classes.

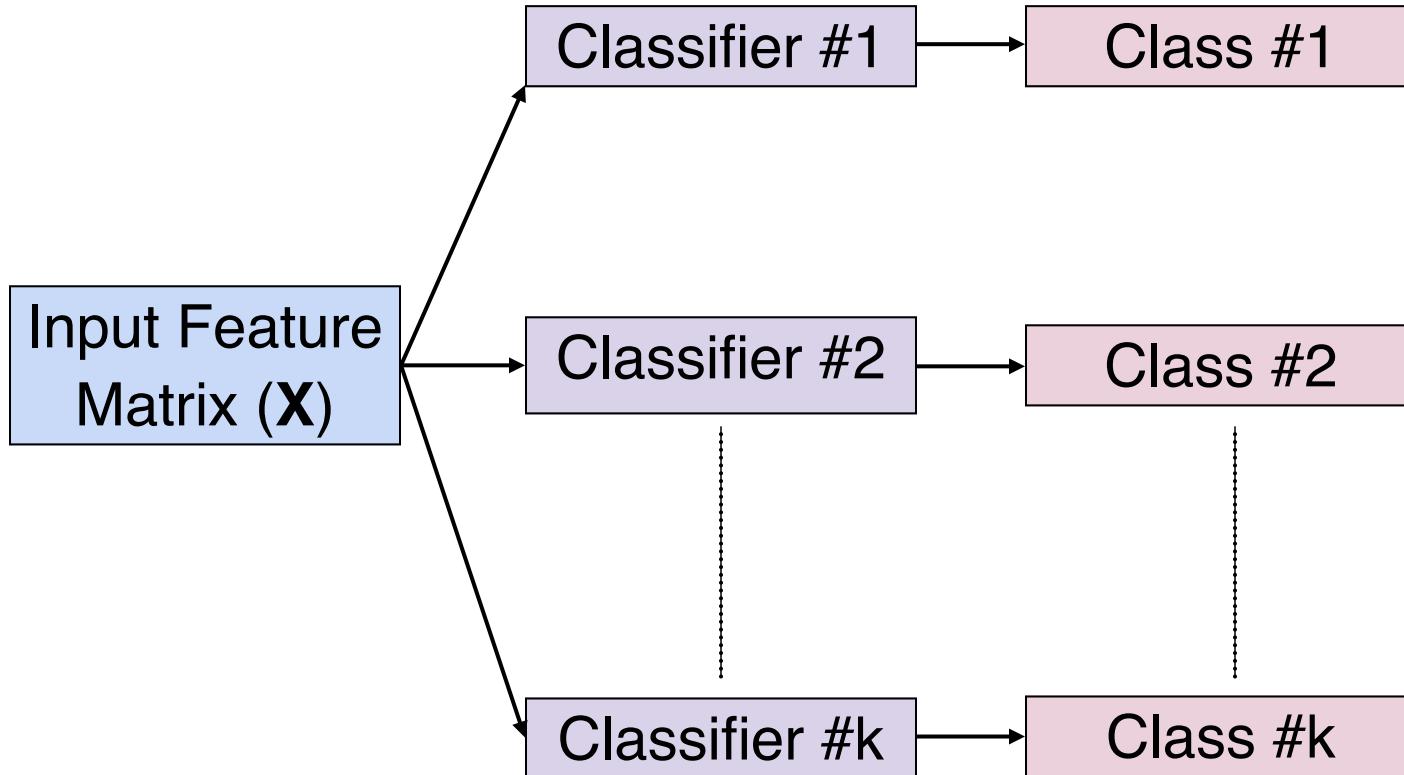
OneVsOneClassifier

- Fits one classifier per pair of classes.
- At prediction time, the class which received the most votes is selected.

Now we will learn how to perform **multilabel**
and **multi-output** classification.

How MultiOutputClassifier works?

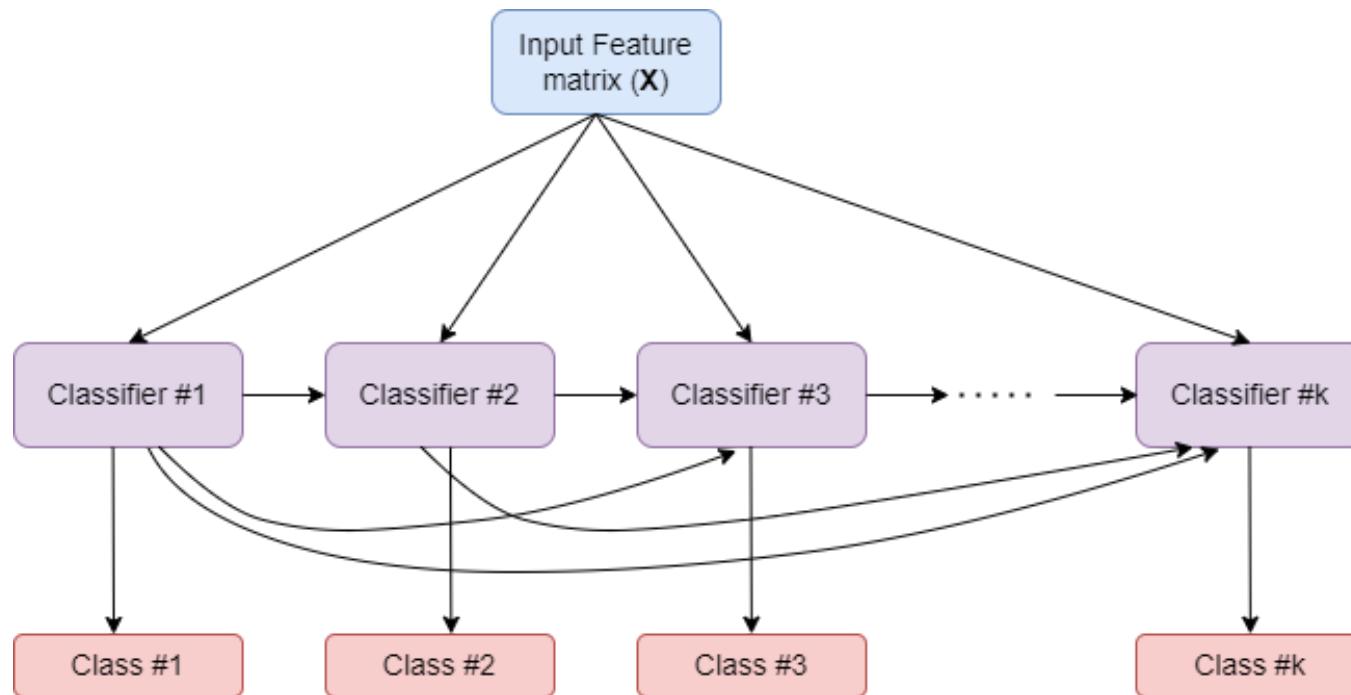
- Strategy consists of fitting one classifier per target.



- Allows multiple target variable classifications.

How ClassifierChain works?

- A multi-label model that arranges binary classifiers into a chain.
- Way of combining a number of binary classifiers into a single multi-label model.



Comparison of **MultiOutputClassifier** and **ClassifierChain**

MultiOutputClassifier

- Able to estimate a **series of target functions** that are trained on a **single predictor matrix** to predict a **series of responses**.
- Allows **multiple target variable classifications**.

ClassifierChain

- Capable of **exploiting correlations** among targets.
- For a multi-label classification problem with k classes, k binary classifiers are assigned an integer between 0 and $k - 1$.
- These integers define the order of models in the chain.

Summary

- Different types of multi-learning setups: **multi-class**, **multi-label**, **multi-output**.
- **type_of_target** to determine the nature of supplied labels.
- Meta-estimators:
 - **multi-class**: **One-vs-rest**, **one-vs-one**
 - **multi-label**: **Classifier chain** and **multi-output classifier**

Evaluating Classifiers

So far we learnt how to **train** classifiers for **binary**, **multi-class** and **multi-label/output** cases.

We will learn how to evaluate these classifiers with **different scoring functions** and with **cross-validation**.

We will also study how to set **hyper-parameters** for classifiers.

Many cross-validation and HPT methods discussed in the regression context are also applicable in classifiers.

- We will not repeat that discussion in this topic.
- Instead we will focus on only additional methods that are specific to classifiers.

Stratified cross validation iterators

There may be issues like **class imbalance** in classification, which tend to impact the cross validation folds.

The **overall class distribution** and the ones in **folds** may be **different** and this has implications in effective model training.

`sklearn.model_selection` module provides three **stratified APIs** to create folds such that the **overall class distribution** is replicated in individual folds.

`sklearn.model_selection` module provides the following three **stratified APIs** to create folds such that the **overall class distribution is replicated in individual folds**.

- `StratifiedKFold`
- `RepeatedStratifiedKFold`
- `StratifiedShuffleSplit`

Note: Folds obtained via `StratifiedShuffleSplit` may not be completely different.

LogisticRegressionCV

- Support in-build cross validation for optimizing hyperparameters
- The following are key parameters for HPT and cross validation

`cv` specifies
cross validation
iterator

`scoring` specifies
scoring function to
use for HPT

`cs` specifies
regularization
strengths to
experiment with.

- Choosing the best hyper-parameters

`refit = True`

Scores averaged across folds, values
corresponding to the best score are selected
and final refit with these parameters

`refit = False`

the `coefs`, `intercepts` and `C` that correspond
to the best scores across folds are averaged.

Now let's look at classification metrics
implemented in sklearn.

Classification metrics

`sklearn.metrics` implements a bunch of `classification scoring metrics` based on `true labels` and `predicted labels` as inputs.

`accuracy_score`

`balanced_accuracy_score`

`top_k_accuracy_score`

`roc_auc_score`

`precision_score`

`recall_score`

`f1_score`

`score(actual_labels, predicted_labels)`

Confusion matrix

- `confusion_matrix` evaluates classification accuracy by computing the confusion matrix with each row corresponding to the true class.

```
1 from sklearn.metrics import confusion_matrix  
2 confusion_matrix(y_true, y_predicted)
```

Example:

```
array([[2, 0, 0],  
       [0, 0, 1],  
       [1, 0, 2]])
```

Entry i, j in a confusion matrix

number of observations actually in group i ,
but predicted to be in group j .

Confusion matrix can be displayed with [ConfusionMatrixDisplay](#) API in [sklearn.metrics](#).

- Confusion matrix

```
1 ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
```

- From estimators

```
1 ConfusionMatrixDisplay.from_estimator(clf, X_test, y_test)
```

- From predictions

```
1 ConfusionMatrixDisplay.from_predictions(y_test, y_pred)
```

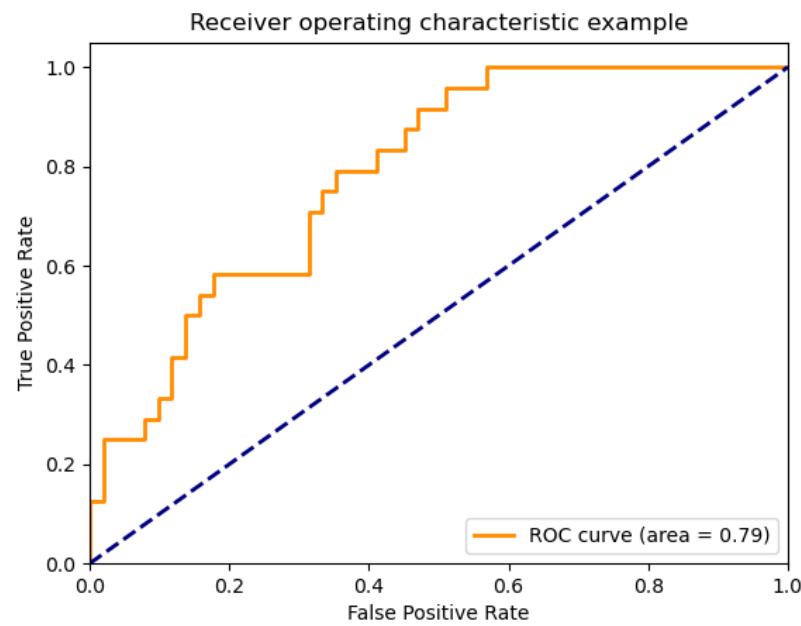
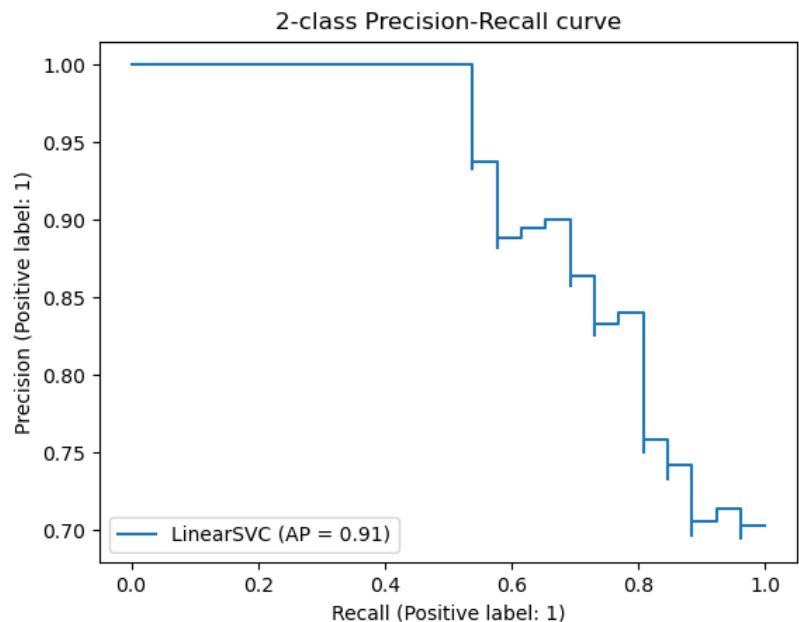
The `classification_report` function builds a text report showing the main classification metrics.

```
1 from sklearn.metrics import classification_report  
2 print(classification_report(y_true, y_predicted))
```

	precision	recall	f1-score	support
class 0	0.67	1.00	0.80	2
class 1	0.00	0.00	0.00	1
class 2	1.00	0.50	0.67	2
accuracy			0.60	5
macro avg	0.56	0.50	0.49	5
weighted avg	0.67	0.60	0.59	5

Classifier Performance across probability thresholds

```
1 from sklearn.metrics import precision_recall_curve  
2 precision, recall, thresholds = precision_recall_curve(y_true, y_predicted)
```



```
1 from sklearn.metrics import roc_curve  
2 fpr, tpr, thresholds = metrics.roc_curve(y_true, y_scores, pos_label=2)
```

How to extend binary metric to multiclass or multilabel problems?

- Treat data as a collection of binary problems, one for each class.
- Then, average binary metric calculations across the set of classes.
 - Can be done using `average` parameter.

`macro`

calculates the mean of the binary metrics

`weighted`

computes the average of binary metrics in which each class's score is weighted by its presence in the true data sample.

`micro`

gives each sample-class pair an equal contribution to the overall metric

`samples`

calculates the metric over the true and predicted classes for each sample in the evaluation data, and returns their average

`None`

returns an array with the score for each class

Summary

- Classification specific cross validation iterator based on stratification.
- Classification metrics
- Extending binary metrics to multi-learning set up.

Naive Bayes in sci-kit learn

Dr. Ashish Tendulkar

IIT Madras

Machine Learning Practice

Naive Bayes Classifier

Naive Bayes classifier

- Naive Bayes classifier applies **Bayes' theorem** with the “naive” assumption of conditional independence between every pair of features given the value of the class variable.

For a given class variable y and dependent feature vector x_1 through x_m ,

the naive conditional independence assumption is given by:

$$P(x_i|y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_m) = P(x_i|y)$$

Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods.

List of NB Classifiers

- Implemented in `sklearn.naive_bayes` module

GaussianNB

BernoulliNB

CategoricalNB

MultinomialNB

ComplementNB

- Implements `fit` method to estimate parameters of NB classifier with `feature matrix` and `labels` as inputs.
- The prediction is performed using `predict` method.

Which NB to use if data is only numerical?

GaussianNB

implements the Gaussian Naive Bayes algorithm for classification

Instantiate a **GaussianNBClassifier** estimator and then call fit method using X_train and y_train.

```
1 from sklearn.naive_bayes import GaussianNB  
2 gnb = GaussianNB()  
3 gnb.fit(X_train, y_train)
```

Which NB to use if data is multinomially distributed?

MultinomialNB

implements the naive Bayes algorithm for
multinomially distributed data
(text classification)

Instantiate a [MultinomialNBClassifier](#) estimator and then call fit method using X_train and y_train.

```
1 from sklearn.naive_bayes import MultinomialNB  
2 mnb = MultinomialNB()  
3 mnb.fit(X_train, y_train)
```

What to do if data is imbalanced ?

ComplementNB

implements the complement naive Bayes (CNB) algorithm.

Instantiate a [ComplementNBClassifier](#) estimator and then call fit method using X_train and y_train.

```
1 from sklearn.naive_bayes import ComplementNB  
2 cnb = ComplementNB()  
3 cnb.fit(X_train, y_train)
```

CNB regularly outperforms MNB (often by a considerable margin) on text classification tasks.

What to do if data has multivariate Bernoulli distributions?

BernoulliNB

- implements the naive Bayes algorithm for data that is distributed according to multivariate Bernoulli distributions
- each feature is assumed to be a binary-valued (Bernoulli, boolean) variable

Instantiate a `BernoulliNBClassifier` estimator and then call fit method using `X_train` and `y_train`.

```
1 from sklearn.naive_bayes import BernoulliNB  
2 bnb = BernoulliNB()  
3 bnb.fit(X_train, y_train)
```

What to do if data is categorical ?

CategoricalNB

implements the categorical naive Bayes algorithm suitable for classification with discrete features that are categorically distributed

assumes that each feature, which is described by the index i , has its own categorical distribution.

Instantiate a CategoricalNBClassifier estimator and then call fit method using X_train and y_train.

```
1 from sklearn.naive_bayes import CategoricalNB  
2 canb = CategoricalNB()  
3 canb.fit(X_train, y_train)
```

K Nearest Neighbours

Dr. Ashish Tendulkar

IIT Madras

Machine Learning Practice

Nearest neighbor classifier

- It is a type of **instance-based** learning or **non-generalizing** learning
 - does not attempt to **construct** a model
 - simply **stores instances** of the training data
- Classification is computed from a simple **majority vote** of the **nearest neighbors** of each point.
- Two different implementations of nearest neighbors classifiers are available.
 1. KNeighborsClassifier
 2. RadiusNeighborsClassifier

How are KNeighborsClassifier and RadiusNeighborsClassifier different?

KNeighborsClassifier

- learning based on the k nearest neighbors
- most commonly used technique
- choice of the value k is highly data-dependent

RadiusNeighborsClassifier

- learning based on the number of neighbors within a fixed radius r of each training point
- used in cases where the data is not uniformly sampled
- fixed value of r is specified, such that points in sparser neighborhoods use fewer nearest neighbors for the classification

How do you apply KNeighborsClassifier?

Step 1: Instantiate a `KNeighborsClassifier` estimator without passing any arguments to it to create a classifier object.

```
1 from sklearn.neighbors import KNeighborsClassifier  
2 kneighbor_classifier = KNeighborsClassifier()
```

Step 2: Call `fit` method on `KNeighbors classifier` object with `training feature matrix` and `label vector` as arguments.

```
1 # Model training with feature matrix X_train and  
2 # label vector or matrix y_train  
3 kneighbor_classifier.fit(X_train, y_train)
```

How do you specify the number of nearest neighbors in `KNeighborsClassifier`?

- Specify the number of nearest neighbors K from the training dataset using `n_neighbors` parameter.
 - value should be `int`.

```
1 kneighbor_classifier = KNeighborsClassifier(n_neighbors = 3)
```

What is the default value of K ?

`n_neighbors = 5`

How do you assign weights to neighborhood in KNeighborsClassifier?

- It is better to weight the neighbors such that nearer neighbors contribute more to the fit.

weights

- ‘uniform’ : All points in each neighborhood are weighted equally.
- ‘distance’ : weight points by the inverse of their distance.
 - closer neighbors of a query point will have a greater influence than neighbors which are further away.

Default:

```
1 kneighbor_classifier = KNeighborsClassifier(weights= 'uniform')
```

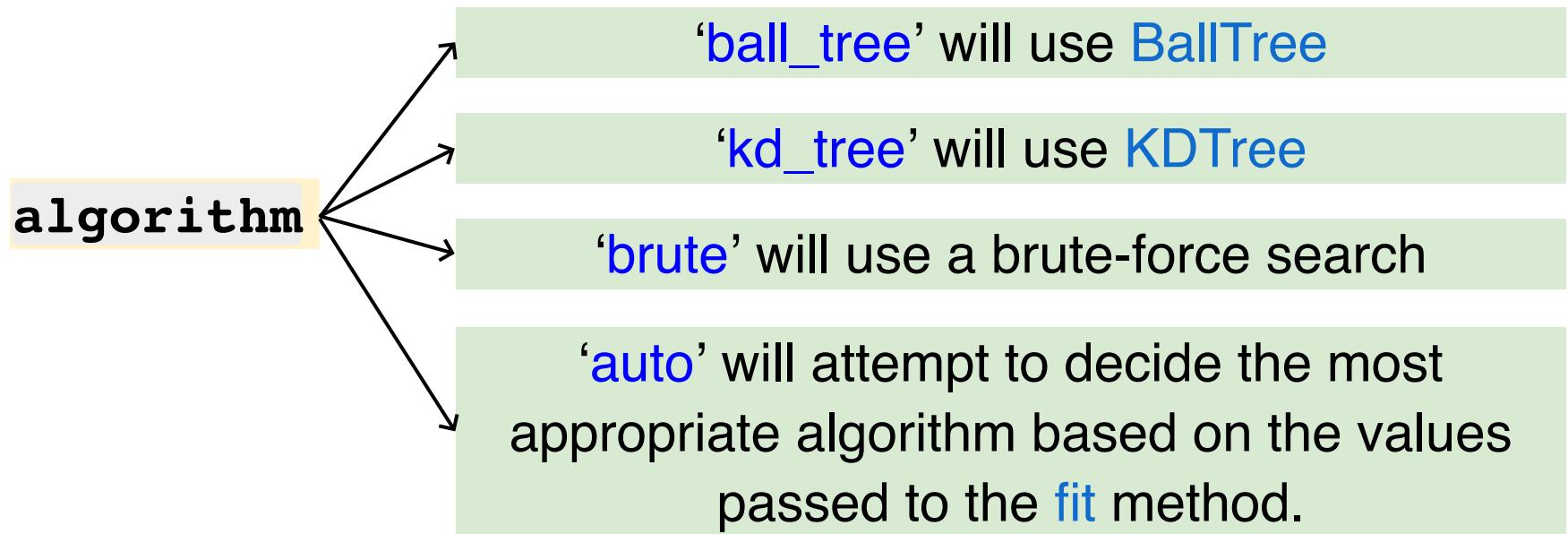
Can we define our own weight values for KNeighborsClassifier?

- Yes, it is possible if you have an array of distances.
- **weights** parameter also accepts a user-defined function which takes an array of distances as input, and returns an array of the same shape containing the weights.

Example:

```
1 def user_weights(weights_array):  
2     return weights_array  
3  
4 kneighbor_classifier = KNeighborsClassifier(weights=user_weights)
```

Which **algorithm** is used to compute the nearest neighbors in **KNeighborsClassifier**?



Default:

```
1 kneighbor_classifier = KNeighborsClassifier(algorithm='auto')
```

Some additional parameters for tree algorithm in KNeighborsClassifier?

For 'ball_tree' and 'kd_tree' algorithms, there are some other parameters to be set.

leaf_size

- can affect the speed of the construction and query, as well as the memory required to store the tree
- default = 30

metric

- Distance metric to use for the tree
- It is either string or callable function
 - some metrics are listed below:
 - “euclidean”, “manhattan”, “chebyshev”, “minkowski”, “wminkowski”, “seuclidean”, “mahalanobis”
 - default = 'minkowski'

p

- Power parameter for the Minkowski metric.
- default = 2

How do you apply RadiusNeighborsClassifier?

Step 1: Instantiate a `RadiusNeighborsClassifier` estimator without passing any arguments to it to create a classifier object.

```
1 from sklearn.neighbors import RadiusNeighborsClassifier  
2 radius_classifier = RadiusNeighborsClassifier()
```

Step 2: Call `fit` method on `RadiusNeighbors` classifier object with training feature matrix and label vector as arguments.

```
1 # Model training with feature matrix X_train and  
2 # label vector or matrix y_train  
3 radius_classifier.fit(X_train, y_train)
```

How do you specify the number of neighbors in RadiusNeighborsClassifier?

- The number of neighbors is specified within a fixed radius *r* of each training point using `radius` parameter.
- *r* is a float value.

```
1 radius_classifier = RadiusNeighborsClassifier(radius=1.0)
```

What is the default value of *r* ?

`r = 1.0`

Parameters for RadiusNeighborsClassifier

weights

'uniform'

'distance'

[callable]
function

default =
'uniform'

algorithm

'ball_tree'

'kd_tree'

'brute'

'auto'

default = 'auto'

leaf_size

default = 30

metric

default =
'minkowski'

p

default = 2

Support Vector Machines

Dr. Ashish Tendulkar

IIT Madras

Machine Learning Practice

- In this week, we will study how to implement support vector machines for classification tasks with `sklearn`.

Support Vector Machines

- Support Vector Machines (SVM) are a set of supervised learning methods used for classification, regression and outliers detection.
- SVM constructs a hyper-plane or set of hyper-planes in a high or infinite dimensional space, which can be used for classification, regression or other tasks.
- In `sklearn`, we have three methods to implement SVM.

SVC

NuSVC

LinearSVC

These are similar methods but, accept slightly different sets of parameters.
Implementation is based on `libsvm`.

Faster implementation of linear SVM classification with only linear kernel.
Implementation is based on `liblinear`.

Training data

Array X : holding the training samples

```
1 x = [[0, 0], [1, 1]]
```

shape → (n_samples, n_features)

Array y : holding the class labels (strings or integers)

```
1 y = [0, 1]
```

shape → (n_samples)

How to implement SVC (C-Support Vector Classification)?

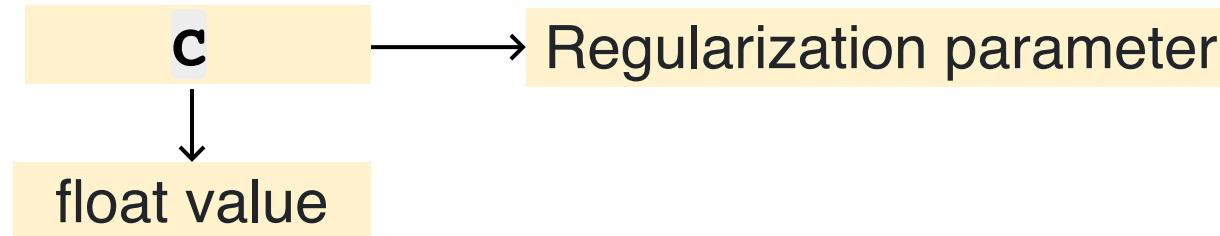
Step 1: Instantiate a **SVC** classifier estimator.

```
1 from sklearn.svm import SVC  
2 SVC_classifier = SVC()
```

Step 2: Call **fit** method on **SVC classifier object** with **training feature matrix** and **label vector** as arguments.

```
1 # Model training with feature matrix x_train and  
2 # label vector or matrix y_train  
3 SVC_classifier.fit(x_train, y_train)
```

How to perform regularization in SVC classifier?



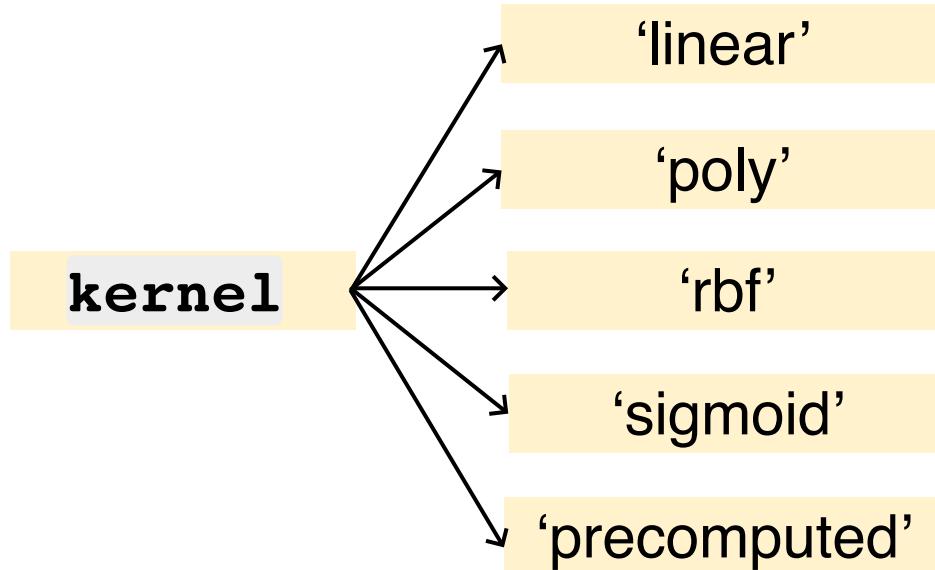
Default:

```
1 svc_classifier = SVC(C=1.0)
```

Note:

- strength of the regularization is inversely proportional to C
- strictly positive
- penalty is a squared l2 penalty

How to specify **kernel type** to be used in the algorithm ?



Default:

```
1 SVC_classifier = SVC(kernel = 'rbf')
```

- If **kernel = poly** , set **degree** (any integer value)
- If **kernel = callable** is given it is used to pre-compute the kernel matrix from data matrices

How to set kernel coefficient for 'rbf', 'poly' and 'sigmoid' kernels?

gamma

→	'scale'	value of gamma = $\frac{1}{\text{number of features} * \text{X.Var()}}$
→	'auto'	value of gamma = $\frac{1}{\text{number of features}}$
→	float value	

Default: 1 SVC_classifier = SVC(gamma = 'scale')

- If **kernel** = '**poly**' or '**sigmoid**' , set **coef0** which is an independent term in kernel function (any integer value)

How to view support vectors?

After the classifier is fit on the training data, there are few attributes which reveal the details of support vectors.

```
1 from sklearn.svm import SVC
2 SVC_classifier = SVC()
3 clf = SVC_classifier.fit(X_train, y_train)
4
5 #to view indices of the support vectors
6 clf.support_
7
8 #to view the support vectors
9 clf.support_vectors_
10
11 #to view the number of support vectors for each class
12 clf.n_support_
```

How to implement NuSVC (ν -Support Vector Classification)?

Step 1: Instantiate a NuSVC classifier estimator.

```
1 from sklearn.svm import NuSVC  
2 NuSVC_classifier = NuSVC()
```

Step 2: Call fit method on NuSVC classifier object with training feature matrix and label vector as arguments.

```
1 # Model training with feature matrix x_train and  
2 # label vector or matrix y_train  
3 NuSVC_classifier.fit(X_train, y_train)
```

What is the significance of ν in NuSVC?

Instead of C in SVC, ν is introduced in NuSVC to control the number of support vectors and margin errors.

ν is an upper bound on the fraction of margin errors and a lower bound of the fraction of support vectors.

Value of ν should $\in (0, 1]$

Default: $\nu = 0.5$

Other parameters for NuSVC are same as that of SVC.

How to implement `LinearSVC` (Linear Support Vector Classification)?

Step 1: Instantiate a `LinearSVC` classifier estimator.

```
1 from sklearn.svm import LinearSVC  
2 LinearSVC_classifier = LinearSVC()
```

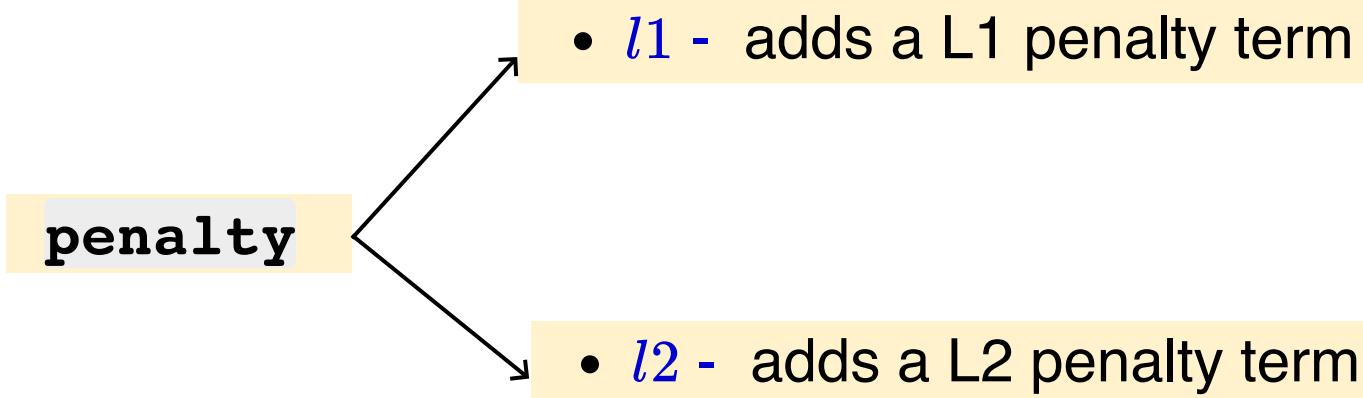
Step 2: Call `fit` method on `SVC` classifier object with `training feature matrix` and `label vector` as arguments.

```
1 # Model training with feature matrix x_train and  
2 # label vector or matrix y_train  
3 LinearSVC_classifier.fit(X_train, y_train)
```

Advantages of LinearSVC

- It has more flexibility in the choice of penalties and loss functions since it is implemented in terms of liblinear.
- Scales better to large numbers of samples.
- Supports both dense and sparse input.

How to provide **penalty** in LinearSVC classifier?

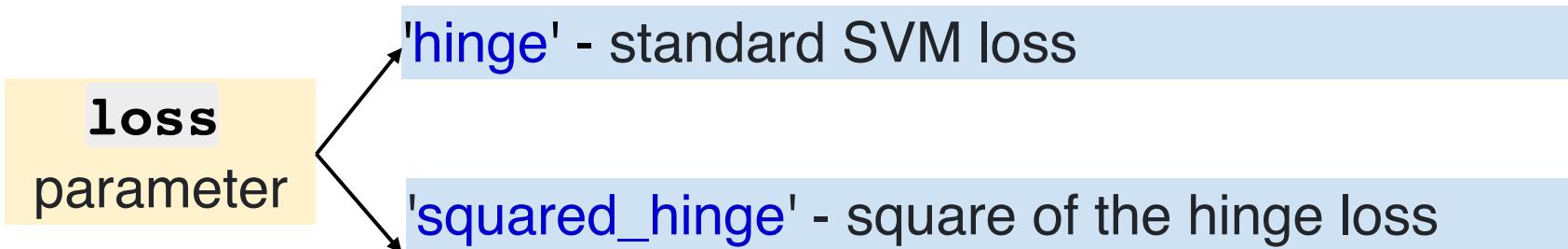


- *l1* - leads to **coef_** vectors that are sparse.

Default:

```
1 LinearSVC_classifier = Linear_SVC(penalty = 'l2')
```

How to choose **loss** functions in LinearSVC classifier?



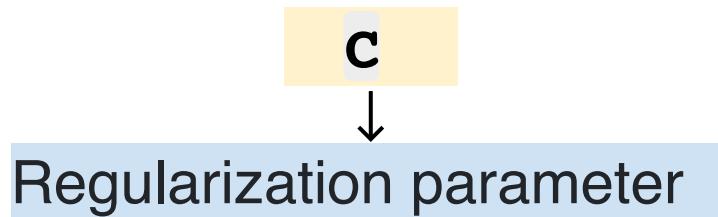
Default:

```
1 LinearSVC_classifier = Linear_SVC(loss = 'squared_hinge')
```

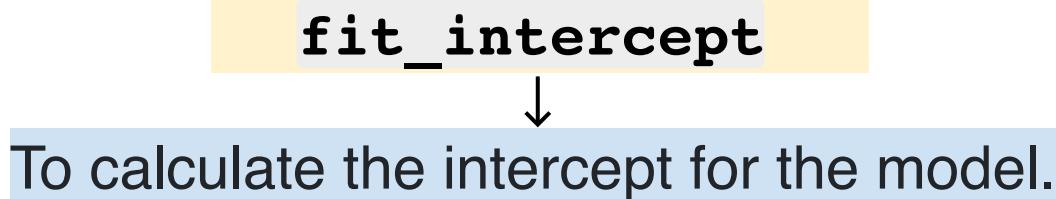
Combination not supported:

penalty='l1' and **loss='hinge'**

Some parameters in LinearSVC classifier



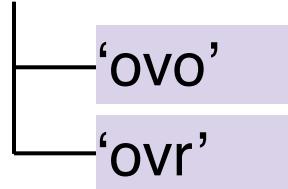
- Select the algorithm to either solve the dual or primal optimization problem.
- When n_samples > n_features, prefer dual=False.



How to perform multi-class classification using SVM?

- **SVC** and **NuSVC** implement the “**one-versus-one**” approach for multi-class classification.

decision_function_shape



- **LinearSVC** implements “**one-vs-the-rest**” approach for multi-class classification.

multi_class



Advantages of SVM

- Effective in high dimensional spaces.
- Effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel Functions can be specified for the decision function.

Disadvantages of SVM

- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation.
- Avoid over-fitting in choosing Kernel functions if the number of features is much greater than the number of samples.

Decision trees

Machine Learning Practice

Dr. Ashish Tendulkar

IIT Madras

Quick recap

- Non-parametric supervised learning methods.
- Can learn classification and regression models.
- Predicts label based on rules inferred from the features in the training set.

Tree algorithms

ID3

- ID3= Iterative Dichotomiser 3
- Creates a multiway tree

C4.5

- Successor to ID3
- Converts the trained trees into sets of if-then rules

C5.0

- Quinlan's latest version release under a proprietary license
- Uses less memory and builds smaller rulesets

CART

- Classification and Regression Trees
- Supports numerical target variables (regression) and does not compute rule sets

sklearn implementation of trees

scikit-learn uses an optimized version of the **CART algorithm**; however, it **does not support categorical variables** for now

Classification

`sklearn.tree.DecisionTreeClassifier`

Regression

`sklearn.tree.DecisionTreeRegressor`

Both these estimators have the same set of parameters
except for `criterion` used for tree splitting.

`splitter`

`max_depth`

`min_samples_split`

`min_samples_leaf`

sklearn tree parameters

splitter

Strategy for splitting at each node.

best

random

max_depth

Maximum depth of the tree.

int

When `None`, the tree expanded until all leaves are pure or they contain less than `min_samples_split` samples.

min_samples_split

int

float

The `minimum number of samples` required to `split an internal node`.

2

min_samples_leaf

int

float

The `minimum number of samples` required to be at a `leaf node`.

1

sklearn tree parameters

criterion

Specifies function to measure the quality of a split.

Classification

gini

entropy

Regression

squared_error

friedman_mse

absolute_error

poisson

Tree visualization

`sklearn.tree.plot_tree`

`decision_tree`

The decision tree to be plotted.

`max_depth`

The maximum depth of the representation. If `None`, the tree is fully generated.

`feature_names`

Names of each of the features.

`None`

`class_names`

Names of each of the target classes in ascending numerical order.

`None`

`label`

Whether to show informative labels for impurity.

`None`

Avoiding overfitting of trees

Pre-pruning

Uses hyper-parameter search like `GridSearchCV` for finding the best set of parameters.

Post-pruning

First grows trees without any constraints and then uses `cost_complexity_pruning` with `max_depth` and `min_samples_split`.

Tips for practical usage

- Decision trees tend to **overfit** data with a **large number of features**. Make sure that we have the **right ratio** of samples to number of features.
- Perform **dimensionality reduction** (PCA, or Feature Selection) on a data before using it for training the trees. It gives a better chance of finding discriminative features.
- **Visualize** the trained tree by using **max_depth=3** as an initial tree depth to get a feel for the fitment and then increase the depth.
- Balance the dataset before training to prevent the tree from being biased toward the classes that are dominant.

- Use `min_samples_split` or `min_samples_leaf` to ensure that multiple samples influence every decision in the tree, by controlling which splits will be considered.
 - A very small number will usually mean the tree will overfit.
 - A large number will prevent the tree from learning the data.

Bagging and Boosting

Machine Learning Practice

Dr. Ashish Tendulkar

IIT Madras

Part 2: Boosting

There are two boosting estimators:

- AdaBoost estimator
- Gradient boosting estimator

AdaBoost estimator

Class: `sklearn.ensemble.AdaBoostClassifier`

Class: `sklearn.ensemble.AdaBoostRegressor`

Class: `sklearn.ensemble.AdaBoostClassifier`

`base_estimator`

- Default estimator is `DecisionTreeClassifier` with `depth = 1`.

`n_estimators`

- Maximum number of estimators where boosting is terminated. The default value is 50.

`learning_rate`

- Weight applied to each classifier during boosting.
- Higher value here would increase contribution of individual classifiers.
- There is a trade-off between `n_estimators` and `learning_rate`.

Class: `sklearn.ensemble.AdaBoostRegressor`

`base_estimator`

- Default estimator is `DecisionTreeRegressor` with `depth = 3`.

`n_estimators`

- Maximum number of estimators where boosting is terminated. The default value is 50.

`learning_rate`

- Weight applied to each regressor at each boosting iteration.
- Higher value here would increase contribution of individual regressor.
- There is a trade-off between `n_estimators` and `learning_rate`.

The main parameters to tune to obtain good results are

- **n_estimators** and
- Complexity of the base estimators (e.g. its depth **max_depth** or **min_samples_split**).

Attributes of AdaBoost estimators

base_estimator_

Base estimator of ensemble.

estimators_

Collection of fitted sub-estimators.

estimator_weights_

Weights for each estimator in ensemble.

estimator_errors_

Errors for each estimator in ensemble.

Gradient boosting estimators

Class: `sklearn.ensemble.GradientBoostingClassifier`

Class: `sklearn.ensemble.GradientBoostingRegressor`

There are two most important parameters of these estimators:

- `n_estimators`
- `learning_rates`

`sklearn.ensemble.GradientBoostingClassifier` supports both binary and multiclass classification.

We will directly demonstrate XGBoost through colab demonstration.

Bagging and Boosting

Machine Learning Practice

Dr. Ashish Tendulkar

IIT Madras

Contents

Part 1: Voting, bagging and random forest

Part 2: Boosting and gradient boosting

Part 3: XGBoost

Voting estimators

Class: `sklearn.ensemble.VotingClassifier`

Class: `sklearn.ensemble.VotingRegressor`

Both these estimators take the following **common parameters**:

`base_estimator`

`weights`

Both these estimators implement the following **functions**:

`fit`

`predict`

`fit_transform`

`score`

`VotingClassifier` takes an **additional argument**:

`voting`

`hard`

`soft`

Bagging estimators

Class: `sklearn.ensemble.BaggingClassifier`

Class: `sklearn.ensemble.BaggingRegressor`

Common parameters

base_estimator

default=None

base estimator to fit on
random subsets of dataset
number of base estimators
in the ensemble

n_estimators

default=10

number of samples to
draw from X to train each
base estimator (**with**
replacement by default)

max_samples

default=1.0

number of samples to
draw from X to train each
base estimator (**without**
replacement by default)

max_features

default=1.0

bootstrap

default=True

Whether samples are
drawn with replacement

Common parameters

bootstrap_features

default=False

Whether features are drawn with replacement

oob_score

default=False

Whether to use out-of-bag samples to estimate generalization error

Random forest estimators

Class: `sklearn.ensemble.RandomForestClassifier`

Class: `sklearn.ensemble.RandomForestRegressor`

The parameters can be classified as

- Bagging parameters
- Decision tree parameters

Bagging parameters

- The number of trees are specified by `n_estimators` .
 - Default #trees for classification = 10
 - Default #trees for regression = 100
- `bootstrap` specifies whether to use bootstrap samples for training.
 - `True` : bootstrapped samples are used.
 - `False` : whole dataset is used.
- `oob_score` specifies whether to use out-of-bag samples for estimating generalization error. It is only available when `bootstrap` = `True` .

Bagging parameters

- `max_samples` specifies the number of samples to be drawn while bootstrapping.
 - `None` : Use all samples in the training data.
 - `int` : Use `max_samples` samples from the training data.
 - `float` : Use
 $\text{max_samples} * \text{total number of samples from training data}$
The value should be between 0 and 1.
- `random_state` controls randomness of features and samples selected during bootstrap.

- The number of features to be considered while splitting is specified by `max_features`.
 - `auto` , `sqrt` , `log2` , `int` , `float`

Value	max_features
<code>int</code>	value specified
<code>float</code>	value * # features
<code>auto</code>	$\text{sqrt}(\#\text{features})$
<code>sqrt</code>	$\text{sqrt}(\#\text{features})$
<code>log2</code>	$\text{log2}(\#\text{features})$
<code>None</code>	#features

Decision tree parameters

- The criteria for splitting the node is specified through `criterion`.
 - Default for classification: `gini`
 - Default for regression: `squared_error`
- The `depth of the tree` is controlled by `max_depth`. The default value is `None`, which means the tree will be `grown until all leaf nodes are pure or until leaves contain less than` `min_samples_splits` `samples`.
- We will continue to split the internal node until they contain `min_samples_splits` `samples`.
 - Whenever it is specified as an integer, then it is considered as a number.
 - Whenever it is specified as a float, and the `min_samples_splits` is calculated as `min_samples_splits × n`.

- The tree growth can also be controlled by `min_impurity_decrease` parameter.
 - A node will be split if it reduces impurity at least by the value specified in this parameter.
- The complexity of tree can also be controlled by `ccp_alpha` parameter through minimal cost complexity pruning procedure.

Trained random forest estimators

- `estimators_` member variable contains a collection of fitted estimators.
- `feature_importances_` member variable contains a list of important features.

Training and inference for random forest

- `fit` builds forest of trees from the training dataset with the specified parameters.
- `decision_path` returns decision path in the forest.
- `predict` returns class label in classification and output value in regression.
- `predict_proba` and `predict_log_proba` returns probabilities and their logs for classification set up.

Neural Networks

Dr. Ashish Tendulkar

IIT Madras

Machine Learning Practice

- In this week, we will study how to implement Multilayer Perceptron neural network models for classification and regression tasks with `sklearn`.

Multilayer Perceptron (MLP)

- It is a supervised learning algorithm.
- MLP learns a **non-linear function approximator** for either classification or regression depending on the given dataset.
- In **sklearn**, we implement MLP using:
 1. **MLPClassifier** for classification
 2. **MLPRegressor** for regression
- **MLPClassifier** supports **multi-class classification** by applying Softmax as the output function.
- It also supports **multi-label classification** in which a sample can belong to more than one class.
- **MLPRegressor** also supports **multi-output regression**, in which a sample can have more than one target.

Training data

Array X : holds the training samples



shape → (n_samples, n_features)

Array y : holds the target



shape → (n_samples,)

MLPClassifier

- How to implement MLPClassifier?

Step 1: Instantiate a **MLP** classifier estimator.

```
1 from sklearn.neural_network import MLPClassifier  
2 MLP_clf = MLPClassifier()
```

Step 2: Call **fit** method on **MLP classifier object** with **training feature matrix** and **label vector** as arguments.

```
1 # Model training with feature matrix x_train and  
2 # label vector or matrix y_train  
3 MLP_clf.fit(x_train, y_train)
```

MLPClassifier

Step 3: After fitting (training), the model can make predictions for new samples (X_{test}) using two methods:

```
1 MLP_clf.predict(X_test)  
2 MLP_clf.predict_proba(X_test)
```

predict



- gives labels for new samples
- for example:

```
array([1, 0])
```

predict_proba



- gives vector of probability estimates per sample
- for example:

```
array([1.967...e-04, 9.998...-01])
```

- MLPClassifier supports only the **Cross-Entropy loss function**

How to set the number of hidden layers?

hidden_layer_sizes

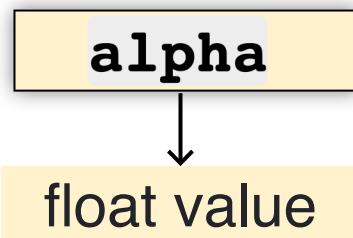
- This parameter sets the number of layers and the number of neurons in each layer.
- It is a **tuple** where **each element in the tuple represents the number of neurons at the i th position** where i is the index of the tuple.
- The **length of tuple** denotes the **total number of hidden layers** in the network.

To create a 3 hidden layer neural network with 15 neurons in first layer, 10 neurons in second layer and 5 neurons in third layer:

```
1 MLPClassifier(hidden_layer_sizes=(15,10,5))
```

How to perform regularization in MLPClassifier?

- The alpha parameter sets L2 penalty Regularization parameter



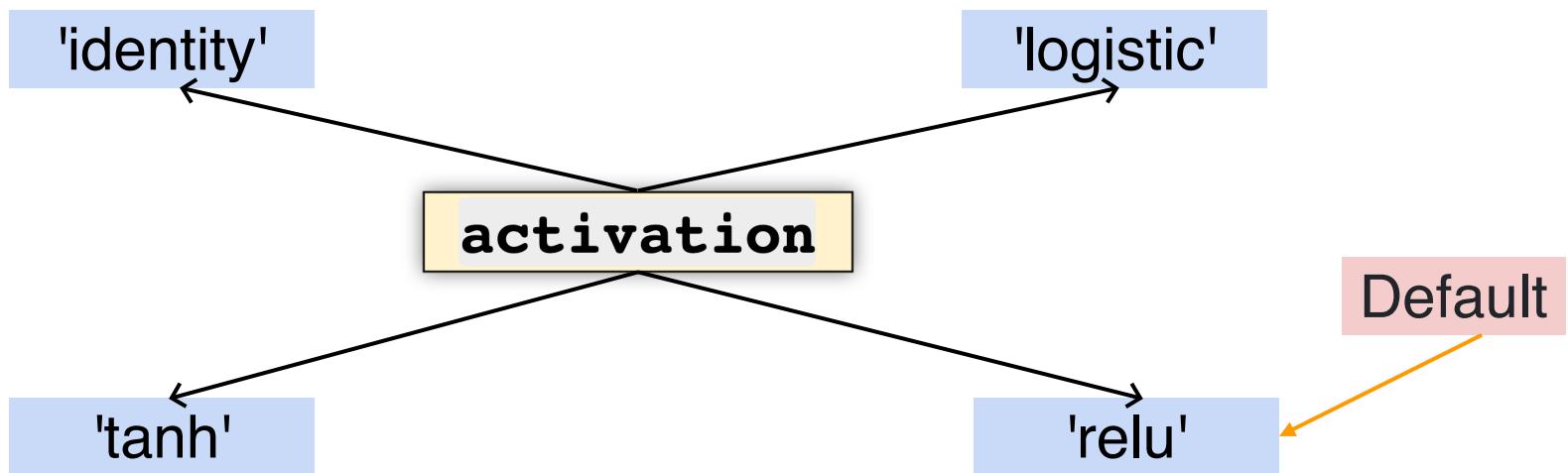
Default:

```
1 alpha = 0.0001
```

How to set the activation function for the hidden layers?

no-op activation
returns $f(x) = x$

logistic sigmoid function
returns $f(x) = \frac{1}{(1+exp(-x))}$

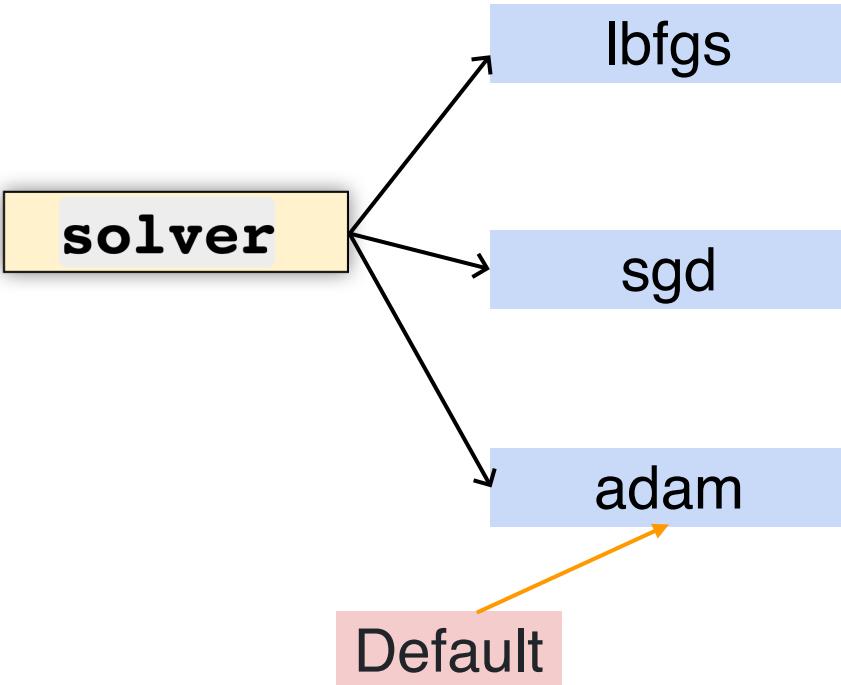


hyperbolic tan function
returns $f(x) = \tanh(x)$

rectified linear unit function
returns $f(x) = \max(0, x)$

How to perform **weight optimization** in MLPClassifier?

- MLPClassifier optimizes the log-loss function using LBFGS or stochastic gradient descent



- If the **solver** is 'lbfgs', the classifier will not use minibatch.
- Size of minibatches can be set to other stochastic optimizers: **batch_size** (int)

- default batch_size is 'auto'.

```
1 batch_size=min(200, n_samples)
```

How to view weight matrix coefficients of trained MLPClassifier?

coefs_

- It is a **list** of shape (n_layers - 1,)
- The i th element in the list represents the weight matrix corresponding to layer i .

Example:

- "weights between input and first hidden layer:"

```
1 print(MLP_clf.coefs_[0])
```

- "weights between first hidden and second hidden layer:"

```
1 print(MLP_clf.coefs_[1])
```

```
weights between input and first hidden layer:  
[[-0.14203691 -1.18304359 -0.85567518 -4.53250719 -0.60466275]  
 [-0.69781111 -3.5850093 -0.26436018 -4.39161248 0.06644423]]
```

```
weights between first hidden and second hidden layer:  
[[ 0.29179638 -0.14155284]  
 [ 4.02666592 -0.61556475]  
 [-0.51677234 0.51479708]  
 [ 7.37215202 -0.31936965]  
 [ 0.32920668 0.64428109]]
```

How to view bias vector of trained MLPClassifier?

intercepts_

- It is a **list** of shape (n_layers - 1,)
- The i th element in the list bias vector corresponding to layer $i + 1$.

Example:

- "Bias values for first hidden layer:"

```
1 print(MLP_clf.intercepts_[0])
```

- "Bias values for second hidden layer:"

```
1 print(MLP_clf.intercepts_[1])
```

```
Bias values for first hidden layer:  
[-0.14962269 -0.59232707 -0.54724811 7.02667699 -0.87510813]  
  
Bias values for second hidden layer:  
[-3.61417672 -0.76834882]
```

Some parameters in MLPClassifier

learning_rate

'constant'

'invscaling'

'adaptive'

default: 'constant'

learning_rate_init

float value

default: 0.001

power_t

float value

default: 0.5

max_iter

int value

default: 500

- `learning_rate` and `power_t` are used only for `solver = 'sgd'`
- `learning_rate_init` is used when `solver='sgd'` or '`adam`'.
- `shuffle` is used to shuffle samples in each iteration when
`solver='sgd' or 'adam'`
- `momentum` is used for gradient descent update when `solver='sgd'`

MLPRegressor

- MLPRegressor trains using backpropagation with no activation function in the output layer.
- Therefore, it uses the **square error as the loss function**, and the **output is a set of continuous values**.

The parameters of MLPRegressor are the same as that of MLPClassifier.

How to implement MLPRegressor?

Step 1: Instantiate a **MLP** regressor estimator.

```
1 from sklearn.neural_network import MLPRegressor  
2 MLP_reg = MLPRegressor()
```

Step 2: Call **fit** method on **MLP regressor object** with training feature matrix and label vector as arguments.

```
1 # Model training with feature matrix x_train and  
2 # label vector or matrix y_train  
3 MLP_reg.fit(x_train, y_train)
```

Step 3: After fitting (training), the model can make predictions for new samples (X_test):

```
1 MLP_reg.predict(X_test)
```

- returns predicted values for new samples
- for example:
array([-0.9..., -7.1...])

```
1 MLP_reg.score(X_test,y_test)
```

- returns R^2 score
- for example:
0.45678889